Создай свой собственный язык программирования

Клинтон Л. Джеффери

ДМК издательство

Клинтон Л. Джеффери

Создай свой собственный язык программирования

Создай свой собственный язык программирования

Руководство программиста по разработке компиляторов, интерпретаторов и доменно-ориентированных языков для решения современных вычислительных задач

Клинтон Л. Джеффери

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
Москва, 2023

УДК 004.42
ББК 32.372
Д40

Клинтон Л. Джеффери

Д40 Создай свой собственный язык программирования. Руководство программиста по разработке компиляторов, интерпретаторов и доменно-ориентированных языков для решения современных вычислительных задач / пер. с англ. С. В. Минца. – М.: ДМК Пресс, 2022. – 408 с.: ил.

ISBN 978-5-93700-140-5

Книга рассказывает о том, как разрабатывать уникальные языки программирования, чтобы сократить время и стоимость создания приложений для новых или специализированных областей применения вычислительной техники. Вы начнете с реализации интерфейса компилятора для вашего языка, включая лексический и синтаксический анализатор, а к концу чтения сможете разрабатывать и воплощать в коде свои собственные языки, позволяющие компилировать и запускать программы.

Издание адресовано разработчикам программного обеспечения, заинтересованным в создании собственного языка. Для изучения материала потребуется опыт программирования на языке высокого уровня, таком как Java или C++.

УДК 004.42
ББК 32.372

First published in the English language under the title 'Build Your Own Programming Language' – (9781800204805)

ISBN (анг.) 978-1-80020-480-5
ISBN (рус.) 978-5-93700-140-5

Памяти переводчика

Семена Викторовича Минца,

просто очень хорошего человека

Эта книга – его последний перевод.

С Семеном было очень приятно работать – он был немногословен и деловит.

Отлично знал информатику и программирование и перевел для нас «Введение в логическое программирование», «Объяснимые модели искусственного интеллекта на Python», «Искусство неизменяемой архитектуры» и эту последнюю.

Он ушел слишком несправедливо рано, мог бы еще многое сделать.

Будет не хватать его. Людей, особенно талантливых, заменить невозможно.

Заместитель главного редактора
Сенченкова Елена

Оглавление

Эта книга посвящается Сьюзи, Кертису, Кэри и всем, кто создает свои собственные языки программирования.

Клинтон Л. Джеффери

Об авторах

Клинтон Л. Джеффри – профессор и заведующий кафедрой компьютерных наук и инженерии Горно-технологического института Нью-Мексико. Он получил степень бакалавра в Вашингтонском университете, а также степень магистра и доктора философии в Университете Аризоны в области компьютерных наук. Проводил исследования и написал много книг и статей по языкам программирования, мониторингу программ, отладке, графике, виртуальным средам и визуализации. Вместе с коллегами изобрел язык программирования Unicon.

О рецензентах

Филлип Ли – доброволец Корпуса мира в Сараваке, Малайзия. Он получил степень бакалавра в Университете штата Орегон, магистра, докторскую степень в Университете Вашингтона, степень магистра в области малайской/индонезийской литературы в Университете Малайзии и степень магистра в области вычислительной техники в Университете Мердока в Перте. Преподавал для студентов и аспирантов в Оклендском университете и Университете Мердока. У Филиппа есть публикации по латинской, греческой, малайской и индонезийской литературе. Он является сопрограммистом библиотеки Конгресса *thomas.loc.gov*, поисковой системы Конгресса Национальной медицинской библиотеки *toxnet.nlm.nih.gov*. Кроме того, трудится разработчиком программ анализа текста для англо-иранского словаря Фонда Тун Джуга.

Стив Уамплер получил степень доктора философии в области компьютерных наук в Университете Аризоны. После чего он был адъюнкт-профессором компьютерных наук с 1981 по 1993 год. Стив работал разработчиком программного обеспечения в нескольких крупных проектах телескопов, включая проект Gemini 8m Telescopes Project и Солнечный телескоп Daniel K Inouye, в рамках Ассоциации исследований в области астрономии. Наряду с этим он был рецензентом программного обеспечения для ряда крупных телескопов, в том числе LSST, TMT, GMT, Keck, VLT ESO и GTC. Стив был техническим рецензентом первого издания книги Марка Собелла «Практическое руководство по операционной системе Linux», 1997 год.

Предисловие

После 60 лет высокоуровневой разработки языков программирование все еще остается сложным. Спрос на программное обеспечение постоянно увеличивающегося объема и сложности реализации резко возрос из-за аппаратных достижений, в то время как языки программирования совершенствуются гораздо медленнее. Создание новых языков для конкретных целей – одно из противоядий от кризиса программного обеспечения.

Эта книга посвящена созданию новых языков программирования. Вводится тема проектирования языка программирования, хотя основной акцент делается на реализации языка программирования. В рамках этой интенсивно изучаемой темы новым аспектом данной книги является слияние традиционных инструментов компиляции (Flex и Byacc) с двумя языками реализации более высокого уровня. Язык очень высокого уровня (Unicon) обрабатывает структуры данных и алгоритмы компилятора, как нож масло, в то время как основной современный язык (Java) показывает, как реализовать тот же код в более типичной производственной среде.

Для кого эта книга

Эта книга предназначена для разработчиков программного обеспечения, заинтересованных в идее создания собственного языка или разработки языка, специфичного для конкретной предметной области. Студенты, изучающие информатику на курсах построения компиляторов, также найдут эту книгу весьма полезной в качестве практического руководства по реализации языка в дополнение к другим теоретическим учебникам. Чтобы извлечь максимальную пользу из данной книги, требуются знания среднего уровня и опыт работы с языком высокого уровня, таким как Java или C++.

Что скрывает обложка

В *главе 1 «Зачем создавать другой язык программирования?»* обсуждается, когда следует создавать язык программирования, а когда вместо этого создавать библиотеку функций или библиотеку классов. Многие читатели этой книги уже знают, что они хотят создать свой собственный язык программирования. Некоторые должны вместо этого создать библиотеку.

Глава 2 «Проектирование языка программирования» описывает, как точно определить язык программирования, что важно знать, прежде чем пытаться создать язык программирования. Это включает в себя разработку лексических и синтаксических особенностей языка, а также его семантики. Хорошие языковые проекты обычно используют как можно больше знакомого синтаксиса.

Глава 3 «Сканирование исходного кода» представляет лексический анализ, включая регулярные обозначения выражений и инструменты Ulex и JFlex.

В конце вы будете открывать файлы исходного кода, читать их символ за символом и сообщать об их содержимом в виде потока токенов, состоящих из отдельных слов, операторов и знаков препинания в исходном файле.

В *главе 4 «Синтаксический анализ»* представлен синтаксический анализ, включая контекстно-свободные грамматики и инструменты iyacc и byacc/j. Вы узнаете, как отлаживать проблемы в грамматиках, которые препятствуют синтаксическому анализу, и сообщать о синтаксических ошибках, когда они возникают.

В *главе 5 «Синтаксические деревья»* рассматриваются синтаксические деревья. Основным побочным продуктом процесса синтаксического анализа является построение древовидной структуры данных, которая представляет логическую структуру исходного кода. Построение узлов дерева происходит в семантических действиях, которые выполняются для каждого правила грамматики.

В *главе 6 «Таблицы символов»* показано, как создавать таблицы символов, вставлять в них символы и использовать таблицы для выявления двух видов семантических ошибок: необъявленных и незаконно повторно объявленных переменных. Чтобы понять ссылки на переменные в исполняемом коде, необходимо отслеживать область действия и время жизни каждой переменной. Это достигается с помощью табличных структур данных, которые являются вспомогательными для синтаксического дерева.

Глава 7 «Проверка базовых типов» посвящена проверке типов, которая является основной задачей, требуемой в большинстве языков программирования. Проверка типов может выполняться во время компиляции или во время выполнения. В этой главе рассматривается общий случай статической проверки типов во время компиляции для базовых типов, также называемых атомарными, или скалярными, типами.

В *главе 8 «Проверка типов массивов, вызовов методов и доступа к структурам»* показано, как выполнять проверку типов массивов, параметров и возвращаемых типов вызовов методов в подмножестве Java Jzero. Более сложные части проверки типов – это когда должны быть проверены несколько массовов или составные массивы.

Глава 9 «Генерация промежуточного кода» показывает вам, как генерировать промежуточный код, рассматривая примеры для языка Jzero. Прежде чем сгенерировать код для выполнения, большинство компиляторов превращают синтаксическое дерево в список машинно независимых инструкций промежуточного кода. На этом этапе обрабатываются ключевые аспекты потока управления, такие как генерация меток и инструкции goto.

В *главе 10 «Раскрашивание синтаксиса в среде IDE»* рассматривается задача включения информации из синтаксического анализа в среду IDE, чтобы обеспечить раскрашивание синтаксиса и визуальную обратную связь о синтаксических ошибках. Язык программирования требует большего, чем просто компилятор или интерпретатор, – он требует экосистемы инструментов для разработчиков. Эта экосистема может включать в себя отладчики, интерактивную справку или интегрированную среду разработки. Эта глава представляет собой пример Unicon, взятый из среды разработки Unicon IDE.

Глава 11 «Интерпретаторы байт-кода» посвящена разработке набора команд и интерпретатора, который выполняет байт-код. Новый язык, специфичный для конкретной предметной области, может включать в себя высокоуровневые функ-

ции программирования предметной области, которые напрямую не поддерживаются основными процессорами. Наиболее практичным способом генерации кода для многих языков является генерация байт-кода для абстрактной машины, набор команд которой напрямую поддерживает целевое назначение языка с последующим выполнением этой программы путем интерпретации команд.

Глава 12 «Генерация байт-кода» рассматривает прохождение по гигантскому связанному списку, перевод каждой инструкции промежуточного кода в одну или несколько инструкций байт-кода. Как правило, это цикл для обхода связанного списка с разным фрагментом кода для каждой промежуточной кодовой инструкции.

Глава 13 «Генерация собственного кода» содержит обзор генерации собственного кода для x86_64. Некоторые языки программирования требуют собственного кода для достижения своих требований к производительности. Генерация собственного кода похожа на генерацию байт-кода, но более сложна, включает в себя выделение регистров и режимы адресации памяти.

Глава 14 «Реализация операторов и встроенных функций» описывает, как поддерживать языковые функции очень высокого уровня и специфичные для предметной области, добавляя операторы и функции, встроенные в язык. Языковые возможности очень высокого уровня и специфичные для предметной области часто лучше всего представлены операторами и функциями, встроенными в язык, а не библиотечными функциями. Добавление встроенных модулей может упростить ваш язык, улучшить его.

Глава 15 «Структуры управления доменом» описывает, когда вам нужна новая структура управления, и предоставляет примеры структур управления, которые обрабатывают текст с помощью сканирования строк и отображают графические области. Общий код в предыдущих главах охватывал основные условные и циклические структуры управления, но языки, зависящие от предметной области, часто имеют уникальную или настраиваемую семантику, для которой они вводят новые структуры управления. Добавление новых структур управления существенно сложнее, чем добавление новой функции или оператора, но именно это делает языки, специфичные для предметной области, достойными разработки, а не просто написания библиотек классов.

В *главе 16 «Сборка мусора»* представлена пара методов, с помощью которых вы можете реализовать сборку мусора на вашем языке. Управление памятью является одним из наиболее важных аспектов современных языков программирования, и все классные языки программирования имеют функцию автоматического управления памятью с помощью сборки мусора. В этой главе приводится несколько вариантов того, как вы могли бы реализовать сборку мусора на вашем языке, включая подсчет ссылок и сборку мусора с пометкой и разверткой.

Глава 17 «Заключительные мысли» отражает основные темы, представленные в книге, и дает вам некоторую пищу для размышлений. В ней рассматривается то, что было извлечено из написания этой книги, и дается множество рекомендаций для дальнейшего чтения.

Приложение «Unicon Essentials» описывает язык программирования Unicon в достаточном количестве, чтобы понять те примеры в этой книге, которые находятся в Unicon. Большинство примеров приведены рядом на Unicon и Java, но версии Unicon обычно короче и легче читаются.

Как получить от этой книги максимальную пользу

Чтобы понять эту книгу, вы должны быть программистом среднего уровня на Java или подобном языке; программист C, который знает объектно-ориентированный язык, подойдет.

Программное обеспечение, упомянутое в книге	Необходимая операционная система
Unicon 13.2, Uflex,	Windows, Linux
Java, Jflex, and Byacc/J	
GNU Make	

Инструкции по установке и использованию инструментов немного расширены, чтобы сократить время запуска, и приведены в главе 3 «Сканирование исходного кода» и главе 5 «Синтаксические деревья». Если вы технически одарены, вы, возможно, сможете запустить все эти инструменты на macOS, но во время написания этой книги они не использовались и не тестировались.

> **Примечание**
>
> Если вы используете цифровую версию этой книги, мы советуем вам ввести код самостоятельно или получить доступ к коду из репозитория книги на GitHub (ссылка доступна в следующем разделе). Это поможет вам избежать любых потенциальных ошибок, связанных с копированием и вставкой кода.

Загрузка примеров

Примеры для данной книге вы можете загрузить на сайте нашего издательства по ссылке: *https://dmkpress.com/catalog/computer/programming/978-5-93700-140-5/*.

Видео

Видеоролики с кодом в действии для этой книги можно посмотреть по адресу *https://bit.ly/3njc15D*.

Цветные иллюстрации

Цветные иллюстрации к данной книге вы можете загрузить на сайте нашего издательства по ссылке: *https://dmkpress.com/catalog/computer/programming/978-5-93700-140-5/*.

Используемые сокращения

Вот несколько текстовых условных обозначений, используемых на протяжении всей этой книги.

Код в тексте: указывает кодовые слова в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, пути, фиктивные URL-

адреса, вводимые пользователем, и дескрипторы Twitter. Вот пример: «Соответствующий Java main() должен быть помещен в класс».

Блок кода задается следующим образом:

```
procedure main(argv)
procedure main(argv)
 yyin := open(argv[1])
 yyin := open(argv[1]) yyin := open(argv[1])
 yyin := open(argv[1])
 yyin := open(argv[1])
```

Когда мы хотим привлечь ваше внимание к определенной части блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
procedure main(argv)
procedure main(argv)
 yyin := open(argv[1])
 yyin := open(argv[1]) yyin := open(argv[1])
 yyin := open(argv[1])
```

Ввод или вывод из командной строки записывается так:

```
procedure main(argv)
 yyin := open(argv[1])
```

Жирный шрифт: обозначает новый термин, важное слово или слова, которые вы видите на экране. Например, слова в меню или диалоговых окнах выделены **жирным** шрифтом. Вот пример: «Выберите **Информацию о системе** на панели администрирования».

> **Советы или важные примечания**
>
> Представляют собой текст, помещенный в рамку.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии данной книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу *dmkpress@gmail.com*, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Packt очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно

выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты *dmkpress@gmail.com* со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Часть I

Интерфейсы языка программирования

В этой части вы создадите базовую конструкцию языка и реализуете интерфейс компилятора для него, включая лексический анализатор и синтаксический анализатор, который строит синтаксическое дерево из входного исходного файла.

Эта часть включает в себя следующие главы:

- глава 1 «Зачем создавать еще один язык программирования»;
- глава 2 «Проектирование языка программирования»;
- глава 3 «Сканирование исходного кода»;
- глава 4 «Парсинг»;
- глава 5 «Деревья синтаксиса».

Глава 1

Зачем создавать еще один язык программирования?

Эта книга покажет вам, как создать свой собственный язык программирования, но сначала вы должны спросить себя: зачем мне это нужно? Для некоторых из вас ответ будет простым – потому что это очень весело. Однако для остальных создание языка программирования – это большой труд, и вы должны быть уверены в необходимости этого, прежде чем начинать. Хватит ли у вас терпения и настойчивости?

В этой главе приведены несколько веских причин для создания собственного языка программирования, а также некоторые ситуации, когда вам не обязательно создавать задуманный язык. В конце концов, разработка библиотеки классов для вашей области применения может быть более простой и столь же эффективной. Однако у библиотек есть свои недостатки, и иногда подходит только новый язык.

После этой главы, в остальной части этой книги, тщательно все обдумав, вы примете как должное, что решили создать язык. В этом случае вы должны определить некоторые требования к вашему языку. Но сначала мы рассмотрим следующие основные темы данной главы:

- мотивы для написания собственного языка программирования;
- различие между языками программирования и библиотеками;
- применимость инструментов языка программирования к другим программным проектам;
- определение требований к языку;
- тематическое исследование, в котором обсуждаются требования к языку Unicon.

Давайте начнем с мотивации.

Итак, вы хотите создать свой собственный язык программирования...

Конечно, некоторые изобретатели языков программирования – это рок-звезды компьютерной науки, такие как Деннис Ричи (Dennis Ritchie) или Гвидо ван Россум (Guido van Rossum)! Но стать рок-звездой компьютерной науки было проще в те времена. Давным-давно я услышал следующее сообщение от од-

ного из участников второй конференции по истории языков программирования: «*Все сошлись во мнении, что область языков программирования мертва. Все важные языки уже изобретены*». Это было признано в корне ошибочным через год или два, когда на сцену вышел Java, и, возможно, десятки раз с тех пор, как появились такие языки, как Go. Спустя всего шесть десятилетий было бы неразумно утверждать, что наша область является зрелой и что нет ничего нового, что можно было бы изобрести, чтобы стать знаменитым.

Тем не менее знаменитость – это плохая причина для создания языка программирования. Шансы приобрести славу или богатство благодаря изобретению языка программирования невелики. Любопытство и желание узнать, как все устроено, являются вескими причинами, если у вас есть время и склонность, но, возможно, лучшими причинами для создания собственного языка программирования являются необходимость и потребность.

Некоторым необходимо создать новый язык или новую реализацию существующего языка программирования для нового процессора или для соперничества с конкурирующей компанией. Если это не вы, то, возможно, вы изучили лучшие языки (и компиляторы или интерпретаторы), доступные для какой-то области, в которой вы разрабатываете программы, и для того, что вы делаете, не хватает некоторых ключевых функций, и эти недостающие функции причиняют вам боль. Время от времени кто-то придумывает совершенно новый стиль вычислений, для которого в соответствии с новой парадигмой программирования требуется новый язык.

Пока мы обсуждаем ваши мотивы для создания языка, давайте поговорим о различных видах языков, организации и примерах, которые будут использованы в этой книге. Каждая из данных тем заслуживает внимания.

Типы реализации языков программирования

Какими бы ни были ваши причины, прежде чем создавать язык программирования, вы должны выбрать лучшие инструменты и технологии, которые можно найти для выполнения этой работы. Эта книга подберет их для вас. Во-первых, возникает вопрос о языке реализации, на котором вы создаете свой язык. Академики, изучающие языки программирования, любят хвастаться тем, что пишут свой язык на этом самом языке, но обычно это лишь полуправда (или кто-то был очень непрактичным и в то же время выпендривался). Существует также вопрос о том, какую реализацию языка программирования строить:

- чистый *интерпретатор*, который сам выполняет исходный код;
- *собственный компилятор* и среду выполнения, как, например, в C;
- *транспайлер*, который переводит ваш язык на другой язык высокого уровня;
- *компилятор байт-кода* с сопутствующей машиной байт-кода, например Java.

Первый вариант веселый, но обычно слишком медленный. Второй вариант – самый лучший, но обычно он слишком трудоемкий. Хороший собственный компилятор может потребовать многих человеко-лет усилий.

Хотя третий вариант, безусловно, самый простой и, вероятно, самый веселый, и я уже с успехом использовал его, но если это не прототип, то это своего

рода обман. Конечно, первая версия C++ была транспайлером, но она уступила место компиляторам, и не только потому, что была глючной. Странно, но генерация высокоуровневого кода, кажется, делает ваш язык еще более зависимым от базового языка, чем другие варианты, а языки – это движущиеся мишени. Хорошие языки умерли, потому что их базовые зависимости исчезли или непоправимо отказали. Это может быть смерть от тысячи мелких порезов.

В этой книге выбран четвертый вариант: мы построим компилятор байт-кода с сопутствующей машиной байт-кода, потому что это обеспечивает наибольшую гибкость и притом обладает достойной производительностью. Глава о компиляции собственного кода включена для тех, кому требуется максимально быстрое выполнение.

Понятие машины байт-кода очень старое. Оно стало известным благодаря реализации Pascal в Калифорнийском университете в Сан-Диего (University of California, San Diego – UCSD) и классической реализации SmallTalk-80[1] среди прочих. Оно стало повсеместным и вошло в повседневный английский язык с появлением JVM в Java. Машины байт-кода – это абстрактные процессоры, интерпретируемые программным обеспечением, их часто называют *виртуальными машинами* (как в *Java Virtual Machine*), хотя я не буду использовать данную терминологию, потому что она применяется для обозначения программных средств, использующих реальные наборы инструкций аппаратного обеспечения, таких, например, как классические платформы IBM или более современные инструменты, допустим *Virtual Box*.

Машина байт-кода обычно имеет более высокий уровень, чем часть аппаратного обеспечения, поэтому реализация байт-кода обеспечивает большую гибкость. Давайте вкратце рассмотрим, что для этого нужно...

Организация реализации языка байт-кода

В значительной степени организация этой книги соответствует классической организации компилятора байт-кода и соответствующей виртуальной машины. Эти компоненты определены ниже, а затем приведена диаграмма для их обобщения:

- *лексический анализатор* (сканер) считывает символы исходного кода и определяет, как они сгруппированы в последовательность слов или токенов;
- *синтаксический анализатор* (парсер) считывает последовательность токенов и определяет, является ли эта последовательность допустимой в соответствии с грамматикой языка. Если токены расположены в законном порядке, то получается дерево синтаксиса;
- *семантический анализатор* проверяет, что все используемые имена являются законными для тех операций, в которых они используются. Он проверяет их типы, чтобы точно определить, какие операции выпол-

[1] Smalltalk-80 – объектно-ориентированный язык программирования, разработанный в 1970-х годах. Представляет собой интегрированную среду разработки и исполнения, программирование в которой в итоге сводится к модификации ее собственного поведения. – *Прим. перев.*

няются. Все эти проверки делают дерево синтаксиса толстым, отягощенным дополнительной информацией о том, где объявлены переменные и каковы их типы;

- *генератор промежуточного кода* определяет места в памяти для всех переменных и всех мест, где ход выполнения программы может резко измениться, например циклов и вызовов функций. Он добавляет их в дерево синтаксиса, а затем проходит по этому еще более толстому дереву, прежде чем построить список машинно независимых инструкций промежуточного кода;
- *генератор окончательного кода* превращает список инструкций промежуточного кода в фактический байт-код в формате файла, который будет эффективно загружаться и выполняться.

Для загрузки и выполнения программ независимо от шагов компилятора виртуальной машины байт-кода написан интерпретатор байт-кода. Он представляет собой гигантский цикл с оператором `switch`. Для экзотических языков программирования компилятор может не иметь большого значения, и вся магия будет происходить в интерпретаторе байт-кода. Всю организацию можно обобщить схемой, показанной на рис. 1.1.

Потребуется много кода, чтобы проиллюстрировать построение машинной реализации байт-кода языка программирования. То, как представлен этот код, важно, и много информации вы можете почерпнуть как раз из нашей книги.

Языки, используемые в примерах

В этой книге приводятся примеры кода на двух языках с использованием модели параллельных переводов. Первый язык – *Java*, потому что этот язык повсеместно распространен. Надеемся, что вы знаете его или *C++* и сможете прочитать примеры со средним уровнем владения. Второй язык примеров – это собственный язык автора, *Unicon*. Читая эту книгу, вы сможете сами оценить, какой язык лучше подходит для создания вашего собственного языка программирования. Как можно больше примеров будет приведено на обоих языках, и примеры на двух языках будут написаны максимально одинаково. Иногда предпочтение будет отдано меньшему языку.

Различия между Java и Unicon будут очевидны, но они несколько уменьшаются благодаря инструментам построения компилятора, которые мы будем использовать. А использовать мы будем современных потомков старинных инструментов Lex и YACC для генерации нашего сканера и парсера и придерживаться инструментов для Java и Unicon, которые остаются максимально совместимыми с оригинальными Lex и YACC. В результате фронтенды нашего компилятора будут практически идентичны в обоих языках. Lex и YACC – это языки декларативного программирования, которые решают некоторые из наших трудных проблем на еще более высоком уровне, чем Java или Unicon.

Пока мы применяем Java и Unicon в качестве языков реализации, нам придется поговорить еще об одном языке – о языке примеров, который мы создаем. Он является заменой для любого языка, который вы решите создать. Несколько произвольно я введу для этой цели язык под названием *Jzero*. Никлаус Вирт (Niklaus Wirth) придумал игрушечный язык под названием *PL/0* (нулевой язык

программирования – *programming language zero*, название является аналогом названия языка PL/1), который использовался на курсах по созданию компиляторов. Jzero будет крошечным подмножеством Java, которое служит для аналогичной цели. Я довольно *усердно искал* (то есть гуглил *Jzero*, а затем *Jzero compiler*), не публиковал ли кто-нибудь определение Jzero, которое мы могли бы использовать, и не обнаружил этого, так что мы просто придумаем его по ходу дела.

Примеры Java в этой книге будут протестированы с использованием *OpenJDK 14*, возможно, другие версии Java (например, OpenJDK 12 или Oracle Java JDK) будут работать так же, а возможно, и нет. Вы можете получить OpenJDK на сайте *http://openjdk.java.net*, или, если вы работаете в Linux, ваша операционная система, вероятно, имеет пакет OpenJDK, который вы можете установить. Дополнительные инструменты для построения языка программирования (Jflex и byacc/j), которые требуются для примеров Java, будут представлены в последующих главах по мере их использования. Реализации Java, которые мы будем поддерживать, могут быть более ограничены тем, в каких версиях будут работать эти средства построения языка, чем чем-либо другим.

Примеры Unicon, приведенные в этой книге, работают с Unicon версии 13.2, которую можно получить на сайте *http://unicon.org*. Чтобы установить Unicon на Windows, необходимо загрузить файл **.msi** и запустить инсталлятор. Для установки на Linux обычно делают git-клон исходных текстов и вводят `make`. Затем нужно добавить каталог *unicon/bin* в **PATH**:

```
git clone git://git.code.sf.net/p/unicon/unicon
make
```

Пройдя нашу организацию и реализацию, которая будет использоваться в этой книге, возможно, стоит еще раз взглянуть на то, когда необходим язык программирования и когда можно обойтись без него, разработав вместо него библиотеку.

Язык и библиотека – в чем разница?

Не создавайте язык программирования, если с этой задачей справится библиотека. Библиотеки являются наиболее распространенным способом расширения существующего языка программирования для выполнения новой задачи. Библиотека – это набор функций или классов, которые могут быть использованы совместно для написания приложений для некоторых аппаратных или программных технологий. Многие языки, включая C и Java, почти полностью разработаны для того, чтобы использовать богатый набор библиотек. Язык сам по себе очень простой и общий, в то время как большая часть того, что разработчик должен изучить для создания приложений, заключается в том, как использовать различные библиотеки.

Библиотеки могут делать следующее:

- представлять новые типы данных (классы) и предоставлять публичные функции (API) для управления ими;
- предоставлять уровень абстракции поверх набора вызовов аппаратного обеспечения или операционной системы.

Что не могут делать библиотеки:

- вводить новые управляющие структуры и синтаксис для поддержки новых областей применения;
- встраивать/поддерживать новую семантику в существующей среде выполнения языка.

Библиотеки делают некоторые вещи плохо, в том смысле, что в конечном счете вы можете предпочесть создать новый язык:

- библиотеки часто становятся больше и сложнее, чем нужно;
- библиотеки могут иметь еще более крутые кривые обучения и более скудную документацию, чем языки;
- время от времени библиотеки конфликтуют с другими библиотеками, а несовместимость версий часто повреждает приложения, использующие библиотеки.

Существует естественный эволюционный путь от библиотеки к языку. Разумный подход к созданию нового языка для поддержки прикладной области заключается в том, чтобы начать с создания или покупки лучшей библиотеки, доступной для этой прикладной области. Если результат не удовлетворяет вашим требованиям с точки зрения поддержки области и упрощения написания программ для данной области, то у вас есть веские аргументы в пользу нового языка.

Эта книга о создании собственного языка, а не только о создании собственной библиотеки. Оказывается, изучение этих инструментов и методов полезно и в других контекстах.

Применимость к другим задачам разработки программного обеспечения

Инструменты и технологии, о которых вы узнаете в процессе создания собственного языка программирования, могут быть применены к ряду других задач разработки программного обеспечения. Например, можно разделить практически любую задачу обработки файлов или сетевого ввода на три категории:

- чтение данных XML с помощью библиотеки XML;
- чтение данных JSON с помощью библиотеки JSON;
- чтение чего-либо еще путем написания кода для анализа его в собственном формате.

Технологии, описанные в данной книге, полезны в широком спектре задач разработки программного обеспечения, и именно здесь мы сталкиваемся с третьей из этих категорий. Часто структурированные данные должны быть прочитаны в пользовательском формате файла.

Для некоторых из вас опыт создания собственного языка программирования может стать самой большой программой, которую вы написали до сих пор. Если вы будете упорны и доведете дело до конца, то это научит вас многим практическим навыкам разработки программного обеспечения, помимо того что вы узнаете о компиляторах, интерпретаторах и т. п. Среди прочих навыков это будет включать в себя работу с большими динамическими

структурами данных, тестирование программного обеспечения, диагностику сложных проблем.

Достаточно вдохновляющей мотивации. Давайте поговорим о том, что вы должны сделать в первую очередь, а именно выяснить свои требования.

Определение требований к вашему языку

После того как вы убедились, что вам нужен новый язык программирования для того, чем вы занимаетесь, потратьте несколько минут на определение требований. Это открытый вопрос. Вы сами определяете, как будет выглядеть успех вашего проекта. Мудрые изобретатели языков не создают совершенно новый синтаксис с нуля. Вместо этого они определяют его в виде набора модификаций популярного существующего языка. Многие великие языки программирования (Lisp, Forth, SmallTalk и многие другие) были значительно ограничены в своем успехе из-за того, что их синтаксис неоправданно отличался от основных языков. Тем не менее ваши требования к языку включают в себя то, как он будет выглядеть, в том числе и синтаксис.

Что еще более важно, вы должны определить набор управляющих структур или семантику, в которых ваш язык программирования должен выйти за рамки существующих языков. Иногда это будет включать в себя специальную поддержку прикладной области, которая не очень хорошо обслуживается существующими языками и их библиотеками. Такие *языки, специфичные для конкретной области (доменно-ориентированные языки – domain-specific languages – DSLs)*, настолько распространены, что этой теме посвящены целые книги. Цель данной книги – сосредоточиться на деталях создания компилятора и среды выполнения для такого языка, независимо от област, в которой вы работаете.

В обычном процессе разработки программного обеспечения анализ требований начинается с мозгового штурма списков функциональных и нефункциональных требований. Функциональные требования к языку программирования включают в себя специфику того, как конечный пользователь – разработчик будет взаимодействовать с ним. Вы можете не предусмотреть все опции командной строки для языка, но вы, вероятно, знаете, требуется ли интерактивность или достаточно отдельного шага компиляции. При обсуждении интерпретаторов и компиляторов прошлого, а также компилятора в этой книге может показаться, что выбор за вас уже сделан. Однако Python – это пример языка, который предоставляет полностью интерактивный интерфейс, несмотря на то что исходный код, который вы набираете в нем, преобразуется в байт-код, а не интерпретируется.

Нефункциональные требования – это свойства вашего языка программирования, которые не связаны напрямую со взаимодействием с конечным пользователем – разработчиком. К ним относятся такие вещи, как то, в какой операционной системе (системах) он должен работать, насколько быстрым должно быть выполнение или какой объем памяти должны занимать программы, написанные на вашем языке.

Нефункциональное требование относительно того, насколько быстрым должно быть выполнение, обычно определяет ответ на вопрос, можно ли использо-

вать программную машину (машину байт-кода) или необходимо использовать собственный код. Собственный код не только быстрее, его также значительно сложнее генерировать, и это может сделать ваш язык значительно менее гибким с точки зрения возможностей среды выполнения. Вы можете сначала выбрать байт-код, а затем работать над генератором собственного кода.

Первым языком, на котором я научился программировать, был интерпретатор BASIC, в котором программы должны были выполняться в пределах 4 Кб оперативной памяти. В то время BASIC имел низкие требования к объему памяти. Но даже в наше время нередко можно встретиться с платформой, где Java не запускается по умолчанию! Например, на виртуальных машинах с настроенными ограничениями памяти для пользовательских процессов вам, возможно, придется изучить некоторые неудобные параметры командной строки, чтобы скомпилировать или запустить даже простые Java-программы.

Многие процессы анализа требований также определяют набор вариантов использования и просят разработчика написать описания для них. Изобретение языка программирования отличается от вашего обычного проекта по разработке программного обеспечения, но прежде чем закончите, вы, возможно, захотите этим заняться. Пример использования – это задача, которую кто-то выполняет с помощью программного приложения. Когда программным приложением является язык программирования, если вы не будете осторожны, примеры использования могут быть слишком общими, чтобы быть полезными, например «*написать мое приложение*» или «*запустить мою программу*». Хотя эти два варианта могут быть не очень полезными, вы, возможно, захотите подумать о том, должна ли ваша реализация языка программирования поддерживать разработку программ, отладку, отдельную компиляцию и компоновку, интеграцию с внешними языками и библиотеками и т. д. Большинство этих тем выходят за рамки данной книги, но мы рассмотрим некоторые из них.

Поскольку в этой книге будет представлена реализация языка под названием Jzero, ниже приведены некоторые требования к нему. Некоторые из этих требований могут показаться произвольными. Если вам не ясно, откуда взялось одно из них, то оно либо пришло из нашего исходного языка вдохновения (PL/0), либо из предыдущего опыта обучения построению компиляторов:

- *Jzero должен быть строгим подмножеством Java*. Все легальные программы Jzero должны быть легальными программами Java. Это требование позволяет проверять поведение наших тестовых программ при отладке реализации нашего языка;
- *Jzero должен предоставлять достаточно возможностей для проведения интересных вычислений*. Это включает операторы `if`, циклы `while` и множество функций с параметрами;
- *Jzero должен поддерживать несколько типов данных, включая булевы, целые числа, массивы и тип String*. Он должен поддерживать только часть их функциональности, как будет описано позже. Этих типов достаточно, чтобы обеспечить ввод и вывод интересных значений в вычислениях;
- *Jzero должен выдавать приемлемые сообщения об ошибках, показывая имя файла и номер строки, включая сообщения о попытках использования возможностей Java, не предусмотренных в Jzero*. Нам понадобятся разумные сообщения об ошибках для отладки реализации;

- *Izero должен работать достаточно быстро, чтобы быть практичным.* Это требование расплывчато, но оно подразумевает, что мы не будем делать чистый интерпретатор. Чистые интерпретаторы – это очень древняя вещь, напоминающая о 1960-х и 1970-х годах;
- *Izero должен быть как можно более простым, чтобы я мог его объяснить.* К сожалению, это не позволяет генерировать собственный код или даже байт-код JVM. Мы предоставим нашу собственную простую машину байт-кода.

Возможно, по ходу дела появятся дополнительные требования, но это только начало. Поскольку мы ограничены во времени и пространстве, возможно, этот список требований более важен тем, что в нем не сказано, а не тем, что в нем сказано. Для сравнения, вот некоторые из требований, которые привели к созданию языка программирования Unicon.

Тематическое исследование – требования, которые вдохновили на создание языка Unicon

Для запуска конкретного примера в этой книге будет использован язык программирования Unicon, расположенный по адресу *http://unicon.org*. Мы можем начать с разумных вопросов, таких как «зачем создавать Unicon?» и «каковы его требования?». Чтобы ответить на первый вопрос, мы будем работать, отталкиваясь от второго.

Unicon существует благодаря более раннему языку программирования *Icon*, разработанному в университете Аризоны (*http://www.cs.arizona.edu/icon/*). Icon имеет особенно хорошие возможности обработки строк и списков, используется для создания многих скриптов и утилит, а также в проектах по обработке языков программирования и естественного языка. Фантастические встроенные типы данных Icon, включая структурные типы, такие как списки и (хеш-) таблицы, оказали влияние на несколько языков, включая Python и Unicon. Фирменным исследовательским вкладом Icon в знакомый основной синтаксис является интегрированная целенаправленная оценка, в том числе перебор с возвратом и автоматическое возобновление генераторов. Требование № 1 к Unicon – сохранить эти лучшие части Icon.

Требование Unicon № 1 – сохранять то, что люди любят в Icon

Одна из тех вещей, которые люди любят в Icon, – это семантика выражений, включая генераторы и целенаправленную оценку. Icon также предоставляет богатый набор встроенных функций и типов данных, так что многие или большинство программ могут быть поняты непосредственно из исходного кода. Целью Unicon была стопроцентная совместимость с Icon. В конце концов мы достигли более 99%-ной совместимости.

Это своего рода скачок от *сохранения лучших фрагментов* к цели бессмертия – обеспечению того, чтобы старый исходный код работал вечно, но для Unicon мы включили это в требование № 1. Мы предъявили более жесткие тре-

бования к обратной совместимости, чем большинство современных языков. В то время как C обладает очень хорошей обратной совместимостью, C++, Java, Python и Perl являются примерами языков, которые ушли, в некоторых случаях очень далеко, от совместимости с программами, написанными на них в дни их славы. В случае с Unicon, возможно, 99 % программ Icon работают без изменений, как программы Unicon.

Icon был разработан для максимальной производительности программистов в небольших проектах. Типичная программа Icon – это менее 1000 строк кода, но Icon – это очень высокий уровень, и вы можете создать множество программ в несколько сотен строк кода! Тем не менее компьютеры продолжают становиться более мощными, и пользователи хотят писать гораздо более крупные программы, чем те, на которые рассчитан Icon. Требованием № 2 Unicon была поддержка программирования в крупномасштабных проектах.

Требование Unicon № 2 – поддержка крупномасштабных программ, работающих с большими данными

По этой причине Unicon добавляет классы и пакеты в Icon, подобно тому как C++ добавляет их в C. Unicon также улучшил формат объектного файла байткода и сделал многочисленные улучшения масштабируемости компилятора и среды выполнения. Он также улучшает существующую реализацию Icon с целью сделать его более масштабируемым во многих специфических аспектах, например используя гораздо более сложную хеш-функцию.

Icon предназначен для классической текстовой обработки локальных файлов UNIX с помощью конвейеров и фильтров. Со временем все больше и больше людей хотели писать на нем и требовали более сложных форм ввода/вывода, таких как работа с сетью или графикой. Требование Unicon № 3 заключается в поддержке повсеместных возможностей ввода/вывода на том же высоком уровне, что и встроенные типы.

Требование Unicon № 3 – высокоуровневый ввод/вывод для современных приложений

Поддержка ввода/вывода – это движущаяся цель. Сначала она включала сетевые средства, средства GDBM и базы данных ODBC для сопровождения двухмерной графики Icon. Затем она расширилась и стала включать различные популярные интернет-протоколы и трехмерную графику. Определение того, какие возможности ввода/вывода являются повсеместными, продолжает развиваться и варьироваться в зависимости от платформы. Примерами вещей, которые стали в настоящий момент достаточно распространенными, являются сенсорный ввод и жесты, а также программирование шейдеров.

Пожалуй, несмотря на миллиардное увеличение в скорости процессора и объеме памяти, самая большая разница между программированием в 1970 и в 2020 годах заключается в том, что мы ожидаем, что современные приложения будут использовать огромное количество сложных форм ввода-вывода – графику, сети, базы данных и т. д. Библиотеки могут обеспечить доступ

к такому вводу/выводу, но поддержка на уровне языка может сделать его более простым и интуитивно понятным.

Icon довольно портативен, его запускали на всех компьютерах – от Amigas до Crays и мейнфреймов IBM с набором символов EBCDIC. Хотя за прошедшие годы платформы изменились почти невероятно, Unicon по-прежнему сохраняет цель Icon – максимальную переносимость исходного кода: код, написанный на Unicon, должен продолжать работать без изменений на всех важных вычислительных платформах. Это приводит к требованию Unicon № 4.

Требование Unicon № 4 – обеспечить универсально реализуемые системные интерфейсы

В течение очень долгого времени переносимость означала работу на ПК, Mac и рабочих станциях UNIX. Но, опять же, набор важных вычислительных платформ является движущейся целью. В настоящее время в Unicon ведется работа над поддержкой Android и iOS, в случае если вы считаете их вычислительными платформами. Будут ли они считаться таковыми, зависит от того, достаточно ли они открыты и используются ли для общих вычислительных задач, но они определенно могут быть использованы в этом качестве.

Все те впечатляющие средства ввода-вывода, которые были реализованы для требования № 3, должны быть разработаны таким образом, чтобы они могли быть многоплатформенными и переносимыми на все основные платформы.

Имея некоторые из основных требований к Unicon, можно ответить на вопрос, зачем вообще создавать Unicon. Один из ответов заключается в том, что после изучения многих языков я пришел к выводу, что генераторы Icon и целенаправленная оценка (требование № 1) были теми функциями, которые я хотел бы иметь при дальнейшем написании программ. Но после того как я позволил добавить двумерную графику в свой язык, изобретатели Icon больше не хотели рассматривать дальнейшие дополнения для удовлетворения требований № 2 и № 3. Другой ответ заключается в том, что существовал общественный спрос на новые возможности, включая партнеров-добровольцев и некоторую финансовую поддержку. Таким образом, родился Unicon.

Заключение

В этой главе вы узнали о разнице между изобретением языка программирования и изобретением библиотеки API для поддержки любых видов вычислений, которыми вы хотите заниматься. Было рассмотрено несколько различных форм реализации языка программирования. Первая глава позволила вам подумать о функциональных и нефункциональных требованиях к вашему собственному языку. Эти требования могут отличаться от приведенных в примерах требований, рассмотренных для подмножества Java Jzero и языка программирования Unicon.

Требования важны, потому что они позволяют вам установить цели и определить, как будет выглядеть успех. В случае реализации языка программирования требования включают в себя то, как все будет выглядеть и ощущаться программистами, использующими ваш язык, а также то, на каких аппарат-

ных и программных платформах он должен работать. Внешний вид и ощущение языка программирования включают в себя ответы на внешние вопросы о том, как будет выглядеть реализация языка и как вызываются программы, написанные на нем, а также на внутренние вопросы, такие как многословие – сколько программист должен написать, чтобы выполнить заданную вычислительную задачу.

Возможно, вам захочется сразу перейти к кодированию. Хотя классический принцип менталитета начинающих программистов «*собери и исправь*» может работать со скриптами и короткими программами, для такой большой части программного обеспечения, как язык программирования, нужно немного больше планирования. После рассмотрения требований в этой главе *глава 2* «*Проектирование языка программирования*» подготовит вас к составлению подробного плана реализации, который займет все наше внимание до конца нашей книги.

Вопросы

1. Каковы плюсы и минусы написания транспайлера языка (генерирует код на языке C) вместо традиционного компилятора (генерирует ассемблер или собственный машинный код)?
2. Каковы основные компоненты или шаги традиционного компилятора?
3. Каковы, по вашему опыту, некоторые болевые точки, в которых программирование оказывается сложнее, чем должно быть? Какая новая функция(и) языка программирования решает эти болевые точки?
4. Напишите набор функциональных требований для нового языка программирования.

Глава 2

Дизайн языка программирования

Прежде чем пытаться создать язык программирования, необходимо его определить. Это включает в себя разработку особенностей языка, которые видны на его поверхности, в том числе основные правила формирования слов и пунктуации. Сюда также входят правила более высокого уровня, называемые *синтаксисом*, которые определяют количество и порядок слов и знаков препинания в более крупных фрагментах программы, таких как выражения, операторы, функции и программы. Дизайн языка также включает в себя основной смысл, известный как *семантика*.

Разработка языка программирования часто начинается с написания примеров кода, иллюстрирующих каждую из важных особенностей вашего языка, а также показывающих возможные вариации для каждой конструкции. Написанные примеры кода позволяют найти и исправить многие возможные несоответствия в ваших первоначальных идеях. Затем на основе этих примеров вы можете зафиксировать общие правила, которым следует каждая конструкция языка. Запишите предложения, которые описывают ваши правила, как вы поняли их из примеров. Обратите внимание, что существует два вида правил. *Правила лексики* определяют, какие символы должны рассматриваться вместе, например слова или операторы из нескольких символов, такие как ++. С другой стороны, *правила синтаксиса* – это правила для сочетания нескольких слов или знаков препинания для формирования более широкого смысла. В естественном языке часто это фразы, предложения или абзацы, в то время как в языке программирования это могут быть выражения, операторы, функции или программы.

Как только вы придумаете примеры всего, что вы хотите, чтобы делал ваш язык, а также запишете правила лексики и синтаксиса, напишите документ по дизайну языка (или спецификацию языка), на который вы сможете ссылаться при кодировании языка. Вы можете изменить его позже, но это поможет иметь план, по которому нужно работать.

В этой главе мы рассмотрим следующие основные темы:

- определение типов слов и знаков препинания, которые следует предусмотреть в вашем языке;
- определение потока управления;
- решение о том, какие типы данных следует поддерживать;
- общая структура программы;

- завершение определения языка Jzero;
- тематическое исследование – проектирование графических средств в Unicon.

Давайте начнем с определения основных элементов, которые допустимы в исходном коде вашего языка.

Определение видов слов и пунктуации в вашем языке

В языках программирования есть несколько различных категорий слов и знаков препинания. В естественном языке слова делятся на *части речи* – существительные, глаголы, прилагательные и т. д. Категории, соответствующие частям речи, которые вам придется придумать для языка программирования, можно построить с помощью следующих действий:

- определение набора *зарезервированных слов* или ключевых слов;
- определение символов в *идентификаторах* переменных, функций и констант;
- создание формата *литеральных* постоянных значений для встроенных типов данных;
- определение одно- и многобуквенных *операторов* и знаков препинания.

Вы должны записать точные описания каждой из этих категорий как части вашего документа по дизайну языка. В некоторых случаях вы можете просто составить списки определенных слов или пунктуации, но в других случаях вам понадобятся шаблоны или иные способы передачи информации о том, что допустимо и недопустимо в данной категории.

Для зарезервированных слов пока достаточно списка. Для имен объектов точное описание должно включать такие детали, как то, какие небуквенные символы разрешены в таких именах. Например, в Java имена должны начинаться с буквы, а затем могут включать буквы и цифры, подчеркивания разрешены и рассматриваются как буквы. В других языках дефисы разрешены в именах, поэтому три символа `a`, – и `b` составляют правильное имя, а не вычитание `a` из `b`. Когда точное описание невозможно, достаточно привести полный набор примеров.

Постоянные значения, также называемые *литералами*, являются неожиданным и основным источником сложности в лексических анализаторах. Попытка точного описания вещественных чисел в Java выглядит примерно так: в Java есть два разных вида вещественных чисел – с плавающей точкой и двойной точности, но они выглядят одинаково, пока вы не дойдете до конца, где есть опциональные `f` (или `F`) или `d` (или `D`) для отличия чисел с плавающей точкой от чисел с двойной точностью. До этого вещественные числа должны иметь либо десятичную точку (`.`), либо экспоненту (`e` или `E`), либо и то, и другое. Если есть десятичная точка, то должна быть хотя бы одна цифра с одной или другой стороны от нее. Если есть экспоненциальная часть, то она должна содержать букву `e` (или `E`), за которой следует необязательный знак минус и одна или несколько цифр. Чтобы усугубить ситуацию, в Java есть странный шестнадцатеричный формат вещественных констант, о котором слышали немногие программисты, состоящий из `0x` или `0X`, за которыми следуют цифры в шестнадцатеричном

формате, с необязательной десятичной и обязательной экспоненциальной частью, состоящей из `p` (или `P`), за которой следуют цифры в десятичном формате.

Описать операторы и знаки препинания обычно почти так же легко, как перечислить зарезервированные слова. Одно из основных отличий заключается в том, что операторы обычно имеют *правила приоритета*, которые вам нужно будет определить. Например, при обработке чисел оператор умножения почти всегда имеет более высокий приоритет, чем оператор сложения, поэтому `x + y * z` умножит `y` на `z` прежде, чем прибавит `x` к произведению `y` и `z`. В большинстве языков существует по крайней мере 3–5 уровней приоритета, а многие популярные основные языки имеют от 13 до 20 уровней приоритета, которые необходимо тщательно учитывать. На рис. 2.1 показана таблица приоритетов операторов для Java. Она понадобится нам для работы с Jzero.

Этот рисунок показывает, что в Java есть множество операторов, организованных в 10 уровней приоритета, хотя я, возможно, немного упрощаю. В вашем языке вы можете обойтись их меньшим количеством, но вам придется решить вопрос о приоритете операторов, если вы хотите создать настоящий язык.

Аналогичная проблема связана с *ассоциативностью* операторов. Во многих языках большинство операторов ассоциируются слева направо, но некоторые странные операторы ассоциируются справа налево. Например, выражение `x + y + z` эквивалентно `(x + y) + z`, но выражение `x = y = 0` эквивалентно `x = (y = 0)`.

Принцип наименьшего удивления применим к приоритету и ассоциативности операторов, а также к тому, какие операторы вы вводите в язык в первую очередь. Если вы определите арифметические операторы и дадите им странный приоритет или ассоциативность, люди отвергнут ваш язык сразу же. Если вы вводите в свой язык новые, возможно, специфические для конкретной области типы данных, у вас есть гораздо больше свободы в определении приоритета и ассоциативности для любых новых операторов, которые вы вводите в язык.

После того как вы выяснили, какими должны быть отдельные слова и знаки препинания в вашем языке, можно переходить к более крупным конструкциям. Это переход от лексического анализа к синтаксису, а синтаксис важен, потому что это уровень, на котором фрагменты кода становятся достаточно большими, чтобы определить некоторые вычисления, которые должны

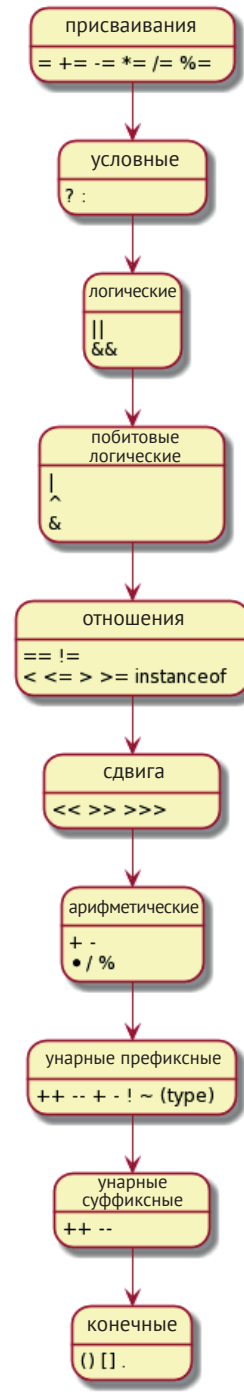


Рис. 2.1. Приоритет операторов Java

быть выполнены. Мы рассмотрим это более подробно в последующих главах, но на этапе проектирования вы должны, по крайней мере, подумать о том, как программисты будут определять поток управления, объявлять данные и строить программы. Во-первых, вы должны спланировать поток управления.

Определение потока управления

Поток управления – это то, как выполнение программы переходит от одного места к другому в пределах исходного кода. Большинство конструкций потока управления должны быть знакомы программистам, которые изучали основные языки программирования. Инновации в дизайне вашего языка могут быть сосредоточены на особенностях, которые являются новыми или специфичными для конкретной области и в первую очередь побудили вас к созданию нового языка. Сделайте эти новинки как можно более простыми и как можно более читаемыми. Представьте, как эти новые возможности должны вписываться в остальную часть языка программирования.

В каждом языке должны быть условные выражения и циклы, и почти все они используют `if` и `while` для своего запуска. Вы можете придумать свой собственный специальный синтаксис для выражения `if`, но если у вас нет на то веских причин, вы только навредите себе. Вот некоторые конструкции потока управления из Java, которые, безусловно, должны быть в Jzero:

```
if (e) s;
if (e) s1 else s2;
while (e) s;
for (…) s;
```

Вот некоторые другие менее распространенные конструкции потока управления Java, которых нет в Jzero. Если они появятся в программе, что должен сделать с ними компилятор Jzero?

```
switch (e) { … }
do s while (e);
```

По умолчанию наш компилятор будет выводить загадочное сообщение, которое не очень хорошо объясняет ситуацию. В следующих двух главах мы заставим наш компилятор для Jzero выводить красивое сообщение об ошибке обращения к функциям Java, которые он не поддерживает.

Помимо условных выражений и циклов, языки обычно имеют синтаксис для вызова подпрограмм и возврата после этого. Все эти вездесущие формы потока управления являются абстракциями для обозначения возможности базовой машины изменять место выполнения инструкций – GOTO. Если вы изобретете лучшую нотацию для изменения места выполнения инструкций, это будет большая удача.

Наибольшие разногласия при разработке многих или большинства конструкций потока управления возникают по поводу того, являются они операторами или их следует сделать выражениями, которые производят результат, который может быть использован в окружающем выражении. Я использовал

языки, в которых результат выражений `if` полезен. В C/C++/Java даже есть оператор для этого – условный оператор `i?t:e`. Я не нашел языка, который сделал бы что-то очень значимое при превращении цикла `while` в выражение. Лучшее, что они сделали, – это то, что выражение `while` выдает результат, сообщающий, завершился цикл из-за условия проверки или из-за внутреннего прерывания.

Если вы изобретаете новый язык с нуля, один из главных вопросов для вас – сможете ли вы придумать новую структуру (структуры) управления для поддержки вашей предполагаемой области применения. Например, предположим, вы хотите, чтобы ваш язык обеспечивал специальную поддержку для инвестирования на фондовом рынке. Если вам удастся придумать лучшую управляющую структуру для задания условий, ограничений или итеративных операций в этой области, вы сможете обеспечить конкурентное преимущество тем, кто кодирует на вашем языке для этой области. Программа должна будет работать на базовом наборе инструкций фон Неймана, поэтому вам придется выяснить, как сопоставить любую новую структуру управления с такими инструкциями, как булевы логические тесты и инструкции GOTO.

Какие бы конструкции потока управления вы ни решили поддерживать, вам также потребуется разработать набор типов данных и деклараций, которые отражают информацию, обрабатываемую программами на вашем языке.

Решение о том, какие типы данных поддерживать

Существует как минимум три категории типов данных, которые необходимо учитывать при разработке языка. Первая – это *атомарные*, скалярные примитивные типы, часто называемые типами данных первого класса. Вторая – *составные*, или контейнерные, типы, которые способны хранить и организовывать совокупности значений. Третья (которая может быть вариантом первой или второй категории) – это типы, *специфичные для конкретной области*. Вы должны сформулировать план для каждой из этих категорий.

Атомарные типы

Атомарные типы обычно являются встроенными и неизменяемыми. Вы не изменяете существующие значения, просто используете операторы для создания новых значений. Практически все языки имеют встроенные типы для чисел и несколько дополнительных типов. Булев тип, тип `null` и, возможно, строковый тип являются обычными атомами, но есть и другие.

Вы сами решаете, насколько сложными должны быть атомы – сколько различных машинных представлений целых и вещественных чисел вам нужно? Некоторые языки могут предоставлять один тип для всех чисел, а другие могут предоставлять 5 или 10 (или больше) типов для целых чисел и еще несколько для вещественных чисел. Чем больше типов вы добавляете, тем больше гибкости и контроля вы предоставляете программистам, использующим ваш язык, но тем более сложной будет ваша задача для реализации.

Точно так же невозможно разработать единый строковый тип данных, который идеально подходит для всех приложений, которые часто используют

строки. А сколько типов строк вы хотите поддерживать? Одна из крайностей – это вообще не иметь строкового типа, а только короткий целочисленный тип для хранения символов. Такие языки будут считать строки частью составных типов. Возможно, строки поддерживаются только библиотекой, а не языком. Строки могут быть массивами или объектами, но даже в таких языках обычно есть специальные лексические правила, которые позволяют строковым константным значениям быть заданными как последовательности символов в двойных кавычках. Другая крайность заключается в том, что, учитывая важность строк во многих областях применения, ваш язык может захотеть поддерживать несколько типов строк для различных наборов символов (ASCII, UTF8 и т. д.) со вспомогательными типами (наборами символов) и специальными типами и управляющими структурами, поддерживающими анализ и построение строк. Многие популярные языки рассматривают строки как специальный атомарный тип.

Если вы особенно умны, то можете решить поддерживать только несколько встроенных типов для чисел и строк, но сделать эти типы настолько гибкими, насколько это возможно. Популярные существующие языки программирования сильно различаются по количеству типов, используемых для классических встроенных типов и многих других возможных типов данных, которые вы можете включить. Как только вы выходите за пределы целых чисел, вещественных чисел и строк, единственные универсальные типы – это контейнерные типы, которые позволяют собирать структуры данных.

Некоторые из вещей, о которых вы должны подумать в отношении атомарных типов, включают следующее:

- сколько значений они имеют?
- как все эти значения кодируются в исходном коде в виде литеральных констант?
- какие типы операторов или встроенных функций используют операнды или параметры?

Ответ на первый вопрос скажет вам, сколько байт займет тип в памяти. Второй и третий вопросы связаны с определением правил для слов и пунктуации в языке. Третий вопрос также может дать представление о том, сколько усилий, с точки зрения генератора кода или среды выполнения, потребуется для реализации поддержки типа в вашем языке. Атомарные типы могут быть более или менее трудоемкими в реализации, но они редко бывают такими сложными, как составные типы. Мы обсудим их далее.

Составные типы

Составные типы – это типы, которые помогают согласованно выделять и получать скоординированный доступ к нескольким значениям. Языки очень сильно различаются по степени поддержки синтаксиса для составных типов. Некоторые поддерживают только массивы и структуры и требуют от программистов создавать все собственные структуры данных на их основе. Многие предоставляют все составные типы более высокого уровня через библиотеки. Однако некоторые языки более высокого уровня предоставляют множество сложных структур данных в виде встроенных модулей с поддержкой синтаксиса.

Наиболее распространенным составным типом является тип *массива*, в котором доступ к нескольким значениям осуществляется с использованием численно непрерывного диапазона целочисленных индексов. Вероятно, в вашем языке есть что-то, похожее на массив. Основными соображениями при проектировании должны быть следующие: как задаются индексы и как обрабатываются изменения размера составного значения? Большинство популярных языков используют индексы, начинающиеся с нуля. Индексы массивов, начинающиеся с нуля, упрощают вычисления индексов и проще для изобретателя языка, но они менее интуитивно понятны для начинающих программистов. Некоторые языки используют индексы, основанные на единице, или позволяют программисту указать диапазон индексов, начинающийся с произвольного целого числа, отличного от 0.

Что касается изменения размера, то некоторые языки вообще не допускают изменения размера в своих типах массивов, или они заставляют программиста прыгать сквозь обручи, чтобы создать новые массивы различных размеров на основе существующих массивов. Другие языки обеспечивают добавление значений в массив при помощи дешевой и простой операции. Ни одно решение не является идеальным для всех приложений, поэтому вы просто выбираете одно из них и живете с последствиями, поддерживаете несколько типов данных, похожих на массивы, для разных целей, или разрабатываете очень умный тип, который хорошо подходит для широкого спектра распространенных применений.

Помимо массивов, вы должны подумать о том, какие еще составные типы вам нужны. Почти все языки поддерживают тип записи (record), структуры (struct) или класса (class) для группировки значений нескольких различных типов вместе и доступа к ним по именам, называемым полями. Чем более сложными они становятся, тем сложнее будет реализация вашего языка. Если вам нужна правильная объектная ориентация в вашем языке, будьте готовы заплатить за это временем, потраченным на написание кода компилятора и исполняемого кода. Как проектировщик предупреждаю: не усложняйте, но как программист я бы не хотел использовать язык программирования, который не предоставляет мне эту возможность в той или иной форме.

Возможно, вы сможете вспомнить еще несколько составных типов, которые необходимы для вашего языка. Это замечательно, особенно если они будут часто использоваться в программах, которые вам небезразличны. Я расскажу еще об одном составном типе, который имеет большое практическое значение, – (хеш-) *табличном* типе данных, который также часто называют *словарным* типом. Табличный тип – это нечто среднее между массивом и типом записи. Вы индексируете значения с помощью имен, и эти имена не являются фиксированными, новые имена могут быть вычислены в процессе работы программы. Любой современный язык, который не использует этот тип, просто оставляет без внимания многих своих потенциальных пользователей. По этой причине ваш язык может захотеть включить табличный тип. Составные типы – это «клей» общего назначения, который используется для сборки сложных структур данных. Но вы должны также рассмотреть вопрос о том, нужны ли в вашем языке типы специального назначения, атомарные или составные, для поддержки приложений, которые трудно написать на языках общего назначения.

Типы, специфичные для конкретной области

Кроме того, какие бы атомарные и составные типы общего назначения вы ни решили включить, вы должны подумать о том, ориентирован ли ваш язык программирования на специфическую область. Если да, то какие типы данных может включать ваш язык для поддержки этой области? Существует плавный континуум между языками, специфичными для конкретной области, которые предоставляют *специфичные для данной области* типы, и управляющими структурами и языками общего назначения, такими как C++ и Java, которые предоставляют библиотеки для всего. Библиотеки классов являются мощным инструментом, но для некоторых приложений и доменов библиотечный подход может быть более сложным и подверженным ошибкам, чем язык, специально разработанный для поддержки данной области. Например, в Java и C++ есть строковые классы, но они не поддерживают сложные приложения для обработки текста лучше, чем языки, которые имеют специализированные типы и управляющие структуры для обработки строк. Помимо типов данных, при проектировании языка вам понадобится представление о том, как собираются и организуются программы.

Общая структура программы

Рассматривая общую структуру программы, мы должны обратить внимание на то, как программы организованы и собраны вместе, а также на громоотводный вопрос о том, какова вложенность вашего языка. Кажется, что это не важно, но как и где будет выполняться исходный код в программах? В языках, основанных на C, выполнение начинается с функции `main()`, в то время как в скриптовых языках исходный код выполняется по мере его чтения, поэтому нет необходимости в функции `main()`, чтобы начать выполнение.

Структура программы также поднимает основной вопрос о том, должна ли вся программа транслироваться и выполняться целиком или же различные пакеты, классы или функции могут быть отдельно скомпилированы, а затем связаны и/или загружены вместе для запуска программы. Изобретатель языка может избежать многих сложностей реализации, либо встроив что-то в язык (если это встроено, то нет необходимости разбираться с линковкой), требуя, чтобы весь исходный код программы был представлен во время выполнения, либо генерируя код для некоторого хорошо известного стандартного формата выполнения, где чужой компоновщик и загрузчик сделает всю тяжелую работу.

Возможно, самый большой вопрос проектирования, связанный с общей структурой программы, заключается в том, какие конструкции могут быть вложенными и какие ограничения на вложенность существуют, если таковые имеются. Наверное, это лучше всего проиллюстрировать на примере. Давным-давно, около 1970 года были изобретены два непонятных языка, которые боролись за доминирование, – *C* и *Pascal*.

Язык C был почти плоским – программа представляла собой набор функций, связанных вместе, и только относительно небольшие (мелкозернистые) вещи могли быть вложенными – выражения, операторы и, неохотно, определения структур.

В отличие от него, язык Pascal был невероятно более вложенным и рекурсивным. Почти всё можно было вложить. Примечательно, что функции можно было встраивать внутрь функций, произвольно глубоко. Хотя C и Pascal были примерно равны по мощности и Pascal имел небольшую стартовую фору и был самым популярным в университетских курсах, C в конечном итоге победил. Почему? Оказалось, что вложенность увеличивает сложность, не добавляя при этом особой ценности. Или, может быть, просто из-за могущества американских корпораций.

Поскольку C победил, многие современные распространенные языки (я особенно думаю о C++ и Java) сначала были почти плоскими. Но со временем они добавляли все больше и больше вложенности. Почему так происходит? То ли потому, что среди нас скрываются поклонники Pascal, то ли потому, что естественным для языков программирования является добавление функций с течением времени, пока они не станут чрезмерно инженерными. Никлаус Вирт (Niklaus Wirth) предвидел это и выступал за возвращение к малости и простоте в программном обеспечении, но к его мольбам в основном остались глухи, и в его языках поддерживается высокая степень вложенности.

Каков практический результат для вас как для начинающего разработчика языка? Не переусердствуйте в разработке своего языка. Делайте его как можно более простым. Не встраивайте элементы, если они не нуждаются во вложении. И будьте готовы платить (как разработчик языка) каждый раз, когда вы игнорируете этот совет!

Теперь пришло время привести несколько примеров дизайна языка из Jzero и Unicon. В случае Jzero, так как он является подмножеством Java, дизайн – это либо большой пустой бутерброд (мы используем дизайн Java), либо результат *вычитания* – что мы возьмем из Java, чтобы сделать Jzero, и как это будет выглядеть и ощущаться. Несмотря на ранние попытки сделать его маленьким, Java – большой язык. Если, как часть нашего проекта, мы составим список всего, что есть в Java и чего нет в Jzero, это будет длинный список.

Из-за ограничений объема страницы и времени программирования Jzero должен быть крошечным подмножеством Java. Однако, в идеале, любая допустимая программа на Java, которая вводится в Jzero, не будет позорно проваливаться – она либо компилируется и выполняется правильно, либо выводит полезное сообщение, объясняющее, какая функция(и) Java используется, которую Jzero не поддерживает. Чтобы вы могли легко понять остальную часть этой книги, а также свести ваши ожидания к приемлемому размеру, в следующем разделе будут рассмотрены дополнительные подробности того, что есть в Jzero и чего нет.

Завершение определения языка Jzero

В предыдущей главе мы перечислили требования к языку, который будет реализован в этой книге, а в предыдущем разделе мы подробно остановились на некоторых аспектах его разработки. Для справочных целей в этом разделе будут описаны дополнительные детали, касающиеся языка Jzero. Если вы обнаружите какие-либо несоответствия между этим разделом и нашим компилятором Jzero, то это ошибки. Разработчики языков программирования используют более точные формальные инструменты для определения различных

аспектов языка. Нотации для описания лексических и синтаксических правил будут представлены в следующих двух главах. В этом разделе мы опишем язык в простых терминах.

Программа Jzero состоит из одного класса в одном файле. Этот класс может состоять из множества методов и переменных, но все они статические (`static`). Программа Jzero начинается с выполнения статического метода `main()`, который является обязательным. Виды операторов, которые разрешены в Jzero, – операторы присваивания, операторы `if`, операторы `while` и вызов пустых (`void`) методов. Виды выражений, которые разрешены в программе Jzero, включают арифметические, реляционные и булевы логические операторы, а также вызов непустых (`non-void`) методов.

Язык Jzero поддерживает атомарные типы `boolean`, `char`, `int` и `long`. Типы `int` и `long` эквивалентны 64-битным целочисленным типам данных.

Jzero также поддерживает массивы, типы классов `String`, `InputStream` и `PrintStream` в качестве встроенных типов, а также подмножества их обычной функциональности. Тип `String` поддерживает оператор конкатенации и методы `charAt()`, `equals()`, `length()` и `substring(b,e)`. `String` также поддерживает статический метод класса `valueOf()`. Тип `InputStream` поддерживает методы `read()` и `close()`, а тип `PrintStream` поддерживает методы `print()`, `println()` и `close()`.

Таким образом, мы определили минимальные возможности, необходимые для написания базовых вычислений на игрушечном языке, напоминающем Java. Он не предназначен для того, чтобы быть настоящим языком. Тем не менее вам рекомендуется расширить язык Jzero дополнительными возможностями, для которых не хватило места в этой книге, например типами с плавающей точкой и определяемыми пользователем классами с нестатическими переменными класса. Теперь давайте посмотрим, что мы можем заметить в дизайне языка, рассмотрев один аспект языка Unicon.

Тематическое исследование – проектирование графических объектов в Unicon

Графика Unicon конкретна и нетривиальна по размеру. Проектирование графических средств Unicon является реальным примером, иллюстрирующим некоторые компромиссы при разработке языков программирования. Большинство языков программирования не имеют встроенной графики (или каких-либо встроенных средств ввода/вывода), вместо этого все средства ввода/вывода передаются библиотекам. Язык C, безусловно, осуществляет ввод/вывод через библиотеки, а графические средства Unicon построены поверх API языка C. Когда дело доходит до библиотек, многие языки эмулируют язык нижнего уровня, на котором они реализованы (например, C или Java), и пытаются обеспечить перевод API языка реализации в формате 1:1. Когда языки более высокого уровня реализуются поверх языков более низкого уровня, этот подход обеспечивает полный доступ к базовому API за счет снижения уровня языка при использовании этих средств.

Для Unicon это было невозможно по нескольким причинам. Графика Unicon была добавлена через два отдельных больших дополнения к языку – сна-

чала 2D, а затем 3D. Мы рассмотрим вопросы их проектирования по отдельности. В следующем разделе описываются средства графики 2D Unicon.

Поддержка языка для графики 2D

Поддержка графики 2D в Unicon была последней важной особенностью, которая была введена в язык Icon, перед тем как он был заморожен. При разработке особое внимание уделялось минимизации поверхностных изменений синтаксиса языка, поскольку большие изменения были бы отвергнуты. Единственные поверхностные изменения заключались в добавлении нескольких ключевых слов, обозначающих специальные значения в графической системе. Ключевые слова в Unicon выглядят как имена переменных с амперсандом перед ними.

Добавление 19 ключевых слов помогает создать ощущение, что графические средства принадлежат языку, известному в основном своей обработкой строк. Вы можете удивиться, узнав, что вывод графики – это самая простая часть. Все ключевые слова, кроме одного, посвящены упрощению обработки *входных* событий мыши и клавиатуры. 10 из них являются целочисленными константами, обозначающими события мыши и изменения размера, и используются для удобства, остальные восемь содержат ключевую информацию о последнем полученном событии, и они обновляются автоматически для каждого события. Благодаря целочисленным константам для обработки ввода с помощью мыши не требуется файл заголовка или импорт. Последним и основным ключевым словом, добавляемым в программу, является `&window`. Это ключевое слово содержит окно по умолчанию, все функции графических средств используют это окно, если в качестве первого необязательного аргумента не указано другое значение окна.

Интересно сравнить графику Unicon с графикой, предоставляемой базовой реализацией. В то время базовыми API на языке C были высокоуровневые наборы инструментов X Window System (такие как виджеты HP и Athena) и библиотека нижнего уровня Xlib. Высокоуровневые наборы инструментов были отвергнуты во время создания прототипа из-за их непредсказуемого поведения и недостаточной переносимости. Библиотека Xlib удовлетворяла требованиям к поведению и переносимости, но это был огромный API, требующий множества новых типов (например, отдельного типа `struct` для каждого из десятков различных видов событий) и имеющий около тысячи функций.

Изучение Xlib и последующее программирование графических приложений на C с использованием Xlib является чрезвычайно сложной задачей, а целью Unicon было предоставить высокоуровневые возможности, которые было бы легко использовать. Наиболее прямое влияние в направлении простоты использования оказал BASIC. Использование графики TRS-80 Extended Color BASIC в 1970-х годах было намного проще, чем любой из API X Window C. Для такого языка высокого уровня, как Unicon, графические средства должны быть такими же простыми и более функциональными, чем те, которые предоставляет Extended Color BASIC. Требование сохранить то, что люди любят в Icon, распространяется и на соответствие дизайна графических средств существующим средствам ввода и вывода Icon. Средства ввода и вывода Icon включают тип файла, встроенные функции и операторы, которые выполняют ввод и вывод.

Для Unicon был введен единственный новый тип («окно» – window) как подтип (и расширение) типа данных файла Icon. Окно – это представление, возможно, дюжины различных сущностей Xlib в базовом коде C, но для программиста Unicon это была единственная простая вещь, которую можно было создать и отобразить. Все существующие (текстовые) операции ввода/вывода для файлов были сделаны для работы с окнами, а затем были добавлены возможности вывода графики. Рисунок 2.2 иллюстрирует некоторые из базовых сущностей библиотеки C, все они свернуты в окно Unicon. Листья этой структуры несколько различаются в зависимости от платформы, различия между платформами минимизированы или устранены на уровне Unicon.

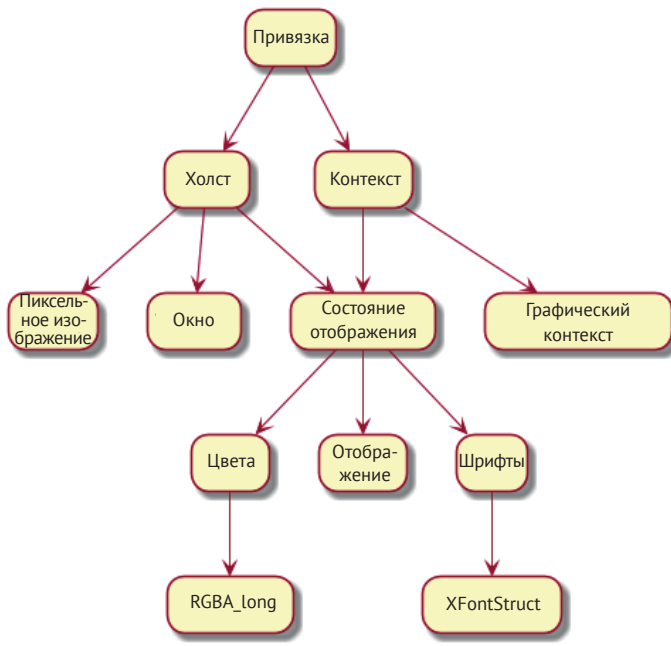


Рис. 2.2. Внутренняя структура окна Unicon

Возможности вывода графики в Unicon представлены в виде набора из 40 или около того встроенных функций для рисования различных графических примитивов. Специфика вывода зависит от многих элементов состояния, таких как копия содержимого окна в памяти, и представления ресурсов, например шрифтов и шаблонов заливки. Вместо того чтобы вводить новые типы для всех этих ресурсов, был создан API для управления ими с помощью строковых значений. Окно было в конечном итоге определено как пара двух базовых сущностей – холста и контекста.

Структуры управления и организация программы являются основными факторами при разработке особенностей языка. При написании графических программ на языке C программиста сразу же учат (и вынуждают) отдать поток управления библиотеке и организовать свои программы как набор функций *обратного вызова*. Это функции, которые вызываются при наступлении различных

событий. Переписать интерпретатор байт-кода Unicon для такой организации не представлялось возможным. Интерпретатор байт-кода должен иметь поток управления циклом выборки–декодирования–исполнения. Многопоточное решение можно было бы заставить работать, но в то время потоки представляли собой неприемлемые проблемы переносимости и производительности. Вместо этого однопоточное, неблокирующее решение было достигнуто за счет того, что время от времени интерпретатор байт-кода проверял наличие графических событий, обрабатывал общие задачи, такие как перерисовка содержимого окна из резервного хранилища и постановка в очередь для последующей обработки на уровне языка Unicon, когда этого требовал поток управления приложением.

Можно было бы ввести библиотеку C для 2D-графики в язык Unicon «как есть» (без изменений), но это не соответствовало бы уровню языка и простоте использования. Вместо этого был введен высокоуровневый тип данных, состоящий из множества базовых частей библиотеки C. Операции обслуживания и обновления, которые поддерживали этот высокоуровневый тип данных, были встроены из нескольких мест в среду выполнения языка, что позволило создать простой в использовании тип окна, который был бы невозможен при строгом использовании библиотеки.

Через несколько лет стала повсеместной аппаратная поддержка 3D-графики. В следующем разделе описаны вопросы проектирования, связанные с добавлением в язык 3D-графики.

Добавление поддержки трехмерной графики

2D-графика была добавлена в Unicon как расширение типа данных файла и поддерживала обычные файловые операции, такие как открытие, закрытие, чтение и запись. Тот факт, что существовало связанное окно, в котором можно было работать с отдельными пикселями и другими графическими примитивами, был бонусом. Аналогично 3D-графика была добавлена как расширение 2D-графики. 3D-окна поддерживают примитивы просмотра камеры в трехмерном пространстве, но они поддерживают те же атрибуты (такие как цвет и шрифты) в той же нотации, что и 2D-объекты, а также предоставляют те же возможности ввода и дополнительные примитивы вывода графики.

Внутренняя реализация OpenGL была необходима для переносимой 3D-графики. Реализация OpenGL кардинально изменила ситуацию, и в конечном итоге это распространилось обратно – на уровень исходного языка. Если холст 2D-окна представляет собой двумерный массив пикселей, которые можно читать и записывать, то холст 3D-окна включает *список отображения* (display list), который перерисовывается каждый кадр. OpenGL предоставил список отображения, который по сути был ускорением производительности, объединяющим примитивы для повторного использования. В Unicon этим списком отображения можно управлять напрямую, чтобы вызвать различные эффекты анимации, такие как изменение размера или положения отдельных примитивов. Список отображения имеет центральное значение для управления *уровнем детализации* (level of detail – LOD) и выбора 3D-объектов. Была добавлена структура управления для маркировки и наименования разделов списка отображения, которые затем могут быть включены/отключены или выбраны для ввода пользователем. Базовая библиотека OpenGL не поддерживает напря-

мую выбор 3D-объектов, что является основополагающим для предоставления пользователям возможности взаимодействовать с объектами в 3D-сцене.

Это обсуждение дизайна графических средств Unicon является неполным из-за ограниченности объема. Изначально в 2D-объектах дизайн был намеренно минималистичным. Хотя результат был успешным, можно утверждать, что графические средства Unicon должны делать больше. Например, можно придумать новые управляющие структуры, которые еще более упростили бы операции вывода графики. В любом случае, это обсуждение дизайна должно дать вам некоторое представление о проблемах, которые могут возникнуть при добавлении в существующий язык поддержки новой области.

Заключение

В этой главе были представлены некоторые вопросы, связанные с разработкой языка. Навыки, которые вы здесь приобрели, включают в себя вопросы лексического дизайна, в том числе создание литеральных константных обозначений для типов данных, проектирования синтаксиса, включая операторы и управляющие структуры, и организации программы, в том числе решение о том, как и где начинать ее выполнение.

Причина, по которой вам следует потратить некоторое время на проектирование, заключается в том, что вам понадобится хорошее представление о том, что будет делать ваш язык программирования, чтобы реализовать его. Если вы отложите принятие решений по дизайну до тех пор, пока не придется их реализовывать, ошибки обойдутся вам дороже. Проектирование языка включает в себя определение типов данных, которые он поддерживает, способов объявления переменных и введения значений, управляющих структур и синтаксиса, необходимого для поддержки кода на различных уровнях детализации, от отдельных команд до целых программ. Как только вы закончите или решите, что закончили с этим, приходит время писать код, начиная с функции для чтения исходного кода, которой посвящена следующая глава.

Вопросы

1. В некоторых языках программирования вообще нет зарезервированных слов, но в большинстве популярных основных языков их несколько десятков. Каковы преимущества и недостатки добавления в язык большего количества зарезервированных слов?
2. Лексические правила для литеральных констант часто являются самыми большими и сложными правилами в лексической спецификации языка программирования. Приведите примеры того, как даже такая простая вещь, как целочисленные литералы, может стать довольно сложной задачей при реализации языка.
3. Точка с запятой часто используется для завершения выражений или для отделения соседних выражений друг от друга. Во многих популярных основных языках единственной наиболее распространенной синтаксической ошибкой является пропущенная точка с запятой. Опишите один или несколько способов, с помощью которых можно сделать точку с запятой ненужной в синтаксисе языка программирования.

4. Многие языки программирования определяют, что программа начинается с функции с именем `main()`. Java необычен тем, что хотя выполнение программы начинается с функции `main()`, каждый класс может иметь свою собственную процедуру `main()`, которая является другим способом запуска программы. Есть ли польза от такой странной организации программы?
5. Большинство языков имеют автоматические, предварительно открытые файлы для стандартного ввода, стандартного вывода и сообщений об ошибках. В современных компьютерах, однако, эти предварительно открытые файлы могут не иметь заметного значения, и программа, скорее всего, будет использовать предварительно открытые стандартные ресурсы сети, базы данных или графического окна. Объясните, практично ли это предложение, и почему.

Глава 3

Сканирование исходного кода

Первым шагом в любом языке программирования является чтение отдельных символов исходного кода и определение того, какие символы сгруппированы. В естественном языке это включает в себя просмотр соседних последовательностей букв для идентификации слов. В языке программирования группы символов образуют имена переменных, зарезервированные слова или, иногда, операторы либо знаки препинания длиной в несколько символов. В этой главе вы узнаете, как использовать *сопоставление с образцом* для чтения исходного кода и идентификации слов и знаков препинания из исходных символов.

В данной главе мы рассмотрим следующие основные темы:

- лексемы, лексические категории и токены;
- регулярные выражения;
- использование UFlex и JFlex;
- написание сканера для Jzero;
- регулярных выражений не всегда достаточно.

Во-первых, давайте рассмотрим несколько видов слов, которые появляются в исходном коде программы. Подобно тому, как читающий на естественном языке должен отличать существительные от глаголов и прилагательных, чтобы понять, что означает предложение, ваш язык программирования должен классифицировать каждую сущность в исходном коде, чтобы определить, как ее интерпретировать.

Технические требования

В этой главе мы рассмотрим некоторые реальные технические требования. Для этого вам потребуется установить некоторые инструменты и загрузить примеры. Давайте начнем с рассмотрения того, как установить *UFlex* и *JFlex*.

UFlex поставляется вместе с Unicon и не требует отдельной установки. Вы можете скачать примеры этой книги из нашего репозитория GitHub: *https://github.com/PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch3*.

Видеоролик «Код в действии» (Code in Action) для этой главы можно найти здесь: *https://bit.ly/3Fnn2c2*.

Для установки JFlex загрузите архив *jflex-1.8.2.tar.gz* (или более новый) с сайта *http://jflex.de/download.html*. Вам нужно будет распаковать его из файла **.tar.gz** в файл **.tar** с помощью приложения *gunzip*, а затем извлечь файлы из файла **.tar** с помощью приложения *tar*. Файл распакуется в подкаталог под каталогом, в котором запущен *tar*.

Например, вы увидите подкаталог с именем *jflex-1.8.2*. В Windows, куда бы вы ни распаковали JFlex, если вы не переместите свою установку JFlex в *C:\JFLEX*, вам нужно будет указать в значении переменной окружения **JFLEX_HOME** то место, куда вы его устанавливаете, а также поместить каталог *JFLEX\bin* в **PATH**. В Linux вы можете добавить каталог *JFLEX/bin* в **PATH** или создать символическую ссылку на скрипт *JFLEX\bin\jflex*.

Если вы распаковали JFlex в */home/myname/jflex-1.8.2*, то можете сделать символическую ссылку с */usr/bin/jflex* на распакованный скрипт */home/myname/jflex-1.8.2/bin/jflex*:

```
sudo ln -s /home/myname/jflex-1.8.2/bin/jflex /usr/bin/jflex
```

Ранее мы упоминали, что примеры в этой книге будут представлены *параллельно* на Unicon и Java. На печатной странице не хватает горизонтального пространства, чтобы показать коды рядом. Вместо этого сначала будет приведен пример на языке Unicon, а затем соответствующий код на Java. Обычно код Unicon представляет собой хороший исполняемый *псевдокод*, на котором можно основывать реализацию на Java. Теперь, когда у вас установлены и готовы к работе UFlex и/или JFlex, пришло время обсудить то, что мы делаем. Итак, мы поговорим о том, как использовать UFlex и JFlex для генерации кода сканера.

Лексемы, лексические категории и токены

Языки программирования читают символы и группируют соседние символы вместе, если они являются частью одной и той же сущности в языке. Это может быть многосимвольное имя или зарезервированное слово, постоянное значение или оператор.

Лексема – это строка соседних символов, которые образуют единую сущность. Большинство знаков препинания являются лексемами сами по себе, в дополнение к отделению того, что было до них, от того, что идет после них. В разумных языках пробельные символы, такие как пробелы и символы табуляции, игнорируются, кроме случаев разделения лексем. Почти все языки также имеют возможность включать комментарии в исходный код, и комментарии обычно рассматриваются так же, как и пробельные символы. Они могут быть границей, разделяющей две лексемы, но отбрасываются и не рассматриваются далее.

Каждая лексема имеет *лексическую категорию*. В естественных языках лексические категории называются частями речи. В реализации языка программирования лексическая категория обычно представлена целочисленным кодом и используется при синтаксическом анализе. Имена переменных являются лексической категорией. Константы представляют собой по крайней мере одну категорию, в большинстве языков существует несколько различных категорий для различных типов константных данных. Большинство зарезервированных слов имеют свою собственную категорию, потому что они разрешены в разных местах синтаксиса. Аналогично операторы обычно получают по крайней мере одну категорию для каждого уровня приоритета,

и часто каждому оператору присваивается своя категория. Типичный язык программирования имеет от 50 до 100 различных лексических категорий, что намного больше, чем количество частей речи, характерных для большинства естественных языков.

Пакет информации, который язык программирования собирает для каждой лексемы, которую он читает в исходном коде, называется *токеном*. Токены обычно представлены структурой (указателем) или объектом. Поля в токене включают следующее:

- лексему (строку);
- категорию (целое число);
- имя файла (строку);
- номер строки (целое число);
- возможно, другие данные.

Читая книги о языках программирования, вы можете обнаружить, что некоторые авторы используют слово token в различных смыслах для обозначения строки (лексемы), целочисленной категории или структуры/объекта (токена), в зависимости от контекста. Имея словарь лексем, категорий и токенов, мы можем рассмотреть нотацию, которая используется для ассоциирования наборов лексем с соответствующими категориями. Шаблоны в этой нотации называются регулярными выражениями.

Регулярные выражения

Регулярные выражения – это наиболее широко используемые обозначения для описания шаблонов символов в файлах. Они формулируются на основе очень простых правил, которые легко понять. Набор символов, над которыми записывается набор регулярных выражений, называется алфавитом. Для простоты в этой книге в качестве алфавита для чтения исходного кода мы будем использовать значения 0–255, которые могут храниться в одном байте.

В некоторых наборах входных символов регулярные выражения представляют собой шаблоны, которые описывают наборы строк, используя члены набора входных символов и несколько операторов регулярных выражений. Поскольку регулярные выражения представляют собой нотацию для множеств, такие термины, как *член*, *объединение* или *пересечение*, применимы для наборов строк, которым могут соответствовать регулярные выражения. В этом разделе мы рассмотрим правила построения регулярных выражений, а затем приведем примеры.

Правила регулярных выражений

В этой книге будут показаны только те операторы, которые необходимы для примеров. Это будет практическое подмножество регулярных выражений, о которых в теоретических книгах говорится, что это все, что нужно. Иметь практическое подмножество операторов, встречающихся в реализациях регулярных выражений некоторых инструментов, – это излишество. Правила регулярных выражений, которые мы будем рассматривать, следующие. После первого правила все остальные относятся к объединению регулярных выра-

жений в более крупные регулярные выражения, которые соответствуют более сложным шаблонам.

- Любой символ, например `a` из алфавита, является регулярным выражением, соответствующим этому символу. Обычный escape-символ, обратная косая черта (\), превращает оператор в регулярное выражение, которое соответствует только символу этого оператора.
- Вокруг регулярного выражения (`r`) можно поставить круглые скобки, чтобы оно соответствовало тому же самому, что и `r`. Это используется для того, чтобы установить *приоритет* операторов регулярного выражения внутри скобок таким, дабы они применялись перед операторами вне круглых скобок.
- Когда два регулярных выражения, `re1` и `re2`, находятся рядом, результирующий шаблон, `re1 re2`, соответствует экземпляру левого регулярного выражения, за которым следует экземпляр правого регулярного выражения. Это называется *конкатенацией* и является хитрым, потому что это невидимый или неявный оператор. Произвольная строка, заключенная в двойные кавычки, – это последовательность символов, заключенная в конкатенацию. Операторы регулярного выражения не применяются внутри двойных кавычек, но можно использовать обычные escape-последовательности, такие как `\n`.
- Между любыми двумя регулярными выражениями, `re1` и `re2`, может быть помещена вертикальная полоса для создания регулярного выражения, `re1 | re2`, которое соответствует `re1` или `re2`. Это называется *чередованием*, поскольку оно допускает любую из альтернатив. Квадратные скобки используются как специальное сокращение для регулярных выражений, состоящих из множества операторов вертикальных полос – `[abcd]` эквивалентно (`a|b|c|d`), то есть либо `a`, либо `b`, либо `c`, либо `d`. У этой стенограммы также есть сокращение – регулярное выражение `[a-d]` является еще более коротким эквивалентом (`a|b|c|d`), а регулярное выражение `[^abcd]` означает любой символ, который не является ни `a`, ни `b`, ни `c`, ни `d`. Полезным сокращением для сокращения сокращений является символ точки, или точка (`.`). Символ точки, или точка, `.`, эквивалентен `[^\n]` и соответствует любому символу, кроме новой строки.
- За любым регулярным выражением, `re`, может следовать `*`, или оператор *звездочка*. Регулярное выражение `re*` соответствует нулю или более вхождений регулярного выражения `re`. Аналогично за любым регулярным выражением может следовать знак плюс. Регулярное выражение `re+` соответствует одному или нескольким вхождениям этого регулярного выражения.

Эти правила ничего не говорят о наличии пробельных символов в регулярных выражениях или комментариях. В языках программирования есть такие вещи, но они не являются частью нотации регулярных выражений! Если вам нужен символ пробела как часть шаблона, который вы сопоставляете, конечно, его можно экранировать, заключить в двойные кавычки или квадратные скобки. Но если вы видите комментарий или пробел, который не экранирован в регулярном выражении, то это ошибка. Если вы хотите

вставить пробелы в регулярное выражение только для того, чтобы сделать его более красивым, вы не можете этого сделать. Если вам необходимо написать комментарий, чтобы объяснить, что делает регулярное выражение, то, вероятно, вы слишком усложнили свое регулярное выражение, регулярные выражения должны быть самодокументируемыми. Если ваши выражения таковыми не являются, вам следует прекратить то, что вы делаете, пойти домой и переосмыслить свою жизнь.

Несмотря на мои аргументы в пользу простоты, пять простых правил для формирования регулярных выражений можно комбинировать различными способами, чтобы сформировать мощные шаблоны, которые соответствуют очень интересным наборам строк. Прежде чем мы погрузимся в инструменты генератора лексического анализатора, которые их используют, рассмотрим несколько дополнительных примеров, которые дадут вам представление о некоторых видах шаблонов, описываемых регулярными выражениями.

Примеры регулярных выражений

Регулярные выражения становятся просты, как только вы напишете несколько из них. Вот некоторые из них, которые могут быть использованы в вашем сканере:

- регулярное выражение *while* представляет собой конкатенацию пяти регулярных выражений, по одному для каждой буквы – *w*, *h*, *i*, *l* и *e*. Оно соответствует строке *«while»*;
- регулярное выражение *«+»|»-»|»*»|»/»* соответствует строке из одного символа, которым является либо плюс, либо минус, либо звездочка, либо косая черта. Двойные кавычки используются для того, чтобы ни один из этих знаков препинания не интерпретировался как оператор регулярного выражения. Вы можете указать тот же шаблон так: *[+\-*/]*. Операторы регулярных выражений, такие как *, не применяются внутри квадратных скобок, но такие символы, как минус или каретка, которые имеют специальную интерпретацию внутри квадратных скобок, должны быть экранированы обратной косой чертой;
- регулярное выражение *[0-9]*\.[0-9]** соответствует нулю или более цифрам, за которыми следует точка, за которой следуют ноль или более цифр. Точка экранируется, потому что в противном случае она означает любой символ, кроме новой строки. Хотя этот шаблон выглядит как хорошая попытка сопоставить действительные числа, он позволяет точке быть без каких-либо цифр с обеих сторон! Вам придется сделать что-то получше. Это довольно громоздко, я признаю, сказать *([0-9]+\.[0-9]*|[0-9]*\.[0-9]+)*, но по крайней мере вы знаете, что токен будет каким-то числом;
- регулярное выражение *«\»[^»]*»\»»* соответствует символу двойной кавычки, за которым следует ноль или более вхождений любого символа, не являющегося двойной кавычкой, за которым следует символ двойной кавычки. Это типичная попытка новичка создать регулярное выражение для строковых констант. Одна вещь, которая в ней неверна, заключается в том, что она допускает новые строки в середине строки, чего большинство языков программирования не допускают. Другая проблема – нет

возможности поместить символ двойной кавычки внутри строковой константы. Большинство языков программирования предоставляют механизм экранирования, который позволяет это сделать. Как только вы начинаете разрешать экранированные символы, вы должны быть очень внимательны. Чтобы просто разрешить экранированные двойные кавычки, вы можете написать *«\»»([^»\\\n]|\\»)*»\»»*. Более общая версия для такого языка, как C, может выглядеть ближе к *«\»([^\\\\n]|\\\\([abfnrtv\\?0]|[0-7][0-7][0-7]|x[0-9a-fA-F][0-9a-fA-F]))*»\»»*.

Эти примеры показывают, что регулярные выражения варьируются от тривиальных до гигантских. Регулярные выражения являются чем-то вроде нотации, предназначенной только для записи – читать их гораздо сложнее, чем писать. Иногда, если вы ошиблись в регулярном выражении, проще переписать его с нуля, чем пытаться отладить. После рассмотрения нескольких примеров регулярных выражений пришло время узнать об инструментах, использующих нотацию регулярных выражений для создания сканеров, применяемых для чтения исходного кода, а именно UFlex и JFlex.

Использование UFlex и JFlex

Написание сканера вручную – это интересная задача для программиста, который хочет знать, как именно все работает, но это замедлит развитие вашего языка и усложнит последующее сопровождение кода.

Хорошие новости! Семейство инструментов, произошедших от UNIX, известное как *lex*, принимает регулярные выражения и генерирует для вас функцию сканера. Совместимые с lex инструменты доступны для большинства популярных языков программирования. Для C/C++ наиболее широко используемым lex-совместимым инструментом является *Flex*, размещенный на сайте *https://github.com/westes/flex/*. Для Unicon мы используем *UFlex*, а для Java можно использовать *JFlex*. Эти инструменты могут иметь различные пользовательские расширения, но в той степени, в которой они совместимы с UNIX lex, мы можем представить их вместе, как один язык для написания сканеров. Примеры в этой книге были тщательно подобраны таким образом, что мы даже можем использовать один и тот же lex-ввод для реализации как в Unicon, так и в Java!

Входные файлы для lex часто называют (lex-) спецификациями. Они используют расширение *.l* и состоят из нескольких разделов, разделенных %%. В этой книге в общем случае делаются ссылки на lex-спецификации, имея в виду входной файл, предоставляемый для UFlex или JFlex, и в большинстве случаев эти файлы также являются входными для C Flex.

В спецификации lex есть обязательные разделы – *раздел заголовка*, за которым следует *раздел регулярных выражений*, и необязательный *раздел вспомогательных функций*. JFlex добавляет *раздел импорта*, потому что Java нуждается в импорте и требует отдельного места для вставки фрагментов кода перед классом и внутри определения класса. Секция заголовка lex и секция регулярных выражений – это те разделы, о которых вам нужно знать прямо сейчас. Начнем с изучения раздела заголовка.

Раздел заголовка

Большинство инструментов Flex имеют опции, которые можно включить в разделе заголовка. Они различны, и мы расскажем о них только в том случае, если мы их используем. Вы также можете включить туда фрагменты кода хост-языка, например объявления переменных. Но основная цель раздела заголовка – определить именованные макросы для шаблонов, которые могут появляться несколько раз в разделе регулярных выражений. В lex эти именованные макросы располагаются в отдельных строках в следующем виде:

```
имя            регулярное выражение
```

В строке макроса *имя* – это последовательность букв, затем один или несколько пробелов и потом *регулярное выражение*. Позже, в разделе регулярных выражений, эти макросы можно подставить в регулярное выражение, окружив имя фигурными скобками, например {name}. Самая распространенная ошибка, которую допускают новички при работе с макросами lex, заключается в том, что они пытаются вставить комментарий после регулярного выражения, поэтому не делайте этого. Язык lex не поддерживает комментарии в таких строках и будет пытаться интерпретировать то, что вы напишете, как часть регулярного выражения.

JFlex нарушает совместимость и требует знака равенства после имени, поэтому его макросы выглядят следующим образом:

```
имя = регулярное выражение
```

Эта несовместимость с UNIX lex настолько вопиюща, что мы решили не использовать макросы в этой книге. Во время написания данной книги мы расширили UFlex для работы с макросами с любым синтаксисом. Если вы добавите несколько макросов, то приведенный здесь код можно немного сократить. Без макросов ваш раздел заголовка будет почти пустым, поэтому давайте рассмотрим следующую часть спецификации lex – раздел регулярных выражений.

Раздел регулярных выражений

Основной раздел спецификации lex – это раздел регулярных выражений. Каждое регулярное выражение приводится в отдельной строке, за ним следует пробел, затем семантическое действие, состоящее из кода на языке хоста (в нашем случае Unicon или Java), который должен быть выполнен, когда это регулярное выражение было встречено. Обратите внимание, что хотя каждое правило регулярного выражения начинается с новой строки, если в семантическом действии используются фигурные скобки для выделения блока операторов, который может охватывать несколько строк исходного кода, lex не начнет искать следующее регулярное выражение, пока не будет найдена соответствующая закрывающая фигурная скобка.

Самая распространенная ошибка, которую допускают новички в разделе регулярных выражений, заключается в том, что они пытаются вставить пробелы или комментарии в регулярное выражение, чтобы улучшить читабельность. Не делайте этого, вставка пробела в середину регулярного выражения обры-

вает регулярное выражение на пробеле, а остальная часть регулярного выражения интерпретируется как код языка хоста. Вы можете получить несколько загадочных сообщений об ошибках, если сделаете это.

Когда вы запускаете UNIX lex, который является инструментом C, он генерирует функцию `yylex()`, которая возвращает целочисленную категорию для каждой лексемы, а глобальные переменные содержат другую полезную информацию. Целое число, называемое `yychar`, содержит категорию, строка `yytext` содержит символы, которые были найдены для данной лексемы, а `yyleng` сообщает, сколько символов было найдено. Инструменты lex различаются по своей совместимости с этим общедоступным интерфейсом, и некоторые инструменты автоматически вычисляют дополнительные данные. Например, JFlex должен генерировать сканер внутри класса и предоставляет `yytext()` в качестве функции-члена. Языки программирования, безусловно, захотят получить больше деталей, например номер строки, с которой пришел токен. Теперь пришло время проработать примеры, которые помогут нам в этом.

Написание простого сканера исходного кода

Этот пример позволяет проверить, можете ли вы запустить UFlex и JFlex. Он также помогает установить, в какой степени их использование сходно или несходно. Сканер примера просто распознает имена, числа и пробелы, файл с именем nnws.l будет использоваться для спецификации lex. Первое, что вы должны сделать при чтении исходного кода, – это определить категорию каждой лексемы и вернуть, какая категория была найдена. В данном примере возвращается 1 для имени и 2 для числа. Пробелы отбрасываются. Все остальное является ошибкой.

В этом разделе приводится тело файла nnws.l. Эта спецификация будет работать в качестве входных данных для UFlex и JFlex. Поскольку семантические действия для UFlex – это код Unicon, а для JFlex – это код Java, это требует некоторой сдержанности. Семантическое действие будет законным как в Java, так и в Unicon, но только если вы ограничите код семантического действия их общим синтаксисом, таким как вызовы методов и возврат выражений. Если вы начнете вставлять операторы `if` или присваивания и синтаксис, специфичный для конкретного языка, ваша спецификация lex станет специфичной для одного языка-хоста, Unicon либо Java.

Даже этот короткий пример содержит некоторые идеи, которые пригодятся нам в дальнейшем. Первые две строки предназначены для JFlex и игнорируются UFlex. Начальный `%%` завершает пустой раздел импорта JFlex. Вторая строка – это опция JFlex в разделе заголовка. По умолчанию функция `yylex()` в JFlex возвращает объект типа Yytoken, а опция `%int` указывает ей возвращать тип integer. Третья строка, начинающаяся с `%%`, переводит нас непосредственно в раздел регулярных выражений. В четвертой строке регулярное выражение `[a-zA-Z]+` соответствует одной или нескольким строчным или заглавным буквам, оно сопоставляет столько соседних букв, сколько может найти, и возвращает 1. Как побочный продукт совпавшие символы будут сохранены в переменной yytext. В пятой строке регулярное выражение `[0-9]+` сопоставляет столько цифр, сколько может найти, и возвращает 2. В шестой строке пробельные символы совпадают с регулярным выражением `[ \t\r\n]+`, и ничего

не возвращается, сканер продолжает просматривать входной файл в поисках следующей лексемы в соответствии с другим регулярным выражением. Вы, вероятно, знаете иные пробельные символы, кроме символа пробела внутри квадратных скобок, но \t – это символ табуляции, \r – символ возврата каретки и \n – символ новой строки. Точка (.) в седьмой строке будет соответствовать любому символу, кроме новой строки, таким образом, будет отслежен любой исходный код, который не был разрешен ни в одном из предыдущих шаблонов, и в этом случае будет сгенерировано сообщение об ошибке. О лексических ошибках сообщается с помощью функции lexErr() в объекте с именем simple. Для последующих этапов работы нашего компилятора нам понадобятся дополнительные функции сообщений об ошибках.

```
%%
%int
%%
[a-zA-Z]+     { return 1; }
[0-9]+                { return 2; }
[ \t\r\n]+    { }
.             { simple.lexErr("unrecognized character"); }
```

Эта спецификация будет вызываться из функции main(), по одному разу для каждого слова во входных данных. Каждый раз, когда она вызывается, она будет сопоставлять текущий ввод со всеми регулярными выражениями (четырьмя в данном случае) и выбирать то регулярное выражение, которое соответствует наибольшему количеству символов в текущем местоположении. Если два или более регулярных выражений сравняются по наибольшему количеству совпадений, победит то из них, которое появится первым в файле спецификации.

Различные инструменты lex могут предоставлять функцию main() по умолчанию, но для полного контроля вам следует написать свою собственную. Написание нашей собственной функции main() также позволяет продемонстрировать на примере, как обращаться к yylex() из отдельного файла. Вам нужно будет уметь это делать, чтобы подключить ваш сканер к синтаксическому анализатору в следующей главе.

Функция main() зависит от языка. В Unicon используется модель организации программы в стиле C++, в которой функция main() начинается вне любого объекта, в то время как Java искусственно помещает функции main() внутри классов, но в остальном коды Unicon и Java имеют много общего.

Реализация функции main() в Unicon может быть помещена в файл с любым именем и расширением .icn, назовем этот файл simple.icn. Этот файл содержит процедуру main() и одноэлементный класс с именем simple, который нужен только потому, что в файле nnws.l мы вызвали вспомогательную функцию лексической ошибки совместимым с Java способом, то есть simple.lexErr(). Процедура main() инициализирует класс simple, заменяя функцию конструктора класса на единственный экземпляр, возвращаемый этой функцией. Затем main() открывает входной файл с именем, заданным в первом аргументе командной строки. Лексический анализатор информируется о том, какой файл читать, с помощью yyin. Затем код вызывает yylex() в цикле до тех пор, пока не завершится работа сканера:

```
procedure main(argv)
   simple := simple()
   yyin := open(argv[1])
   while i := yylex() do
      write(yytext, ": ", i)
end
   class simple()
      method lexErr(s)
         stop(s, ": ", yytext)
      end
end
```

Соответствующий `main()` в Java должен быть помещен в класс, а имя файла должно быть именем класса с расширением .java. Мы назовем этот файл simple.java. Файл открывается путем создания объекта FileReader и подключения его к лексическому анализатору путем передачи объекта FileReader в качестве параметра при создании объекта Yylex лексического анализатора. Поскольку FileReader может не сработать, мы должны объявить, что `main()` выдает исключение. После создания объекта Yylex `main()` обращается к `yylex()` снова и снова, пока входные данные не будут исчерпаны, о чем свидетельствует значение стража (sentinel value[1]) Yylex.YYEOF, возвращаемое из `yylex()`. Несмотря на немного большую длину, `main()` делает то же самое, что и в версии Unicon. По сравнению с простым классом Unicon, версия Java имеет дополнительный прокси-метод yytext(), чтобы другие функции в простом классе или остальной части компилятора могли получить доступ к самой последней строке лексемы, не имея ссылки на объект Yylex простого класса:

```
import java.io.FileReader;
public class simple {
      static Yylex lex;
      public static void main(String argv[]) throws Exception{
              lex = new Yylex(new FileReader(argv[0]));
              int i;
              while ((i=lex.yylex()) != Yylex.YYEOF)
                      System.out.println("token "+ i +": "+ yytext());
      }
      public static String yytext() {
              return lex.yytext();
      }
      public static void lexErr(String s) {
              System.err.println(s + ": " + yytext());
              System.exit(1);
      }
}
```

[1] Sentinel value – это специальное значение в контексте алгоритма, который использует его присутствие в качестве условия завершения, как правило, в цикле или рекурсивном алгоритме. – *Прим. перев.*

Этот простой сканер предназначен главным образом для того, чтобы показать вам, как подключены все эти средства. Чтобы убедиться в том, что эти средства работают так, как задумано, их нужно запустить и выяснить это.

Запуск сканера

Давайте запустим этот пример на следующем (тривиальном) входном файле с именем dorrie.in:

```
Dorrie is 1 fine puppy
```

Прежде чем запустить программу, ее необходимо скомпилировать. UFlex и JFlex записывают Unicon- и Java-код, который вызывается из остальной части вашего языка программирования, написанного либо на Unicon, либо на Java. Процесс компиляции показан на рис. 3.1. В Unicon два исходных файла компилируются и соединяются вместе в исполняемый файл с именем `simple`. В Java два файла компилируются в отдельные файлы `.class`, вы запускаете Java на файле `simple.class`, где находится метод `main()`, и он загружает другие файлы по мере необходимости.

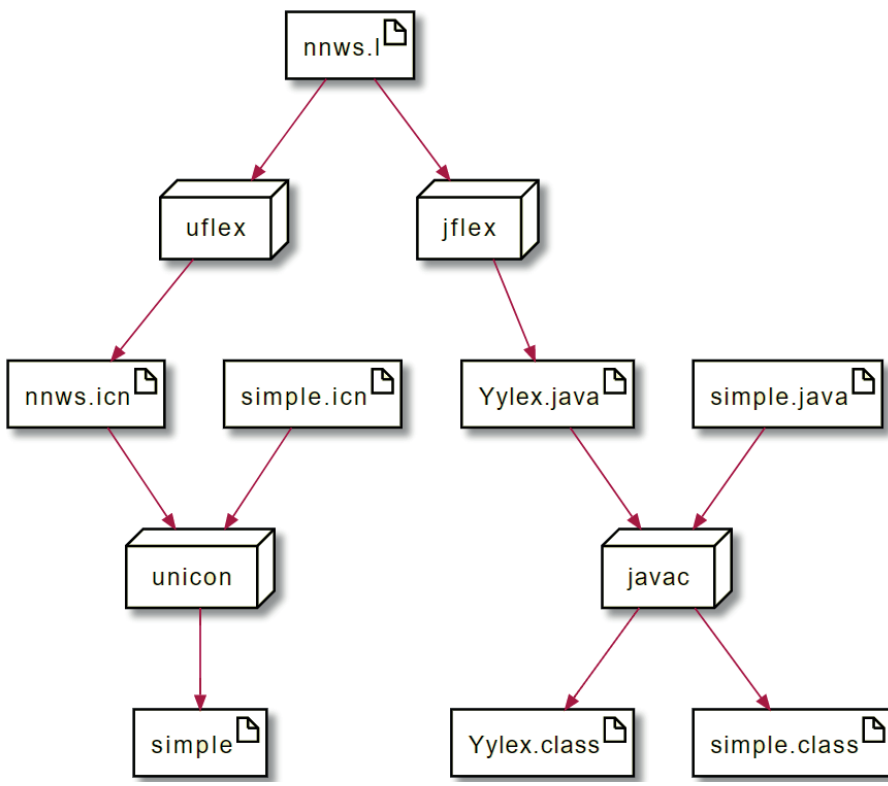


Рис. 3.1. Файл nnws.l используется для создания программ Unicon (слева) и Java (справа)

Вы можете скомпилировать и запустить программу в Unicon или Java, используя левый или правый столбец, как показано ниже:

```
uflex nnws.l
unicon simple nnws
simple dorrie.in
```

```
jflex nnws.l
javac simple.java Yylex.java
java simple dorrie.in
```

При любой реализации вы должны увидеть пять показанных ниже строк:

```
token 1: Dorrie
token 1: is
token 2: 1
token 1: fine
token 1: puppy
```

Пока что все, что делает пример, – это классифицирует группы входных символов с помощью регулярного выражения для определения типа найденной лексемы. Для работы остальной части компилятора нам понадобится дополнительная информация о лексеме, которую мы будем хранить в токене.

Токены и лексические атрибуты

В дополнение к определению того, к какой целочисленной категории принадлежит каждая лексема, остальная часть реализации языка программирования (в нашем случае компилятор) требует от сканера выделить объект, содержащий всю связанную с лексемой информацию. Этот объект называется *токеном*.

Токен содержит группу именованных полей, называемых *лексическими атрибутами*. Элементы информации, которые должны быть записаны о данной лексеме, зависят от языка и реализации. Обычно токены отслеживают целочисленную категорию, строковую лексему и номер строки, из которой взята лексема. В реальном компиляторе токены обычно содержат дополнительную информацию о лексеме. Это, скорее всего, имя файла и столбец в строке, где встретилась лексема. Для некоторых лексем (литеральных констант) компилятор или интерпретатор может посчитать полезным сохранить фактическое двоичное значение, представленное этим литералом.

Вам может быть интересно, зачем хранить в строке, из какого столбца пришла лексема. Учитывая сам текст лексемы, вы обычно можете достаточно легко узнать это, просто посмотрев на строку исходного кода, а большинство компиляторов указывают только номера строк, когда сообщают об ошибках, а не номера столбцов. По правде говоря, не все реализации языков программирования хранят номера столбцов в своих лексических атрибутах. Однако те, которые хранят, могут различать ошибки, когда одна и та же лексема появляется в строке более одного раза – ошибка в первой закрывающей скобке или в третьей? Можно оставить это на откуп человеку, а можно записать дополнительные детали. Выбор хранения информации о колонках также может зависеть от того, будет ли ваш лексический анализатор использоваться в IDE[1], которая при возникновении ошибки переводит курсор на ошибочный токен. Если это одно

[1] IDE (Integrated Development Environment) – это интегрированная, единая среда разработки. IDE представляет собой комплекс из нескольких инструментов – текстового редактора, компилятора либо интерпретатора, средств автоматизации сборки и отладчика. – *Прим. перев.*

из ваших требований, то вам понадобится информация о столбцах для обеспечения корректности этой функции.

Расширение нашего примера для построения токенов

Для каждого вызова `yylex()` выделяется новый экземпляр токена. В lex токены передаются синтаксическому анализатору путем помещения указателя на новый экземпляр в глобальную переменную с именем `yylval` при каждом вызове `yylex()`. В качестве перехода к реальному сканеру языка программирования мы расширим приведенный ранее пример, чтобы он выделял эти объекты токенов. Наиболее элегантный и переносимый способ сделать это – вставить функцию `scan()` в семантические действия. Функция `scan()` выделяет объекты токенов и затем (обычно) возвращает свой параметр, который является кодом целочисленной категории в предыдущем примере.

Спецификацию lex для этого можно найти в файле `nnws-tok.l`. Интересно, что в JFlex символ возврата каретки не является ни частью новой строки, ни частью оператора точки (.), поэтому, если вы пользуетесь JFlex, вы должны учитывать возвраты каретки в явном виде. В этом примере они необязательны перед новыми строками:

```
%%
%int
%%
[a-zA-Z]+     { return simple2.scan(1); }
[0-9]+        { return simple2.scan(2); }
[ \t]+        { }
\r?\n         { simple2.increment_lineno(); }
.             { simple2.lexErr("unrecognized character"); }
```

Пересмотренная процедура `main()` в Unicon показана в следующем файле `simple2.icn`. Функция `scan()` зависит от глобальной переменной `yylineno`, которая устанавливается из функции `main()` и обновляется в `yylex()` каждый раз, когда встречается новая строка. Как и в предыдущем примере, класс `simple2` является одноэлементным классом, который здесь находится для того, чтобы спецификация lex могла работать без изменений как для Unicon, так и для Java. Представление токенов определяется типом записи Unicon, который похож на *struct* в C/C++ или класс без методов. Пока что он содержит только целочисленный код категории, саму строку лексемы и номер строки, из которой она взята:

```
global yylineno, yylval
procedure main(argv)
      simple2 := simple2()
      yyin := open(argv[1]) | stop("usage: simple2 filename")
      yylineno := 1
      while i := yylex() do
              write(yytext, " (line ",yylval.lineno, "): ", i)
end
class simple2()
      method lexErr(s)
              stop(s, ": line ", yylineno, ": ", yytext)
      end
```

```
      method scan(cat)
              yylval := token(cat, yytext, yylineno)
              return cat
      end
         method increment_yylineno()
              yylineno +:= 1
      end
end
record token(cat, text, lineno)
```

Соответствующий Java `main()` в файле `simple2.java` выглядит следующим образом:

```
import java.io.FileReader;
public class simple2 {
      static Yylex lex;
      public static int yylineno;
      public static token yylval;
      public static void main(String argv[]) throws Exception {
              lex = new Yylex(new FileReader(argv[0]));
              yylineno = 1;
              int i;
              while ((i=lex.yylex()) != Yylex.YYEOF)
                      System.out.println("token "+ i +
                              " (line " +yylval.lineno + "): "+ yytext());
      }
      public static String yytext() {
              return lex.yytext();
      }
      public static void lexErr(String s) {
              System.err.println(s + ": line " + yylineno +
                      ": " + yytext());
              System.exit(1);
      }
      public static int scan(int cat) {
              yylval = new token(cat, yytext, yylineno);
              return cat;
      }
      public static void increment_lineno() {
              yylineno++;
      }
}
```

Для примера `simple2` требуется еще один файл Java. Файл `token.java` содержит наше представление токена класса. Этот токен класса будет расширен в следующем разделе.

```
public class token {
      public int cat;
      public String text;
      public int lineno;
```

```
        public token(int c, String s, int l) {
                cat = c; text = s; lineno = l;
        }
}
```

Следующий входной файл, `dorrie2.in`, был расширен до нескольких строк, и в него добавлена точка, чтобы можно было видеть номер строки, когда сообщается о нераспознанных символах.

```
Dorrie
is 1
fine puppy.
```

Вы можете запустить программу на Unicon или Java следующим образом:

```
uflex nnws-tok.l                jflex nnws-tok.l
                                javac token.java
unicon simple2 nnws-tok         javac simple2.java Ylex.java
simple2 dorrie2.in              java simple2 dorrie2.in
```

При любом варианте реализации вы должны увидеть такой результат:

```
token 1 (line 1): Dorrie
token 1 (line 2): is
token 2 (line 2): 1
token 1 (line 3): fine
token 1 (line 3): puppy
unrecognized character: line 3: .
```

Выходные данные этого примера включают номера строк, а входной файл содержит нераспознанный символ, поэтому мы можем видеть, что сообщение об ошибке также включает номер строки.

Написание сканера для Jzero

В этом разделе мы создадим сканер для Jzero, нашего подмножества языка Java. Это расширяет предыдущий пример simple2 до размера реального языка и добавляет информацию о столбцах, а также дополнительные лексические атрибуты для литеральных констант. Большим изменением является введение множества регулярных выражений для более сложных шаблонов, чем те, которые мы видели ранее. Весь язык Java распознается, но значительная часть категорий Java приводит к тому, что выполнение завершается с ошибкой, поэтому наша грамматика в следующей главе, а также остальные части компилятора не должны их учитывать.

Спецификация Jzero flex

По сравнению с предыдущими примерами, в реальной спецификации языка программирования lex будет содержаться гораздо большее число и более

сложных регулярных выражений. Следующий файл называется `javalex.l`, и он будет представлен в нескольких фрагментах.

Начало файла `javalex.l` включает заголовок и регулярные выражения для комментариев и пробельных символов. Эти регулярные выражения соответствуют и потребляют символы из исходного кода, не возвращая для них целочисленный код, они невидимы для остальной части компилятора. Будучи подмножеством Java, Jzero включает как комментарии в стиле C, ограниченные /* и */, так и комментарии в стиле C++, начинающиеся с // и идущие до конца строки. Регулярное выражение для комментариев C – это просто ужас. Если в вашем языке есть подобные шаблоны, то их очень легко и часто можно понять неправильно. Они читаются так: начинайте с /*, а затем игнорируйте куски символов без звездочек или звездочки до тех пор, пока они не заканчивают комментарий, и закончите, когда найдете звездочку (звездочки), за которой следует слеш.

```
%%
%int
%%
«/*»([^*]|»*»+[^/*])*»*»+»/»  { j0.comment(); }
«//».*\r?\n                    { j0.comment(); }
[ \t\r\f]+                     { j0.whitespace(); }
\n                             { j0.newline(); }
```

Следующая часть `javalex.l` содержит зарезервированные слова, регулярные выражения которых тривиальны. Поскольку эти слова часто встречаются в семантических действиях, используйте двойные кавычки, чтобы подчеркнуть, что это просто сами символы, а не код семантического действия. Многие из приведенных здесь кодов категорий целых чисел доступны из класса синтаксического анализатора, указанного в отдельном файле. В остальных главах этой книги целочисленные коды задаются синтаксическим анализатором. Лексический анализатор должен использовать коды синтаксического анализатора, чтобы эти две фазы компилятора успешно взаимодействовали.

Вы можете задаться вопросом: зачем использовать отдельный целочисленный код категории для каждого зарезервированного слова? Отдельный код категории нужен только для каждой уникальной роли в синтаксисе. Зарезервированные слова, которые могут использоваться в одних и тех же местах, могут использовать один и тот же целочисленный код категории. Если вы так сделаете, ваша грамматика будет короче, но вы отложите их различия на более поздний период семантического анализа и сделаете свою грамматику немного расплывчатой. Примером этого могут служить *true* и *false*. Синтаксически они являются одним и тем же видом, поэтому оба возвращаются как BOOLLIT. Мы можем найти другие зарезервированные слова, такие как имена типов, которым мы могли бы присвоить одинаковый код категории. Это решение нужно учитывать при проектировании. Если вы сомневаетесь, будьте осторожны и не будьте расплывчаты, присвоив каждому зарезервированному слову свой целочисленный код.

```
«break»             { return j0.scan(parser.BREAK); }
"double"            { return j0.scan(parser.DOUBLE); }
"else"                      { return j0.scan(parser.ELSE); }
"false"             { return j0.scan(parser.BOOLLIT); }
"for"                       { return j0.scan(parser.FOR); }
"if"                { return j0.scan(parser.IF); }
"int"                       { return j0.scan(parser.INT); }
"null"                      { return j0.scan(parser.NULLVAL); }
"return"            { return j0.scan(parser.RETURN); }
"string"            { return j0.scan(parser.STRING); }
"true"                      { return j0.scan(parser.BOOLLIT); }
"bool"                      { return j0.scan(parser.BOOL); }
"void"                      { return j0.scan(parser.VOID); }
"while"             { return j0.scan(parser.WHILE); }
"class"             { return j0.scan(parser.CLASS); }
"static"            { return j0.scan(parser.STATIC); }
"public"            { return j0.scan(parser.PUBLIC); }
```

Третья часть `javalex.l` состоит из операторов и знаков препинания. Регулярные выражения заключаются в кавычки, чтобы показать, что они означают только сами символы. Как и в случае с зарезервированными словами, в некоторых случаях операторы могут быть объединены в общий код категории, если они имеют одинаковый приоритет и ассоциативность. Это сделает грамматику короче, но за счет неясности. Еще одна сложность по сравнению с зарезервированными словами заключается в том, что многие операторы и знаки препинания состоят только из одного символа. В этом случае короче и читабельнее использовать их ASCII-код, что мы и делаем. Функция `j0.ord(s)` предоставляет способ сделать это, который работает как в Unicon, так и в Java. Для многосимвольных операторов определена константа синтаксического анализатора, как и для зарезервированных слов:

```
«(«         { return j0.scan(j0.ord("(")); }
")"         { return j0.scan(j0.ord(")")); }
"["         { return j0.scan(j0.ord("[")); }
"]"         { return j0.scan(j0.ord("]")); }
"{"         { return j0.scan(j0.ord("{")); }
"}"         { return j0.scan(j0.ord("}")); }
";"         { return j0.scan(j0.ord(";")); }
":"         { return j0.scan(j0.ord(":")); }
"!"         { return j0.scan(j0.ord("!")); }
"*"         { return j0.scan(j0.ord("*")); }
"/"         { return j0.scan(j0.ord("/")); }
"%"         { return j0.scan(j0.ord("%")); }
"+"         { return j0.scan(j0.ord("+")); }
"-"         { return j0.scan(j0.ord("-")); }
"<"         { return j0.scan(j0.ord("<")); }
"<="        { return j0.scan(parser.LESSTHANOREQUAL);}
">"         { return j0.scan(j0.ord(">")); }
">="        { return j0.scan(parser.GREATERTHANOREQUAL);}
"=="        { return j0.scan(parser.ISEQUALTO); }
"!="        { return j0.scan(parser.NOTEQUALTO); }
```

```
"&&"          { return j0.scan(parser.LOGICALAND); }
"||"          { return j0.scan(parser.LOGICALOR); }
"="           { return j0.scan(j0.ord("=")); }
"+="          { return j0.scan(parser.INCREMENT); }
"-="          { return j0.scan(parser.DECREMENT); }
","           { return j0.scan(j0.ord(",")); }
"."           { return j0.scan(j0.ord(".")); }
```

Четвертая (и последняя) часть `javalex.l` содержит более сложные регулярные выражения. Правило для имен переменных, целочисленной категорией которых является IDENTIFIER, должно идти после всех зарезервированных слов. Регулярные выражения зарезервированных слов перекрывают гораздо более общее регулярное выражение идентификатора, но только потому, что семантика lex разрушает связи, выбирая то регулярное выражение, которое стоит первым в спецификации lex.

Если это сделает ваш код более читабельным, вы можете иметь столько регулярных выражений, сколько захотите, возвращающих одну и ту же целочисленную категорию. В данном примере используется несколько регулярных выражений для действительных чисел, которые представляют собой числа либо с десятичной точкой, либо в научной нотации, либо и те, и другие. После последнего регулярного выражения используется универсальный шаблон, который генерирует лексическую ошибку, если в исходном коде появляются двоичные или другие странные символы:

```
[a-zA-Z_][a-zA-Z0-9_]*{ return j0.scan(parser.IDENTIFIER);}
[0-9]+ { return j0.scan(parser.INTLIT); }
[0-9]+».»[0-9]*([eE][+-]?[0-9]+)? { return j0.scan
      (parser.DOUBLELIT);}
[0-9]*"."[0-9]+([eE][+-]?[0-9]+)? { return j0.scan
      (parser.DOUBLELIT);}
([0-9]+)([eE][+-]?([0-9]+)) {return j0.scan
      (parser.DOUBLELIT);}
\"([^\"])|(\\.)*\"    { return j0.scan(parser.STRINGLIT); }
.                 { j0.lexErr("unrecognized character");}
```

Хотя для представления здесь файл `javalex.l` был разбит на четыре части, он является не очень длинным, около 58 строк кода. Поскольку он работает как для Unicon, так и для Java, это достаточно выгодно для вас при кодировании. Поддержка кода Unicon и Java нетривиальна, но здесь мы позволяем lex (UFlex и JFlex) сделать большую часть работы.

Код Unicon Jzero

Реализация сканера Jzero в Unicon находится в файле с именем `j0.icn`. Unicon имеет препроцессор и обычно вводит определенные символические константы через файлы `$include`. Чтобы использовать одну и ту же спецификацию lex в Unicon и Java, этот сканер Unicon создает объект `parser`, поля которого, такие как `parser.WHILE`, содержат целочисленный код категории:

```
global yylineno, yycolno, yylval
procedure main(argv)
        j0 := j0()
     parser := parser(257,258,259,260,261,262,263,264,265,
                      266, 267,268,269,270,273,274,275,276,
                      277,278,280,298,300,301,302,303,304,
                      306,307,256)
     yyin := open(argv[1]) | stop("usage: simple2 filename")
     yylineno := yycolno := 1
     while i := yylex() do
            write(yytext, ":",yylval.lineno, " ", i)
end
```

Вторая часть `j0.icn` состоит из класса `j0`. По сравнению с классом `simple2` из предыдущего примера `simple2.icn` были добавлены дополнительные методы для вызова семантических действий, когда встречаются различные пробельные символы и комментарии. Это позволяет вычислять номер текущего столбца в глобальной переменной `yycolno`:

```
class j0()
     method lexErr(s)
            stop(s, ": ", yytext)
     end
     method scan(cat)
            yylval := token(cat, yytext, yylineno, yycolno)
            yycolno +:= *yytext
            return cat
     end
     method whitespace()
            yycolno +:= *yytext
     end
     method newline()
            yylineno +:= 1; yycolno := 1
     end
     method comment()
            yytext ? {
                    while tab(find("\n")+1) do newline()
                    yycolno +:= *tab(0)
            }
     end
     method ord(s)
            return proc("ord",0)(s[1])
     end
end
```

В третьей части `j0.icn` тип токена был переведен из записи в класс, поскольку теперь он имеет дополнительную сложность в своем конструкторе, а также метод для обработки символов экранирования строк и вычисления двоичного представления констант строковых литералов. В Unicon код конструктора находится в конце метода в секции `initially`.

Метод deEscape() отбрасывает начальные и конечные символы двойных кавычек, а затем обрабатывает строковый литерал символ за символом, используя сканирование строки Unicon. Внутри управляющей структуры сканирования строк s ? { ... } строка s просматривается слева направо. Функция move(1) захватывает следующий символ из строки и перемещает позицию сканирования вперед на единицу. Более подробное объяснение сканирования строк приведено в *приложении «Основы Unicon»*.

В методе deEscape() обычные символы копируются из входной строки sin в выходную строку sout. Escape-символы вызывают один или несколько символов, которые интерпретируются по-разному. Подмножество Jzero работает только с символами табуляции и новой строки, Java имеет гораздо больше escape-символов, которые вы можете добавить. Есть что-то забавное в том, чтобы превратить обратный слеш, за которым следует «t», в символ табуляции, но каждый компилятор, который вы когда-либо использовали, должен был сделать что-то, подобное следующему:

```
class token(cat, text, lineno, colno, ival, dval, sval)
     method deEscape(sin)
             local sout := ""
             sin := sin[2:-1]
             sin ? {
                     while c := move(1) do {
                             if c == "\\" then {
                                     if not (c := move(1)) then
                                             j0.lexErr("malformed string
                                     literal")
                                     else case c of {
                                             "t":{ sout ||:= "\t" }
                                             "n":{ sout ||:= "\n" }
                                             }
                                     }
                             }
                             else sout ||:= c
                     }
             }
             return sout
     end
initially
     case cat of {
             parser.INTLIT: { ival := integer(text) }
             parser.DOUBLELIT: { dval := real(text) }
             parser.STRINGLIT: { sval := deEscape(text) }
     }
end
record parser(BREAK,PUBLIC,DOUBLE,ELSE,FOR,IF,INT,RETURN,VOID,
          WHILE,IDENTIFIER,CLASSNAME,CLASS,STATIC,STRING,
          BOOL,INTLIT,DOUBLELIT,STRINGLIT,BOOLLIT,
          NULLVAL,LESSTHANOREQUAL,GREATERTHANOREQUAL,
          ISEQUALTO,NOTEQUALTO,LOGICALAND,LOGICALOR,
          INCREMENT,DECREMENT,YYERRCODE)
```

Запись одноэлементного парсера здесь выглядит довольно глупо для опытного программиста Unicon, который может просто определить (*$define*) все эти имена категорий токенов и пропустить введение парсерного типа. Если вы программист Unicon, просто напомните себе, что это сделано для совместимости с Java, в частности совместимости с *byacc/j*.

Код Java Jzero

Java-реализация сканера Jzero включает главный класс в файле `j0.java`. Он похож на пример `simple2.java`. Здесь он представлен в четырех частях. Первая часть включает функцию `main()` и должна быть знакомой, за исключением добавления дополнительных переменных, таких как переменная `yycolno`, которая отслеживает номер текущего столбца:

```
import java.io.FileReader;
public class j0 {
      static Yylex lex;
      public static int yylineno, yycolno;
      public static token yylval;
      public static void main(String argv[]) throws Exception {
              lex = new Yylex(new FileReader(argv[0]));
              yylineno = yycolno = 1;
              int i;
              while ((i=lex.yylex()) != Yylex.YYEOF) {
                      System.out.println("token " + i + ": " +
                              yytext());
              }
      }
```

Класс `j0` продолжает несколько вспомогательных функций, которые были показаны в предыдущих примерах:

```
public static String yytext() {
        return lex.yytext();
}
public static void lexErr(String s) {
      System.err.println(s + ": line " + yylineno +
                          ": " + yytext());
      System.exit(1);
}
public static int scan(int cat) {
      last_token = yylval =
              new token(cat, yytext(), yylineno, yycolno);
      yycolno += yytext().length();
      return cat;
}
public static void whitespace() {
      yycolno += yytext().length();
      }
public short ord(String s) {return(short)(s.charAt(0));}
```

Функция класса `j0` для обработки символов новой строки в исходном коде длиннее, чем можно было бы ожидать. Конечно, она увеличивает номер строки и устанавливает столбец обратно в 1, но для вставки точки с запятой она теперь включает оператор `switch`, который определяет, следует вставлять точку с запятой вместо символа новой строки или нет. Метод обработки комментариев проходит символ за символом через комментарий, чтобы сохранить корректно номер строки и номер столбца:

```
        public static void newline() {
                yylineno++; yycolno = 1;
                if (last_token != null)
                        switch(last_token.cat) {
                        case parser.IDENTIFIER: case parser.INTLIT:
                        case parser.DOUBLELIT: case parser.STRINGLIT:
                        case parser.BREAK: case parser.RETURN:
                        case parser.INCREMENT: case parser.DECREMENT:
                        case ')': case ']': case '}':
                                return true;
                }
                return false;
        }
        public static void comment() {
                int i, len;
                String s = yytext();
                len = s.length();
                for(i=0; i<len; i++)
                        if (s.charAt(i) == '\n') {
                                yylineno++; yycolno=1;
                        }
                        else yycolno++;
        }
}
```

Существует вспомогательный модуль с именем `parser.java`. Он предоставляет набор именованных констант, подобно перечисляемому типу, но объявляет константы непосредственно, как короткие целые числа, чтобы они были совместимы с синтаксическим анализатором `iyacc`, который будет обсуждаться в следующей главе. Выбранные целые числа начинаются выше 256, потому что именно с этого числа начинается `iyacc`, и так, чтобы они не конфликтовали с целочисленными кодами однобайтовых лексем, которые мы создаем с помощью обращений к `j0.ord()`:

```
public class parser {
public final static short BREAK=257;
public final static short PUBLIC=258;
public final static short DOUBLE=259;
public final static short ELSE=260;
public final static short FOR=261;
public final static short IF=262;
public final static short INT=263;
public final static short RETURN=264;
```

```
public final static short VOID=265;
public final static short WHILE=266;
public final static short IDENTIFIER=267;
public final static short CLASSNAME=268;
public final static short CLASS=269;
public final static short STATIC=270;
public final static short STRING=273;
public final static short BOOL=274;
public final static short INTLIT=275;
public final static short DOUBLELIT=276;
public final static short STRINGLIT=277;
public final static short BOOLLIT=278;
public final static short NULLVAL=280;
public final static short LESSTHANOREQUAL=298;
public final static short GREATERTHANOREQUAL=300;
public final static short ISEQUALTO=301;
public final static short NOTEQUALTO=302;
public final static short LOGICALAND=303;
public final static short LOGICALOR=304;
public final static short INCREMENT=306;
public final static short DECREMENT=307;
public final static short YYERRCODE=256;
}
```

Существует также вспомогательный модуль `token.java`, который содержит класс `token`. Он расширился и включает в себя номер столбца, а для литеральных констант их двоичное представление хранится в `ival`, `sval` и `dval` для целых чисел, строк и чисел двойной точности соответственно. Метод `deEscape()`, который используется для построения двоичного представления строковых литералов, был рассмотрен в реализации этого класса в Unicon. Опять же, алгоритм проходит символ за символом и просто копирует символ, если только это не обратный слеш, в этом случае он захватывает следующий символ и интерпретирует его иначе. Вы можете убедиться в эффективности класса Java *String*, сравнив этот код с версией Unicon:

```
public class token {
             public int cat;
             public String text;
             public int lineno, colno, ival;
             String sval;
             double dval;
     private String deEscape(String sin) {
             String sout = "";
             sin = String.substring(sin,1,sin.length()-1);
             int i = 0;
             while (sin.length() > 0) {
                     char c = sin.charAt(0);
                     if (c == '\\') {
                             sin = String.substring(sin,1);
                             if (sin.length() < 1)
                                     j0.lexErr("malformed string literal");
```

```
                                else {
                                        c = sin.charAt(0);
                                        switch(c) {
                                        case 't': sout = sout + "\t"; break;
                                        case 'n': sout = sout + "\n"; break;
                                        default: j0.lexErr("unrecognized escape");
                                        }
                                }
                                else sout = sout + c;
                        }
                }
                return sout;
        }
        public token(int c, String s, int ln, int col) {
                cat = c; text = s; lineno = ln; colno = col;
                switch (cat) {
                case parser.INTLIT:
                        ival = Integer.parseInt(s);
                        break;
                case parser.DOUBLELIT:
                        dval = Double.parseDouble(s);
                        break;
                case parser.STRINGLIT:
                        sval = deEscape(s);
                        break;
                }
        }
}
```

Конструктор токенов выполняет те же четыре задания, то есть инициализирует поля токенов для всех токенов. Затем он использует оператор `switch` с ветвями для трех категорий токенов. Только для значений литеральных констант существует дополнительный лексический атрибут, который должен быть инициализирован. Использование встроенных в Java функций `Integer.parseInt()` и `Double.parseDouble()` для преобразования лексем является упрощением для Jzero – настоящему компилятору Java пришлось бы проделать дополнительную работу. Строка `sval` строится методом `deEscape()`, потому что ни один встроенный конвертер в Java не способен преобразовать строку исходного кода Java в фактическое строковое значение. Существуют сторонние библиотеки, которые можно найти, но для целей Jzero проще предоставить наши собственные.

Запуск сканера Jzero

Вы можете запустить программу либо в Unicon, либо в Java, как показано ниже. На этот раз давайте запустим программу на входном файле с именем `hello.java`:

```
public class hello {
        public static void main(String argv[]) {
                System.out.println("hello, jzero!");
        }
}
```

Помните, что для вашего сканера программа `hello.java` – это просто последовательность лексем. Команды для компиляции и запуска сканера Jzero похожи на те, что были в предыдущих примерах, с добавлением большего количества файлов Java:

```
uflex javalex.l                jflex javalex.l
unicon j0 javalex              javac j0.java Yylex.java
                               javac token.java parser.java
j0 hello.java                  java j0 hello.java
```

При любом варианте реализации вывод, который вы увидите, должен выглядеть следующим образом:

```
token 258: public
token 269: class
token 267: hello
token 123: {
token 258: public
token 270: static
token 265: void
token 267: main
token 40: (
token 267: String
token 267: argv
token 91: [
token 93: ]
token 41: )
token 123: {
token 267: System
token 46: .
token 267: out
token 46: .
token 267: println
token 40: (
token 277: "hello, jzero!"
token 41: )
token 59: ;
token 125: }
token 125: }
```

Сканер Jzero будет иметь гораздо больше смысла в следующей главе, в которой его вывод обеспечит входные данные синтаксического анализатора. Однако прежде, чем двигаться дальше, напомним, что регулярные выражения не могут делать все, что может понадобиться лексическому анализатору языка программирования. Иногда необходимо выйти за рамки модели сканирования lex. Следующий раздел является реальным примером этого.

Регулярных выражений не всегда достаточно

Если вы изучаете курс теории вычислений, вам, вероятно, расскажут о том, что регулярные выражения не могут соответствовать некоторым распространен-

ным шаблонам, которые встречаются в языках программирования, в частности шаблонам, в которых экземпляры одного и того же шаблона вложены друг в друга. Этот раздел показывает, что регулярных выражений не всегда достаточно и в других случаях.

Если регулярные выражения не всегда могут справиться со всеми задачами лексического анализа в вашем языке, то что делать? Лексический анализатор, написанный вручную, может справиться со странными случаями, с которыми лексический анализатор, созданный на основе регулярных выражений, не справится, возможно, ценой дополнительного дня, недели или месяца вашего времени. Однако почти во всех реальных языках программирования регулярные выражения могут подвести вас достаточно близко к тому месту, где вам понадобится лишь несколько дополнительных приемов для создания готового сканера. Вот небольшой реальный пример.

Unicon и Go являются примерами языков, обеспечивающих вставку точки с запятой. Язык определяет лексические правила, в соответствии с которыми вставляются точки с запятой, так что программистам по большей части не нужно беспокоиться о них. Вы могли заметить, что в коде Unicon примеры, как правило, содержат очень мало точек с запятой. К сожалению, эти правила вставки точек с запятой не могут быть описаны с помощью регулярного выражения.

В случае языка Go вы можете сделать это, запомнив ранее возвращенный токен и проверяя некоторые условия в семантическом действии на наличие символа новой строки. Этот символ может вернуться как точка с запятой, если условия выполнены. Но в Unicon вы должны сканировать дальше вперед и читать следующий токен после символа новой строки, чтобы решить, нужно ли ставить точку с запятой! Это позволяет вставке точки с запятой в Unicon быть более точной и создавать меньше проблем, чем в языке Go. Например, в Go вы не можете отформатировать свой код в классическом стиле языка C:

```
func main()
{
      ...
}
```

Вместо этого вы должны написать фигурную скобку в строке заголовка функции:

```
func main() {
      ...
}
```

Чтобы избежать этого смехотворного ограничения, лексический анализатор должен предусмотреть один токен просмотра вперед. Он должен будет прочитать первый токен на следующей строке, чтобы решить, нужно ли ставить точку с запятой в новой строке.

Было бы совсем не в стиле Java реализовать вставку точки с запятой в нашем сканере Jzero. Но если мы собираемся сделать это, то можем сделать это способом Go или способом Unicon. Мы покажем вам подмножество способов Go. Для справки: определение семантики вставки точки с запятой в Go можно найти на сайте *https://golang.org/ref/spec#Semicolons*.

Этот пример иллюстрирует правило № 1 из семантики вставки точки с запятой в Go. Итак, вы видите символ новой строки – вставляете ли вы точку с запятой? Давайте просто вспомним последний токен, который мы видели, и если это идентификатор, литерал, `break`, `continue`, `return`, `--`,),] или }, то сама новая строка должна вернуть новый фиктивный токен точки с запятой. Вы можете модифицировать метод `newline()` так, чтобы он возвращал булево значение *true*, если нужно вставить точку с запятой.

Это противоречит нашей стратегии использования общей спецификации lex для Unicon и Java. Нам нужно написать условие в спецификации lex, чтобы сказать, возвращать точку с запятой или нет, но синтаксис в этих двух языках различен. В Unicon наша спецификация lex будет содержать оператор `if`, который может выглядеть следующим образом:

```
\n          { if j0.newline() then return j0.semicolon() }
```

Однако в Java это потребует скобок и не позволит применить зарезервированное слово `then`:

```
\n          { if (j0.newline()) return j0.semicolon(); }
```

Версия Unicon модифицированного главного модуля `j0` с кодом вставки точки с запятой была представлена в репозитории GitHub этой книги в файле *j0go.icn*. Это *j0.icn* с новой глобальной переменной `last_token`, модификацией методов `scan()`, `newline()` и добавлением метода `semicolon()`, который конструирует искусственный токен. Вот измененные методы. Проверка того, является ли последняя категория токена одной из нескольких, которые возвращают точку с запятой, показывает генераторы Unicon. Выражение `!»)]}»` – это хитрый способ написать «)»|»]»|»}», которые будут по очереди загружаться в ord(), пока не будут испробованы все три:

```
method scan(cat)
     last_token := yylval := token(cat, yytext, yylineno)
     return cat
end
method newline()
     yylineno +:= 1
     if (\last_token).cat ===
          ( parser.IDENTIFIER|parser.INTLIT|
            parser.DOUBLELIT|parser.STRINGLIT|
            parser.BREAK|parser.RETURN|
            parser.INCREMENT|parser.DECREMENT|
            ord(!")]}") ) then return
end
method semicolon()
     yytext := ";"
     yylineno -:= 1
     return scan(parser.SEMICOLON)
end
```

Здесь есть два интересных момента. Первый заключается в том, что данный элемент исходного кода – символ новой строки, который в большинстве языков является просто пробелом, иногда возвращает целочисленный код (для вставленной точки с запятой), а иногда нет. Именно поэтому мы ввели оператор `if` в семантические действия спецификации lex для новых строк. Другой интересной вещью является искусственный токен, создаваемый методом `semicolon()`. Он производит вывод, который неотличим от того, что программист набрал бы точку с запятой в исходном коде вашего языка программирования.

> **Важное замечание**
>
> Java-реализация этого слишком длинна, чтобы представить ее здесь, поэтому она была представлена в репозитории GitHub данной книги, в файле j0go.java. В следующем абзаце представлены ее ключевые части.

Реализация Java ведет себя так же, как и версия Unicon в `j0go.icn`, с новой глобальной переменной `last_token`, модификацией методов `scan()`, `newline()` и добавлением метода `semicolon()`, который конструирует искусственный токен. Однако этот метод немного длиннее. В методе `newline()` следующего блока используется оператор Java `switch`, чтобы проверить, вызывает ли категория последнего лексемного символа вставку точки с запятой:

```
public static int scan(int cat) {
      last_token = yylval =
      new token(cat, yytext(), yylineno);
      return cat;
}
public static void newline() {
      yylineno++;
      if (last_token != null)
              switch(last_token.cat) {
              case parser.IDENTIFIER: case parser.INTLIT:
              case parser.DOUBLELIT: case parser.STRINGLIT:
              case parser.BREAK: case parser.RETURN:
              case parser.INCREMENT: case parser.DECREMENT:
              case ')': case ']': case '}':
                      return true;
              }
      return false;
}
public int semicolon() {
      yytext = ";";
      yylineno--;
      return scan(parser.SEMICOLON);
}
```

Полная семантика вставки точки с запятой в Go немного сложнее, но вставить точку с запятой, когда сканер видит регулярное выражение новой строки,

довольно просто. Если вы хотите узнать, как Unicon лучше делает вставку точки с запятой, посмотрите справочник по реализации Unicon (Unicon Implementation Compendium) на сайте *http://www.unicon.org/book/ib.pdf*.

Заключение

В этой главе вы узнали о важнейших технических навыках и инструментах, используемых в языках программирования при чтении символов исходного кода программы. Благодаря этим навыкам остальная часть компилятора или интерпретатора вашего языка программирования имеет гораздо меньшую последовательность слов/токенов, с которыми приходится иметь дело, вместо огромного количества символов, которые были в исходном файле. Если у нас все получилось, вы приобретете следующие навыки, которые сможете использовать в своем языке программирования или подобных проектах.

По мере считывания входных символов они анализируются и группируются в лексемы. Лексемы либо отбрасываются (в случае комментариев и пробельных символов), либо распределяются по категориям для последующего синтаксического анализа.

Кроме категоризации лексем, вы научились создавать из них токены. Токен – это объект, который создается для каждой лексемы при ее категоризации. Токен – это запись о данной лексеме, ее категории и о том, откуда она появилась.

Категории лексем являются основной исходной информацией для алгоритма синтаксического анализа, описанного в следующей главе. Во время синтаксического анализа лексемы будут вставляться в качестве листьев в важную структуру данных, называемую деревом синтаксиса.

Теперь вы готовы начать собирать слова во фразы в вашем исходном коде. В следующей главе будет рассмотрен синтаксический анализ, который проверяет, имеют ли фразы смысл в соответствии с грамматикой языка.

Вопросы

1. Напишите регулярное выражение для сопоставления дат в формате *dd/mym/yyy*. Можно ли написать это регулярное выражение так, чтобы оно допускало только законные даты?
2. Объясните разницу между возвращаемым значением, которое `yylex()` возвращает вызывающей стороне, лексемой, которую `yylex()` оставляет в `yytext`, и значением токена, которое `yylex()` оставляет в `yylval`.
3. Не все регулярные выражения `yylex()` возвращают целочисленную категорию. Что происходит, когда регулярное выражение не возвращает значение?
4. Лексический анализ должен иметь дело с неоднозначностью, и вполне возможно написать несколько регулярных выражений, которые могут совпасть в заданной точке ввода. Опишите правила Flex для выбора регулярного выражения для случая, когда несколько регулярных выражений могут совпасть в одном и том же месте.

Глава 4

Парсинг

В этой главе вы узнаете, как брать отдельные слова и знаки препинания, лексемы и группировать их в более крупные программные конструкции, такие как выражения, утверждения, функции, классы и пакеты. Эта задача называется **парсингом** (синтаксическим анализом). Модуль кода называется парсером. Вы создадите парсер, задав правила синтаксиса с помощью грамматики, а затем используя инструмент генератора парсера, который берет грамматику вашего языка и генерирует парсер. Мы также рассмотрим написание полезных сообщений об ошибках синтаксиса.

В этой главе рассматриваются следующие основные темы:

- синтаксический анализ;
- бесконтекстные грамматики;
- использование iyacc и BYACC/J;
- написание парсера для Jzero;
- улучшение сообщений об ошибках синтаксиса.

Мы рассмотрим технические требования к данной главе, а затем настанет время уточнить ваши представления о синтаксисе и синтаксическом анализе.

Технические требования

В этой главе вам понадобятся следующие инструменты:

- Iyacc, генератор парсера для Unicon. Вам следует использовать версию, размещенную на сайте книги (*https://github.com/PacktPublishing/Build-Your-OwnProgramming-Language*);
- BYACC/J, генератор парсера для Java (*http://byaccj.sourceforge.net*);
- вы можете скачать код этой главы из нашего репозитория GitHub: *https://github.com/PacktPublishing/Build-Your-Own-ProgrammingLanguage/tree/master/ch4*.

Видео «Код в действии» для этой главы можно найти здесь: *https://bit.ly/3ClVCSf*.

На момент написания книги бинарный дистрибутив BYACC/J для Windows состоит из файла *byaccj1.15_win32.zip*, который устарел и содержит один файл с именем *yacc.exe*. Программа установки отсутствует. Вам следует распаковать и скопировать файл *yacc.exe* в каталог с вашим путем или создать для него новый каталог и добавить его в свой путь. Проверьте, что эти пакеты были добавлены в ваш путь, открыв новую командную строку или окно терминала, и попробуйте выполнить команды *iyacc* и *yacc*. Обратите внимание, что на вашем

компьютере уже может быть другая программа с именем *yacc*! В этом случае мы рекомендуем переименовать экземпляр *BYACC/J*, который вы устанавливаете для этой книги, в *byaccj* или *byaccj.exe*, вместо *yacc* или *yacc.exe*. Если вы сделаете это, то везде в этой книге, где говорится об использовании *yacc*, вы должны указать *byaccj*. Для успешного использования данной книги вам необходимо держать свои *yacc* в порядке! Вы были предупреждены!

Для этой главы существует одно дополнительное техническое требование. Вы должны установить переменную среды `CLASSPATH`. Если вы работаете с примерами этой главы в каталоге *C:\users\Alfrede Newmann\ch4*, вам может понадобиться установить `CLASSPATH` так, чтобы она указывала на каталог `Alfrede Newmann` над каталогом `ch4`. В Windows лучше всего установить это раз и навсегда в панели управления или настройках, но вы можете установить это вручную, если вам нужно, с помощью команды, подобной этой:

```
set CLASSPATH=".;c:\users\Alfrede Newmann"
```

В Linux добавление в CLASSPATH каталога выше текущего может быть достигнуто с помощью `..`, в то время как в Windows вы должны указать полный путь к родительскому каталогу. В Linux это лучше всего задать в `~/.bashrc` или подобном файле, а в командной строке это выглядит следующим образом:

```
export CLASSPATH=.:..
```

Прежде чем перейти к деталям `yacc`, давайте рассмотрим общую картину того, чего мы пытаемся достичь с помощью парсинга, которая заключается в анализе синтаксиса входной программы.

Анализ синтаксиса

Как программист вы, вероятно, уже знакомы с **сообщениями об ошибках синтаксиса** и общей идеей синтаксиса, которая заключается в понимании того, какие слова или лексемы и в каком порядке должны появляться, чтобы данное сообщение было правильно сформировано. Большинство человеческих языков придирчивы в этом отношении, хотя некоторые из них более гибки в отношении порядка слов. К счастью, большинство языков программирования намного проще и более ограничены, чем естественные человеческие языки, в отношении того, что является законным вводом.

Входные данные для синтаксического анализа состоят из результатов **лексического анализа**, о котором говорится в предыдущей главе. Информация, такая как сообщение или программа, разбивается на последовательность составляющих слов и знаков препинания. Это может быть массив или список токенов, хотя для парсинга алгоритму требуется только последовательность целочисленных кодов, возвращаемых после обращения к `yylex()`, один за другим. Работа синтаксического анализа заключается в том, чтобы определить, является сообщение на данном языке, таком как английский или Java, корректным или нет. Результатом синтаксического анализа является простое булево значение `true` или `false`. На практике, для того чтобы интерпретировать или перевести сообщение, требуется нечто большее, чем булево значение, которое

говорит нам о правильности синтаксиса. В следующей главе вы узнаете, как построить дерево синтаксиса, которое является основой для последующего перевода программы в код. Но сначала мы должны проверить синтаксис, поэтому давайте рассмотрим, как задается синтаксис языка программирования, что называется **бесконтекстной грамматической** нотацией.

Понимание бесконтекстных грамматик

В этом разделе мы определим обозначения, используемые изобретателями языков программирования для описания синтаксиса своего языка. Вы сможете использовать то, что узнаете в этом разделе, для ввода правил синтаксиса на вход генераторов парсера, используемых в следующем разделе. Давайте начнем с понимания того, что такое бесконтекстные грамматики.

Бесконтекстные грамматики – это наиболее широко используемая нотация для описания синтаксиса, допустимого в языке программирования, в терминах шаблонов лексем. Они сформулированы из очень простых правил, которые легко понять. Бесконтекстные грамматики строятся из следующих компонентов:

- **терминальные символы**. Набор входных символов называется терминальными символами. Терминальные символы в грамматике считываются со сканера, подобного тому, который мы изготовили в предыдущей главе. Хотя они называются символами, терминальные символы соответствуют целому слову, оператору или знаку препинания, терминальный символ определяет категорию лексемы. Как вы видели в предыдущей главе, категории этих символов представлены целочисленными кодами, которым обычно даются мнемонические имена, например `IDENTIFIER`, `INTCONST` или `WHILE`. В наших грамматиках мы также будем использовать символьную литеральную нотацию для более тривиальных терминальных символов. Один символ внутри апострофов – это просто терминальный символ, состоящий из самого символа. Например, `';'` – это терминальный символ, состоящий из одной только точки с запятой и буквально обозначающийся целым числом 59, которое является кодом ASCII для точки с запятой;
- **нетерминальные символы**. В отличие от регулярных выражений, правила бесконтекстной грамматики используют второй набор символов, называемых нетерминальными символами. Нетерминальные символы относятся к последовательностям символов, которые имеют смысл вместе, например фразы-существительные или предложения (в естественных языках), или определения функций или классов, или целые программы (в языках программирования). Один специальный нетерминальный символ обозначается как начальный символ всей грамматики. В грамматике языка программирования начальный символ обозначает весь хорошо сформированный исходный файл;
- **правила производства**. Набор правил, называемых правилами производства, объясняет, как формировать нетерминальные символы из более мелких слов и составных фраз. Поскольку правила производства управляют тем, какие терминальные и нетерминальные символы ис-

пользуются, обычно принято давать грамматику, приводя только правила производства.

Теперь пришло время более подробно рассмотреть правила построения бесконтекстных грамматик, а также примеры.

Написание правил бесконтекстной грамматики

Правила производства, также называемые правилами бесконтекстной грамматики, представляют собой шаблоны, которые описывают законные последовательности лексем с использованием терминальных символов и дополнительных нетерминальных символов, представляющих другие последовательности из нуля или более символов. В этой книге мы будем использовать нотацию *yacc* для записи контекстно-свободных грамматик. Каждое правило производства состоит из одного нетерминального символа, за которым следует двоеточие, затем ноль или более терминальных и нетерминальных символов, заканчивающихся точкой с запятой, как показано в следующей записи:

```
X : symbols ;
```

Слева от двоеточия находится только один символ, который, по определению, является нетерминальным, потому что смысл правила грамматики таков: нетерминал X может быть построен из последовательности терминалов и нетерминалов, которые появляются в правой части.

Бесконтекстная грамматика может иметь столько таких правил, сколько необходимо, включая несколько правил, которые строят один и тот же нетерминал с помощью различных комбинаций символов в правой части. На самом деле множественные правила для одного и того же нетерминала настолько распространены, что у них есть свое собственное сокращение, состоящее из вертикальной полосы. Пример вертикальной полосы можно увидеть в следующем коде:

```
X : symbols | other_symbols ;
```

Когда вертикальная полоса (читается как *или*) используется в грамматике, это говорит о том, что существуют различные способы построения нетерминала X. Использование вертикальной полосы необязательно, потому что вы могли бы написать те же правила как отдельные выражения из нетерминала, двоеточия, правой части и точки с запятой. Например, вот утверждение о различных способах построения X:

```
X : A | B | C ;
```

Эта строка эквивалентна следующим трем строкам:

```
X : A ;
X : B ;
X : C ;
```

Оба кода описывают одни и те же три правила производства. Вертикальная полоса – это просто сокращенное обозначение для записи нескольких правил производства.

Итак, что же означает правило производства? Его можно читать и использовать как вперед, так и назад. Если вы начинаете с начального символа и заменяете нетерминал одной из правых сторон правила производства (это называется **шагом вывода**), то вы проделываете путь вниз от вершины. Если вы повторяете этот процесс и в конце концов добираетесь до последовательности терминальных символов, в которой не осталось нетерминалов, то вы создали законный экземпляр этой грамматики.

С другой стороны, язык программирования начинает с другого конца. Сканер из предыдущей главы выдает последовательность терминальных символов. Имея последовательность терминальных символов, можете ли вы найти в ней правую часть правила производства и заменить ее на нетерминал? Если вы сможете проделать это несколько раз и вернуться к начальному символу, вы доказали, что исходная входная программа является законной в соответствии с грамматикой. Это называется **парсингом**.

Теперь пришло время рассмотреть несколько простых грамматических примеров. Некоторые из наиболее интуитивно понятных грамматик, которые мы можем предложить, происходят из естественных (человеческих) языков. Другие простые примеры показывают, как бесконтекстные грамматики применяются к синтаксису языка программирования.

> **Рекурсия**
>
> Знаете ли вы, что такое рекурсия? В математике и компьютерных науках рекурсия – это когда что-то определяется в терминах более простой версии самого себя (см. *https://en.wikipedia.org/wiki/Recursion*, если вам нужно освежить в памяти). Вам понадобится эта концепция для построения синтаксиса вашего языка программирования. В бесконтекстных грамматиках нетерминал X часто используется в правой части правила производства, которое строит X. Это форма рекурсии. Единственное логическое правило, которое вы должны усвоить при использовании рекурсии, заключается в следующем: должно существовать другое правило грамматики (**базовый случай**), которое не является рекурсивным. В противном случае рекурсия никогда не закончится, и ваша грамматика не будет иметь смысла.

Написание правил для программных конструкций

Бесконтекстные грамматики будут простыми, после того как вы напишете несколько таких грамматик. Вы должны начать с самых простых правил, которые только сможете придумать, и постепенно продвигаться вперед. Простейшие значения в языке – это литеральные константы. Предположим, что в нашем языке есть два вида значений – булевы и целые числа:

```
literal : INTLIT | BOOLLIT ;
```

Приведенное выше правило производства гласит, что существует два вида литеральных значений – целые и булевы числа. Некоторые языковые конструкции, такие как сложение, могут быть определены только для определенных типов, в то время как другие конструкции, такие как присваивание, определены для всех типов. Часто лучше всего использовать общий синтаксис для всех типов, а затем убедиться, что типы являются правильными, позже, во время семантического анализа. Мы рассмотрим это в *главе 7 «Проверка базовых типов»* и *главе 8 «Проверка типов в массивах, обращения к методам и доступ к структурам»*. Сейчас разберем правило грамматики, которое допускает либо переменные, либо литеральные константы:

```
simple_expr : IDENTIFIER | literal ;
```

Как вы видели в *главе 3 «Сканирование исходного кода»*, IDENTIFIER обозначает имя. Приведенное выше правило производства гласит, что в простых выражениях допускаются как переменные, так и литералы. Сложные выражения строятся путем применения операторов или функций к простым выражениям:

```
expr : expr '+' expr | expr '-' expr | simple_expr ;
```

Предыдущие три правила производства представляют собой общий вопрос проектирования. Первые два правила являются рекурсивными, многократно повторяющимися. Они также неоднозначны. Когда несколько операторов соединяются в цепочку, грамматика не указывает, какой оператор применяется первым.

> **Неоднозначность**
>
> Когда грамматика может воспринимать одну и ту же строку двумя или более различными способами, грамматика является неоднозначной. В предыдущем примере 1 + 2 – 3 может быть разобрано с применением правила производства сначала знака плюс, а затем вычитания или наоборот. Неоднозначность иногда может заставить вас переписать вашу грамматику так, чтобы был только один способ разбора входных данных.

В реальных языках существует гораздо больше операторов и есть вопрос приоритета операторов. Вы можете рассмотреть эти темы в разделе данной главы «*Написание парсера для Jzero*». Пока же давайте кратко рассмотрим более крупные структуры языка, такие как инструкции (statements). Здесь приводится простое представление инструкции присваивания:

```
statement : IDENTIFIER '=' expr ';' ;
```

Эта версия присваивания позволяет использовать только имя слева от знака равенства. Правая часть может принимать любое выражение. Существует еще несколько основных видов инструкций, встречающихся во многих языках. Из них рассмотрим две наиболее распространенные – `IF` и `WHILE`:

```
statement : IF '(' expr ')' statement ;
statement : WHILE '(' expr ')' statement ;
```

Эти инструкции содержат другие (под)инструкции. Грамматики строят более крупные конструкции из более мелких, используя рекурсивные правила, такие как это. Инструкции `IF` и `WHILE` имеют почти идентичный синтаксис для предварения инструкции условным выражением:

```
statements : statements statement | statement ;
```

Множественные инструкции могут быть получены путем повторного применения первого правила этой грамматики. Хорошие разработчики языка постоянно пишут рекурсивные правила, для того чтобы повторить конструкцию. В случае таких языков, как Java, точки с запятой не появляются в этом правиле грамматики как разделитель инструкций, но они появляются для завершения различных правил грамматики, например в предыдущем правиле для инструкций присваивания.

В этом разделе вы увидели, что грамматические правила языка программирования используют зарезервированные слова и знаки препинания в качестве строительных блоков. Более крупные выражения и утверждения составляются из более мелких с помощью рекурсии. Теперь пришло время изучить инструменты, которые используют бесконтекстную грамматическую нотацию для построения парсеров для чтения исходного кода, а именно **iyacc** и **BYACC/J**.

Использование iyacc и BYACC/J

Название yacc расшифровывается как **yet-another-compiler-compiler**. Эта категория инструментов принимает бесконтекстную грамматику в качестве входных данных и генерирует из нее парсер. Yacc-совместимые инструменты доступны для большинства популярных языков программирования.

В этой книге для Unicon мы используем **iyacc** (сокращение от **Icon yacc**), а для Java вы можете использовать **BYACC/J** (сокращение от **Berkeley YACC extended for Java**). Они хорошо совместимы с UNIX *yacc*, и мы можем представить их вместе, как один язык для написания парсеров. В остальной части этой главы мы будем просто говорить *yacc*, когда будем иметь в виду и *iyacc*, и *BYACC/J* (который вызывается как *yacc*, по крайней мере в Windows). Полная совместимость потребовала немного Кобаяши Мару (Kobayashi Maru), в основном когда дело доходит до семантических действий, которые написаны в Unicon и Java соответственно.

Файлы *yacc* часто называют (*yacc*-) спецификациями. Они имеют расширение *.y* и состоят из нескольких разделов, разделенных %%. В этой книге под спецификациями *yacc* понимается в общем случае входной файл, предоставляемый *iyacc* или *BYACC/J*, и в большинстве случаев эти файлы будут также действительны для C *yacc*.

В спецификации *yacc* есть обязательные разделы – **раздел заголовка**, за которым следует **раздел бесконтекстной грамматики**, и необязательный

раздел вспомогательных функций. Раздел заголовка *yacc* и раздел бесконтекстной грамматики – это разделы, которые вам необходимо знать для данной книги. В следующем разделе вы узнаете, как объявить свои терминальные символы в разделе заголовка *yacc*. Некоторые версии *yacc* требуют таких объявлений.

> **Кобаяши Мару?**
>
> Сценарий «Кобаяши Мару» – это безвыигрышная ситуация, в которой лучшим решением будет изменить правила игры. В данном случае я немного изменил *iyacc* и *BYACC/J*, чтобы наша безвыигрышная ситуация стала выигрышной.

Объявление символов в разделе заголовка

Большинство инструментов *yacc* имеют опции, которые можно включить в разделе заголовка. Они различны, и мы будем рассматривать их только в том случае, если мы их используем. Вы также можете включать туда фрагменты кода хост-языка, например объявления переменных, внутри блоков %{ ... %}. Основной целью раздела заголовка является объявление терминальных и нетерминальных символов в грамматике. В разделе бесконтекстной грамматики эти символы используются в правилах производства.

О том, является символ терминальным или нетерминальным, можно судить по тому, как этот символ используется в грамматике, но если это не ASCII-коды, вы должны в любом случае объявить все терминальные символы. Терминальные символы объявляются в разделе заголовка с помощью строки, начинающейся с токена %, за которой следует столько имен терминальных символов, сколько вы хотите, разделенных пробелами. Нетерминальные символы могут быть объявлены аналогичной строкой % нетерминалов. Кроме всего прочего, *yacc* использует объявления символов терминалов для создания файла, в котором этим именам присваиваются целочисленные константы, для использования в вашем сканере.

> **Дополнительные объявления *yacc***
>
> Существуют и другие объявления, которые могут быть помещены в разделе заголовка *yacc*, помимо тех, которые используются в этой книге. Если вы не хотите помещать начальный нетерминал в начало грамматики, то можете поместить его куда угодно, а затем определить его в явном виде в заголовке через объявление *%start* для некоторого нетерминального символа. Кроме того, вместо того чтобы просто объявлять токены с помощью *%token*, вы можете использовать *%left*, *%right* и *%nonassoc* для указания приоритета оператора и ассоциативности в порядке возрастания.

Теперь, когда мы узнали о разделе заголовка, давайте посмотрим на раздел бесконтекстной грамматики.

Составление раздела бесконтекстной грамматики *yacc*

Основной раздел спецификации *yacc* – это раздел бесконтекстной грамматики. Заданным считается каждое правило производства бесконтекстной грамматики, за которым следует необязательное семантическое действие, состоящее из кода на языке хоста (в нашем случае Unicon или Java), которое должно быть выполнено, когда это правило производства было найдено. Синтаксис *yacc* обычно выглядит следующим образом:

```
X : symbols { semantic action code } ;
```

Также законно размещать семантические действия перед или между символами в дополнение к концу правила, но если вы это делаете, то на самом деле вы объявляете новый нетерминал с пустым правилом производства, который содержит только это семантическое действие. Мы не будем делать этого в данной книге, так как это – частый источник ошибок.

Yacc менее придирчив к пробельным символам, чем lex. В следующем примере показаны три эквивалентных способа форматирования правил производства с различными пробелами. Какой из них предпочесть, зависит от того, что лучше для читабельности:

```
A : B | C;
A : B |
    C ;
A : B
  | C
  ;
```

Хотя каждое правило производства начинается с новой строки, оно может охватывать несколько строк и завершается одним из следующих символов: точкой с запятой, вертикальной полосой, указывающей на другое правило производства для того же нетерминала, символом %%, указывающим на начало раздела вспомогательных функций или конец файла. Как и в lex, если семантическое действие использует фигурные скобки, чтобы закончить блок оператора обычным способом, семантическое действие может охватывать несколько строк исходного кода. Yacc не начнет искать следующее правило производства, пока не найдет соответствующую закрывающую фигурную скобку для завершения семантического действия, а затем переходит к поиску одного из признаков окончания, перечисленных ранее, таких как точка с запятой или вертикальная полоса, которые завершают правило производства.

Распространенная ошибка, которую допускают новички в разделе бесконтекстной грамматики, – это попытка вставить комментарии в правила производства, чтобы улучшить читабельность. Не делайте этого, иначе вы можете получить очень загадочные сообщения об ошибках.

Когда вы запускаете классический UNIX *yacc*, который является инструментом языка C, он генерирует функцию под названием `yyparse()`, которая проверяет, была ли входная последовательность терминальных символов, возвращенная из `yylex()`, законной в соответствии с грамматикой. Глобальные переменные могут быть заданы с другими полезными битами информации.

Вы можете использовать такие глобальные переменные для хранения чего угодно, например корня дерева синтаксиса. Прежде чем перейти к более крупным примерам, рассмотрим, как работают парсеры *yacc*. Вам необходимо знать это, чтобы отладить свой парсер, когда что-либо идет не по плану.

Понимание парсеров *yacc*

Алгоритм парсера, создаваемого *yacc*, называется LALR(1). Он происходит из семейства алгоритмов парсинга, придуманных Дональдом Кнутом (Donald Knuth) из Стэнфорда и получивших практическое применение благодаря Фрэнку ДеРемеру (Frank DeRemer) из Калифорнийского университета в Санта-Крус и другим. Если вы интересуетесь теорией, загляните на страницу Википедии, посвященную парсеру LALR, на сайте *https://en.wikipedia.org/wiki/LALR_parser*, или обратитесь к серьезной книге по построению компиляторов, такой как «Введение в компиляторы и проектирование языков» Дугласа Тейна (Douglas Thain) с сайта *https://www3.nd.edu/~dthain/compilerbook/*.

Для наших целей вам нужно знать, что сгенерированный алгоритм состоит из длинного цикла *while*. На каждой итерации цикла парсер делает один маленький шаг вперед. Алгоритм использует стек целых чисел для отслеживания того, что он видел. Каждый элемент в стеке синтаксического анализа является целочисленным кодом, называемым **состоянием разбора**, который кодирует все терминальные и нетерминальные символы, встречавшиеся до этого момента. Состояние разбора на вершине стека и **текущий входной** символ, который является целочисленным терминальным символом, полученным из функции `yylex()`, являются двумя частями информации, используемой для принятия решения о том, что делать на каждом шаге. Без какой-либо внутренней причины принято представлять это как горизонтальный кусок нити, на котором бусины справа сдвигаются влево на стек, который также изображен горизонтально. На рис. 4.1 показан стек разбора *yacc* слева и его вход справа.

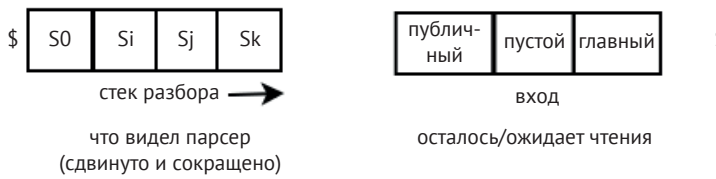


Рис. 4.1. Стек разбора yacc и его вход

Знак доллара слева обозначает дно стека, а знак доллара справа – конец файла. *Yacc* генерирует две большие таблицы, вычисляемые из грамматики, называемые **таблицей действий** и **таблицей перехода**. В этих таблицах кодируется то, что нужно делать на каждом шаге. Основная таблица – это таблица действий, которая просматривает состояние разбора, текущий вход и возвращает одну из следующих возможностей:

- несколько верхних элементов в стеке содержат правило производства, которое может быть использовано для возврата (в конечном итоге) обратно к начальному нетерминалу. Это называется **сокращением**;

- алгоритму нужно посмотреть на следующий входной символ. Поместите текущий входной символ в стек разбора и прочитайте следующий, используя `yylex()`. Это называется **сдвигом**;
- если ни сдвиг, ни сокращение не работают, то сообщается о синтаксической ошибке обращением к функции `yyerror()`, которую вы должны написать;
- если мы смотрим на начальный нетерминал и больше не ожидаем ввода, вы выиграли! `yyparse()` возвращает код, который говорит, что ошибок не было.

В следующем списке вы можете увидеть алгоритм парсинга *Yacc* в виде псевдокода. В этом коде есть несколько ключевых переменных и операций, описанных ниже:

- *parsestk* – стек разбора, массив целочисленных состояний разбора конечного автомата;
- *index top* – отслеживает подстрочный индекс вершины стека разбора;
- *current* – текущий входной символ;
- *shift_n* – означает перемещение входных данных справа налево, сдвигая состояние разбора *n* в стек и перемещая *current* на следующий входной символ;
- *reduce_m* – означает применение правила производства *m*, вытесняя количество состояний разбора, равное правой части правила производства *m*, и выталкивая новое состояние разбора, соответствующее нетерминалу в левой части правила производства *m*. Таблица перехода указывает, какое новое состояние разбора должно быть вытолкнуто сокращением.

Вот алгоритм парсинга в форме псевдокода:

```
repeat:
     x = action_table[parsestk[top], current]
     if x == shift_n then {
             push(state_n, parsestk)
             current = next
             }
     else if x == reduce_m then {
             pop(parsestk) |m| times
             push(goto_table[parsestk[top],m], parsestk)
             }
     else if x == accept then return 0 // no errors
     else { yyerror("syntax error") }
```

Этот псевдокод является прямым воплощением предыдущего маркированного списка. Большое число мировых языков программирования выполняют синтаксический анализ с помощью этого метода. Возможно, вам будет интересно сравнить этот псевдокод со сгенерированным .icn- или .java-файлом, созданным iyacc или BYACC/J.

Важно отметить, что поскольку этот алгоритм синтаксического анализа представляет собой просто цикл с парой поисков в таблице, он работает довольно быстро.

Смысл инструмента *yacc* в том, чтобы просто предоставить бесконтекстную грамматику и получить парсер, не заботясь о том, как он работает. Таким образом, *yacc* является **декларативным** языком. Алгоритм работает, и вам не нужно много знать о нем, но если вы измените грамматику или изобретете новый язык с помощью инструмента *yacc*, вам, возможно, придется узнать об этих сдвигах и операциях сокращения, чтобы отладить вашу бесконтекстную грамматику, если ваш парсер не делает то, что вы хотите. Чаще всего программисты, использующие *yacc*, сталкиваются с этим следующим образом: когда вы запускаете *yacc*, он сообщает о конфликтах, которые вам, возможно, нужно устранить.

Устранение конфликтов в парсерах *yacc*

Ранее в этой главе, в разделе, озаглавленном «Написание правил для программных конструкций», вы узнали, что грамматики могут быть неоднозначными. Когда грамматика неоднозначна, *yacc* будет иметь более одного возможного действия, которое он может закодировать для данного (состояние разбора, текущий вход) поиска в таблице действий. *Yacc* сообщает об этом как о проблеме, и в этом случае созданный парсер будет использовать только одну из возможных интерпретаций неоднозначности. Существует два вида конфликтов, о которых сообщает *yacc*:

- **shift/reduce** – когда одно правило производства говорит, что в данный момент можно сдвинуть текущий вход, но другое правило производства говорит, что все готово и можно сокращать. В этом случае *yacc* будет только сдвигать, и у вас будут проблемы, если вам нужно было сокращать;
- еще хуже обстоит дело с **reduce/reduce**. Различные правила производства говорят, что они хотят что-то сократить в этот момент. Какое из них правильное? *Yacc* произвольно выберет то из них, которое встречается раньше в вашем .y-файле, что правильно в 50 % случаев.

Для конфликтов shift/reduce правило по умолчанию обычно является корректным. Я видел рабочие языковые грамматики с буквально сотнями конфликтов shift/reduce, которые игнорируются и не имеют, казалось бы, каких-либо побочных эффектов – они бессимптомны. Но однажды я увидел в реальной жизни, что умолчание о конфликте shift/reduce – это не то, что нужно языку.

Для конфликтов reduce/reduce мы не можем использовать правило по умолчанию. Часть вашей грамматики не будет работать. В любой ситуации reduce/reduce или если вы определите, что конфликт shift/reduce является проблемой, вам нужно будет изменить грамматику, чтобы устранить конфликт. Модификация грамматики для предотвращения конфликтов выходит за рамки данной книги, но обычно это включает в себя доработку для устранения избыточных частей грамматики или создание новых нетерминалов и более строгих правил производства. Теперь рассмотрим, что происходит, когда парсер сталкивается с ошибкой.

Исправление синтаксических ошибок

Исправление синтаксических ошибок – это когда ваш парсер продолжает работу после сообщения о синтаксической ошибке. Если исправление прошло успешно, компилятор может продолжить поиск остальных ошибок, если таковые имеются. Во времена пакетной обработки было важно исправить как можно лучше. Однако исправление ошибок известно своими впечатляющими неудачами! Компиляторы склонны выдавать многочисленные каскадные сообщения об ошибках после первого, потому что попытка восстановления и продолжения парсинга основана на диких догадках о том, были ли пропущены токены, или присутствовали лишние токены, или был непреднамеренно использован неправильный токен… существует слишком много возможностей. По этой причине в данной книге мы будем придерживаться минимального исправления ошибок.

Парсер *yacc* будет пытаться восстановиться, если в грамматику добавлены дополнительные правила производства, которые показывают вероятные места синтаксических ошибок, используя специальный токен с именем *error*. Когда возникает фактическая синтаксическая ошибка, парсер shift/reduce отбрасывает состояния разбора из своего стека разбора и токены из входных данных до тех пор, пока не найдет состояние, в котором есть правило, позволяющее продолжить работу над ошибкой. В языке Jzero мы можем иметь правило, отбрасывающее синтаксическую ошибку внутри утверждений, которое отбрасывает токены, пока не увидит точку с запятой. В грамматике может быть одно или два места более высокого уровня, где токен *error* переходит в конец тела функции или объявления, и все.

Хотя мы только затрагиваем эту тему, если ваш язык программирования становится известным и популярным, вы, вероятно, должны в конце концов научиться избавляться хотя бы от простейших и наиболее распространенных ошибок. Поскольку ошибки неизбежны, помимо восстановления и продолжения парсинга, вам нужно подумать о том, как сообщать об ошибках. Сообщения об ошибках рассматриваются в разделе «*Улучшение сообщений об ошибках синтаксиса*» в конце этой главы. Теперь давайте соберем несколько рабочих парсеров, используя сканеры, разработанные в предыдущей главе.

Создание игрушечного примера

Этот пример позволяет проверить, можете ли вы установить и запустить iyacc и BYACC/J. Парсер примера просто анализирует последовательности чередующихся имен и чисел. Имя файла ns.y (для последовательности имен) будет использоваться для спецификации *yacc*. Код, сгенерированный *yacc* из этой спецификации, будет использовать две вспомогательные функции, которые на Java могут создать начало нашего раздела заголовка *yacc*, который выглядит следующим образом:

```
%{
import static ch4.lexer.yylex;
import static ch4.yyerror.yyerror;
%}
```

Если вы создаете реализацию на Java, используя стоковую версию BYACC/J 1.15, *ns.y* должен начинаться с этих четырех строк. Код внутри `%{ ... %}` состоит из двух деклараций: `import static`. Таким образом, Java позволяет сгенерированному коду парсера вызывать функции `yylex()` и `yyerror()`, которые находятся в разных классах, в разных исходных файлах.

Мы хотим поместить `yylex()` и `yyerror()` в разные классы и исходные файлы вместо раздела вспомогательных функций файла *.y*, потому что они будут разными в Unicon и Java. Другая причина этого заключается в том, что `yylex()` и `yyerror()` могут генерироваться разными инструментами – `uflex` и `jflex` из предыдущей главы и `merr`, описанным далее в этой главе. К сожалению, Java не может выполнить `import static`, не поместив эти классы и функции в пакет. Пакет имеет имя `ch4`, потому что код данной главы находится в каталоге с именем ch4, а Java требует, чтобы имена пакетов и каталогов совпадали. В результате использования пакетов код из главы 3 «*Сканирование исходного кода*» должен быть немного изменен, и вы можете ожидать сложных проблем с *CLASSPATH* и загадочных сообщений об ошибках.

Поскольку строки `import static` не работают для Unicon, для этой книги я модифицировал BYACC/J, чтобы добавить опции командной строки для статических импортов, которые необходимы. Если вы используете версию этих инструментов с сайта книги или другого нового/актуального источника, то можете пропустить приведенные выше четыре строки кода и выполнить их из командной строки, что позволит использовать файл *ns.y* без изменений в качестве входных данных как для проектов Unicon, так и для проектов Java.

В нижеприведенном примере *ns.y* нет кода семантических действий, эта глава посвящена исключительно анализу синтаксиса. Семантические действия рассматриваются более подробно в следующей главе.

```
%token NAME NUMBER
%%
sequence : pair sequence | ;
pair : NAME NUMBER ;
```

На основе этой спецификации *yacc* создаст функцию *yyparse()*. Она выполняет алгоритм парсинга LALR с сетевым эффектом, описываемый следующим образом:

1) *yyparse()* вызывается из функции *main()*;
2) *yyparse()* вызывает *yylex()* для получения терминального символа;
3) *yyparse()* сопоставляет каждый терминальный символ, возвращенный из *yylex()*, со всеми возможными синтаксическими разборами, используя все возможные комбинации правил производства;
4) парсинг в конечном итоге выбирает то правило производства, которое является корректным в текущем местоположении, и выполняет его семантическое действие (если оно имеется).

Шаги 2–4 повторяются до тех пор, пока не будет разобран весь ввод или не будет обнаружена синтаксическая ошибка. Функция `yylex()` генерируется на основе следующей спецификации *lex*:

```
package ch4;
%%
%int
%%
[a-zA-Z]+     { return Parser.NAME; }
[0-9]+        { return Parser.NUMBER; }
[ \t]+        { }
.             { lexer.lexErr("unrecognized character"); }
```

Это файл `nnws.l` из предыдущей главы, измененный для использования с этим генерируемым *yacc*-парсером. Во-первых, в Java он должен стать частью пакета `ch4`. Кроме того, он должен возвращать целые числа, которые *yacc* использует для `NAME` и `NUMBER`. Как вы, возможно, помните из предыдущей главы, совместимый с Java способ доступа к этим целым числам по имени – через объект `Parser`, который их содержит. Инструмент BYACC/J генерирует этот объект парсера автоматически для Java. Для Unicon традиционная опция *-d* в *iyacc* генерирует макроопределения во включаемом файле (для *ns.y* это будет файл *ns_tab.icn*), как в классическом UNIX C *yacc*. Для этой книги *iyacc* был дополнен опцией командной строки *-dd*, которая вместо этого генерирует Java-совместимый объект `Parser`, который содержит имена и их значения.

Функция `main()` обязательно зависит от языка. К тому времени, когда вы добавляете в программу модуль *yacc* `yyparse()`, все начинает усложняться. Поэтому в предыдущей главе функционал `main()` была настроен таким образом, чтобы вынести инициализацию лексического анализатора и обработку лексических ошибок в отдельные файлы. Сначала обсудим функцию `main()`. После инициализации `main()` обращается к `yyparse()` для проверки синтаксиса исходного кода. Вот версия главного модуля на языке Unicon в файле *trivial.icn*:

```
procedure main(argv)
      yyin := open(argv[1]) | stop("usage: trivial file")
      lexer := lexer()
      Parser := Parser()
      if yyparse() = 0 then write("no errors")
end
procedure yyerror(s)
        stop(s)
end
class lexer()
      method lexErr(s)
              stop("lexical error: ", s)
      end
end
```

Эта реализация `main()` в Unicon открывает входной файл по имени, указанному в первом аргументе командной строки. Лексический анализатор получает информацию о том, какой файл читать, путем присвоения значения переменной *yyin*. Инициализируются объекты лексического анализатора и парсера, они здесь только для совместимости с Java нашей спецификации flex. Затем код обращается к `yyparse()` для разбора входного файла. Следующий Java-код

в файле *trivial.java* содержит функцию `main()`, которая соответствует предыдущему примеру Unicon:

```
package ch4;
public class trivial {
      static ch4.j0p par;
      public static void main(String argv[]) throws Exception
      {
              ch4.lexer.init(argv[0]);
              par = new ch4.Parser();
              int i = par.yyparse();
              if (i == 0)
                      System.out.println("no errors");
      }
}
```

Этот основной модуль короче, чем простой класс в предыдущей главе. Все, что он делает, – это инициализирует лексический анализ, инициализирует парсер, а затем обращается к `yyparse()`, чтобы проверить, являются ли входные данные легальными. Чтобы вызвать функцию `yylex()` из `yyparse()` без ссылки на объект Yylex и без циклической ссылки обратно на основной класс `trivial`, объект Yylex и его инициализация были перенесены в класс-оболочку с именем `lexer`. Следующий файл *lexer.java* содержит этот код:

```
package ch4;
import java.io.FileReader;
public class lexer {
      public static Yylex yylexer;
      public static void init(String s) throws Exception {
              yylexer = new Yylex(new FileReader(s));
      }
      public static int YYEOF() { return Yylex.YYEOF; }
      public static int yylex() {
              int rv = 0;
              try {
                      rv = yylexer.yylex();
              } catch(java.io.IOException ioException) {
                      rv = -1;
              }
              return rv;
      }
      public static String yytext() {
              return yylexer.yytext();
      }
      public static void lexErr(String s) {
              System.err.println(s);
              System.exit(1);
      }
}
```

Метод `init()` инициализирует объект `Yylex` для последующего использования статическим методом `yylex()`, который вызывается из `yyparse()`. Здесь `yylex()` – это просто прокси, который вызывает `yylexer.yylex()`.

Есть еще одна часть пазла – `yyparse()` вызывает функцию `yyerror()`, когда он встречает синтаксическую ошибку. Файл *yyerror.java* содержит класс `yyerror`, который имеет статический метод `yyerror()`, показанный ниже:

```
package ch4;
public class yyerror {
     public static void yyerror(String s) {
             System.err.println(s);
             System.exit(1);
     }
}
```

Эта версия функции `yyerror()` просто вызывает `println()` и завершается, но мы можем изменить ее по мере необходимости. Хотя вы, возможно, захотите сделать это ради совместного использования файла спецификации *yacc* Unicon'ом и Java, это также окупится, когда мы улучшим наши сообщения об ошибках синтаксиса в следующем разделе.

Теперь пришло время запустить нашу игрушечную программу и посмотреть, что она делает. Запустите ее со входным файлом *dorrie3.in*:

```
Dorrie 1 Clint 0
```

Вы можете построить и запустить программу в Unicon или Java следующим образом. Последовательность команд для выполнения в Unicon выглядит так:

```
uflex nnws.l
iyacc -dd ns.y
unicon trivial nnws ns ns_tab
trivial dorrie3.in
```

Последовательность команд для выполнения в Java:

```
jflex nnws.l
yacc -Jpackage=ch4 -Jyylex=ch4.lexer.yylex \
                -Jyyerror=ch4.yyerror.yyerror ns.y
javac trivial.java Yylex.java Parser.java lexer.java \
        yyerror.java ParserVal.java
java ch4.trivial dorrie3.in
```

При любом варианте реализации вы должны увидеть следующий результат:

```
no errors
```

Пока что все, что делает пример, – это классифицирует группы входных символов, используя регулярное выражение для определения типа найденной лексемы. Для работы остальной части компилятора нам понадобится дополнительная информация о лексеме, которую мы будем хранить в токене.

В этом разделе вы узнали, как объединить парсер, созданный на основе yacc, со сканером из предыдущей главы, созданным на основе lex. Одни и те же спецификации lex и yacc были использованы для Unicon и Java, после некоторых незначительных изменений в iyacc и BYACC/J. Основными проблемами были проблемы с внедрением этих декларативных языков в Java, что включало в себя написание и импорт двух статических методов из вспомогательных классов. К счастью, мы смогли заставить эти инструменты работать на игрушечном примере. Теперь пришло время использовать их в настоящем языке программирования.

Написание парсера для Jzero

Следующий пример – это парсер для Jzero, нашего подмножества языка Java. Он расширяет пример Jzero из предыдущей главы. Большим изменением является введение множества правил бесконтекстной грамматики для более сложных синтаксических конструкций, чем те, которые встречались до сих пор. Если бы вы писали новый язык, не основанный на существующем, вам пришлось бы придумывать бесконтекстную грамматику с нуля. Для Jzero это не так. Грамматика, которую мы используем для Jzero, была адаптирована из диалекта Java под названием Godiva. Чтобы работать с настоящей грамматикой Java, вы можете посмотреть сайт *https://docs.oracle.com/javase/specs/*.

Спецификация Jzero lex

Спецификация Jzero lex соответствует спецификации, приведенной в предыдущей главе, с однострочным объявлением пакета в верхней части. Парсер должен быть сгенерирован до компиляции сканера. Это связано с тем, что yacc превращает *j0gram.y* в класс парсера, на значения констант которого ссылаются из сканера. Поскольку статический импорт *yylex()* влечет за собой использование пакетов, вы должны добавить следующую строку в начало файла *javalex.l* из предыдущей главы:

```
package ch4;
```

Для совместимости с *javalex.l* из предыдущей главы модуль *lexer* в тривиальном парсере в начале этой главы назван *j0* в парсере Jzero.

Понимая, что это небольшое изменение в спецификации Jzero Lex, для того чтобы вызывать его из парсера, давайте перейдем к следующему разделу, чтобы узнать о спецификации Jzero yacc.

Спецификация yacc в Jzero

По сравнению с предыдущими примерами, спецификация yacc реального языка программирования имеет более сложные правила производства и гораздо большее их число. Следующий файл называется *j0gram.y* и представлен в нескольких частях.

Первый раздел *j0gram.y* содержит заголовок и объявления терминальных символов. Эти объявления являются источником символьных констант в классе парсера, использованном в предыдущей главе. Недостаточно, чтобы имена

совпадали в сканере и парсере. Чтобы эти два инструмента могли общаться, их целочисленные коды должны быть одинаковыми. Сканер должен возвращать целочисленные коды парсера для своих терминальных символов. Согласно предыдущему описанию раздела заголовка *yacc*, объявления терминальных символов делаются путем указания их имени в строке, начинающейся с `%token`. Jzero объявляет примерно 27 символов для зарезервированных слов, различных видов литеральных констант и многосимвольных операторов:

```
%token BREAK DOUBLE ELSE FOR IF INT RETURN VOID WHILE
%token IDENTIFIER CLASSNAME CLASS STRING BOOL
%token INTLIT DOUBLELIT STRINGLIT BOOLLIT NULLVAL
%token LESSTHANOREQUAL GREATERTHANOREQUAL
%token ISEQUALTO NOTEQUALTO LOGICALAND LOGICALOR
%token INCREMENT DECREMENT
%%
```

После %% идут правила производства бесконтекстной грамматики языка, который мы определяем. По умолчанию нетерминал первого правила в списке является начальным нетерминалом, который в Jzero обозначает один целый исходный файл, модуль или единицу компиляции. В Jzero это всего лишь один класс. Это серьезное упрощение языка Java, в котором обычно существует несколько объявлений, таких как импорт, перед классом в определенном исходном файле.

Объявление класса состоит из слова *class*, за которым следует идентификатор, дающий имя класса, за которым следует тело:

```
ClassDecl:            CLASS IDENTIFIER ClassBody ';' ;
```

Тело класса – это последовательность объявлений полей, методов и конструкторов. Обратите внимание, что правила производства для тела класса (`ClassBody`) допускают ноль или более вхождений объявлений в фигурных скобках – одно правило требует список из одного или более `ClassBodyDecls`, в то время как второе правило явно допускает необычный, но законный случай пустого класса:

```
ClassBody:        '{' ClassBodyDecls '}' | '{' '}' ;
ClassBodyDecls:   ClassBodyDecl | ClassBodyDecls
                  ClassBodyDecl;
ClassBodyDecl:    FieldDecl | MethodDecl | ConstructorDecl ;
```

Объявления полей состоят из типа, за которым следует список переменных, разделенных запятыми. Идентификатор, следующий за словом *class*, становится именем типа. Некоторые реализации языка заставляют лексический анализатор сообщать другой целочисленный код категории, если слово стало именем типа, а не именем переменной. Jzero этого не делает:

```
FieldDecl:          Type VarDecls ';' ;
Type:               INT | DOUBLE | BOOL | STRING | Name ;
Name:               IDENTIFIER | QualifiedName ;
```

```
QualifiedName:        Name '.' IDENTIFIER ;
VarDecls:             VarDeclarator | VarDecls ',' VarDeclarator;
VarDeclarator:        IDENTIFIER | VarDeclarator '[' ']' ;
```

Следующая часть *j0gram.y* состоит из правил синтаксиса для двух других типов объектов, которые могут быть объявлены внутри класса и использовать синтаксис функций – методов и конструкторов. Начнем с того, что они имеют немного разные заголовки, за которыми следует блок операторов:

```
MethodDecl: MethodHeader Block ;
ConstructorDecl: FuncDeclarator Block ;
```

Заголовки методов имеют возвращаемый тип, но в остальном методы и конструкторы имеют одинаковый синтаксис в виде общего использования нетерминалов `FuncDeclarator` и `Block`:

```
MethodHeader: Type FuncDeclarator | VOID FuncDeclarator ;
```

За именем функции (или, в случае конструктора, именем класса) следует список параметров в круглых скобках:

```
FuncDeclarator: IDENTIFIER '(' FormalParmListOpt ')' ;
```

Список параметров – это ноль или более параметров. Нетерминал `FormalParmListOpt` имеет два правила производства – либо существует (непустой) `FormalParmList`, либо нет. Пустое правило после вертикальной черты называется **правилом эпсилона**:

```
FormalParmListOpt: FormalParmList | ;
```

Список формальных параметров – это список, разделенный запятыми, где каждый формальный параметр состоит из типа и имени переменной:

```
FormalParmList: FormalParm | FormalParmList ',' FormalParm;
FormalParm: Type VarDeclarator ;
```

Следующая часть *j0gram.y* содержит **грамматику высказываний**. Высказывание – это фрагмент кода, который не предоставляет значения для использования окружающим кодом. В Jzero есть несколько видов высказываний. Блок (`Block`, например, тело метода) – это высказывание, состоящее из последовательности (под)высказываний, заключенных в фигурные скобки {}:

```
Block: '{' BlockStmtsOpt '}' ;
```

Поскольку блок может содержать нулевые подвысказывания, используется нетерминал с правилом эпсилон:

```
BlockStmtsOpt: BlockStmts | ;
```

При использовании без необязательного случая несколько BlockStmt объединяются в цепочку, используя рекурсию:

```
BlockStmts: BlockStmt | BlockStmts BlockStmt ;
```

Типы высказываний, допустимых в блоке, включают объявления переменных и обычные исполняемые операторы:

```
BlockStmt: LocalVarDeclStmt | Stmt ;
```

Объявления локальных переменных состоят из типа, за которым следует список имен переменных, разделенных запятой, заканчивающийся точкой с запятой. Нетерминал VarDecls был представлен там, где он ранее использовался в объявлениях переменных класса:

```
LocalVarDeclStmt: LocalVarDecl ';' ;
LocalVarDecl: Type VarDecls ;
```

Существует множество видов обычных исполняемых операторов, включая выражения, операторы break и return, выражения if, а также циклы while и for:

```
Stmt:    Block | ';' | ExprStmt | BreakStmt | ReturnStmt
     | IfThenStmt | IfThenElseStmt | IfThenElseIfStmt
     | WhileStmt | ForStmt ;
```

Большинство выражений порождают значение, которое должно быть использовано в окружающем выражении. Три вида выражений можно превратить в высказывание, поставив после них точку с запятой:

```
ExprStmt: StmtExpr ';' ;
StmtExpr: Assignment | MethodCall | InstantiationExpr ;
```

Предусмотрено несколько форм выражений if, позволяющих создавать цепочки выражений else. Если они кажутся избыточными, то это потому, что подмножество Jzero языка Java обычно требует, чтобы в телах условных и циклических конструкций использовались фигурные скобки, что позволяет избежать распространенного источника ошибок:

```
IfThenStmt:          IF '(' Expr ')' Block ;
IfThenElseStmt:      IF '(' Expr ')' Block ELSE Block ;
IfThenElseIfStmt:    IF '(' Expr ')' Block ElseIfSequence
     | IF '(' Expr ')' Block ElseIfSequence ELSE Block ;
ElseIfSequence:      ElseIfStmt | ElseIfSequence ElseIfStmt ;
ElseIfStmt:          ELSE IfThenStmt ;
```

Циклы WHILE имеют простой синтаксис, похожий на выражения IF:

```
WhileStmt: WHILE '(' Expr ')' Block ;
```

С другой стороны, циклы FOR являются достаточно сложными:

```
ForStmt: FOR '(' ForInit ';' ExprOpt ';' ForUpdate ')' Block ;
ForInit:              StmtExprList | LocalVarDecl | ;
ExprOpt:              Expr | ;
ForUpdate:            StmtExprList | ;
StmtExprList:         StmtExpr | StmtExprList ',' StmtExpr ;
```

Операторы BREAK и RETURN очень просты, единственное отличие в их синтаксисе заключается в том, что RETURN может иметь необязательное выражение после него. Методы *VOID* возвращаются без этого выражения, в то время как методы без *VOID* должны включать его. Это должно быть проверено во время семантического анализа:

```
BreakStmt: BREAK ';' ;
ReturnStmt: RETURN ExprOpt ';' ;
```

Следующая часть *j0gram.y* содержит **грамматику выражений**. **Выражение** – это фрагмент кода, который вычисляет значение, обычно для использования в окружающем выражении. Эта грамматика выражений использует один нетерминальный символ на каждый уровень приоритета операторов. Например, утверждение, что умножение имеет более высокий приоритет, чем сложение, заключается в том, что все умножения выполняются нетерминалом MulExpr, а затем экземпляры MulExpr объединяются вместе с помощью операторов плюс (или минус) в правилах производства AddExpr:

```
Primary: Literal | '(' Expr ')' | FieldAccess | MethodCall;
Literal: INTLIT | DOUBLELIT | BOOLLIT | STRINGLIT |NULLVAL;
InstantiationExpr: Name '(' ArgListOpt ')' ;
ArgList: Expr | ArgList ',' Expr ;
ArgListOpt: ArgList | ;
FieldAccess: Primary '.' IDENTIFIER ;
MethodCall: Name '(' ArgListOpt ')'
     | Name '{' ArgListOpt '}'
     | Primary '.' IDENTIFIER '(' ArgListOpt ')'
     | Primary '.' IDENTIFIER '{' ArgListOpt '}' ;
PostFixExpr: Primary | Name ;
UnaryExpr: '-' UnaryExpr | '!' UnaryExpr | PostFixExpr ;
MulExpr: UnaryExpr | MulExpr '*' UnaryExpr
       | MulExpr '/' UnaryExpr | MulExpr '%' UnaryExpr ;
AddExpr: MulExpr | AddExpr '+' MulExpr | AddExpr '-'
MulExpr ;
RelOp: LESSTHANOREQUAL | GREATERTHANOREQUAL | '<' | '>' ;
RelExpr: AddExpr | RelExpr RelOp AddExpr ;
EqExpr: RelExpr | EqExpr ISEQUALTO RelExpr | EqExpr
NOTEQUALTO RelExpr ;
CondAndExpr: EqExpr | CondAndExpr LOGICALAND EqExpr ;
CondOrExpr: CondAndExpr | CondOrExpr LOGICALOR CondAndExpr;
Expr: CondOrExpr | Assignment ;
Assignment: LeftHandSide AssignOp Expr ;
LeftHandSide: Name | FieldAccess ;
AssignOp: '=' | INCREMENT | DECREMENT ;
```

Хотя для представления здесь он разбит на пять частей, файл *j0gram.y* не очень большой – около 120 строк кода. Поскольку он работает как для Unicon, так и для Java, это очень выгодно для вас. Поддержка кода Unicon и Java нетривиальна, но мы позволяем *yacc* (iyacc и BYACC/J) сделать большую часть работы. Файл *j0gram.y* станет больше в следующей главе, когда мы расширим парсер для построения деревьев синтаксиса.

Теперь пришло время рассмотреть вспомогательный код Unicon Jzero, который вызывает и работает с грамматикой Jzero yacc.

Код Unicon Jzero

Реализация парсера Jzero в Unicon использует почти такую же организацию, как и в предыдущей главе, начиная с файла с именем *j0.icn*. Вместо вызова `yylex()` в цикле в программе на основе *yacc* процедура `main()` обращается к `yyparse()`, которая вызывает `yylex()` каждый раз при выполнении операции сдвига.

Как было сказано в предыдущей главе, сканер Unicon использует объект `parser`, поля которого, такие как `parser.WHILE`, содержат целочисленные коды категорий. Объект парсера больше не находится в *j0.icn*, теперь он генерируется yacc в файле *j0gram.icn*, который громаден и не будет здесь показан:

```
global yylineno, yycolno, yylval, parser
procedure main(argv)
      j0 := j0()
      parser := Parser()
      yyin := open(argv[1]) | stop("usage: j0 filename")
      yylineno := yycolno := 1
      if yyparse()=0 then
              write("no errors, ", j0.count, " tokens parsed")
end
```

Вторая часть *j0.icn* состоит из класса `j0` (см. пояснения в главе 3, в разделе «Код Unicon Jzero»):

```
class j0(count)
      method lexErr(s)
              stop(s, ": ", yytext)
      end
      method scan(cat)
              yylval := token(cat, yytext, yylineno, yycolno)
              yycolno +:= *yytext
              count +:= 1
              return cat
      end
      method whitespace()
              yycolno +:= *yytext
      end
      method newline()
              yylineno +:= 1; yycolno := 1
      end
```

```
      method comment()
              yytext ? {
                      while tab(find("\n")+1) do newline()
                      yycolno +:= *tab(0)
              }
      end
      method ord(s)
              return proc("ord",0)(s[1])
      end
initially
        count := 0
end
```

В третьей части *j0.icn* тип токена с его методом `deEscape()` сохранен из предыдущей главы:

```
class token(cat, text, lineno, colno, ival, dval, sval)
      method deEscape(sin)
              local sout := ""
              sin := sin[2:-1]
              sin ? {
                      while c := move(1) do {
                              if c == "\\" then {
                                      if not (c := move(1)) then
                                              j0.lexErr("malformed string
                                      literal")
                                      else case c of {
                                              "t":{ sout ||:= "\t" }
                                              "n":{ sout ||:= "\n" }
                                              }
                                      }
                              else sout ||:= c
                              }
              }
              return sout
      end
initially
      case cat of {
              parser.INTLIT: { ival := integer(text) }
              parser.DOUBLELIT: { dval := real(text) }
              parser.STRINGLIT: { sval := deEscape(text) }
      }
end
```

Вы можете заметить, что код **Unicon Jzero** в этой главе стал немного короче по сравнению с предыдущей благодаря тому, что *yacc* сделал часть работы за нас. Теперь давайте посмотрим на соответствующий код на Java.

Код парсера Jzero на языке Java

Java-реализация парсера Jzero включает главный класс в файле *j0.java*. Он похож на тот же файл из предыдущей главы, за исключением того, что его функция `main()` вызывает `yyparse()`:

```
package ch4;
import java.io.FileReader;
public class j0 {
      public static Yylex lex;
      public static parser par;
      public static int yylineno, yycolno, count;
      public static void main(String argv[]) throws Exception
      {
              lex = new Yylex(new FileReader(argv[0]));
              par = new parser();
              yylineno = yycolno = 1;
              count = 0;
              int i = par.yyparse();
              if (i == 0) {
                      System.out.println("no errors, " + j0.count +
                                         " tokens parsed");
              }
      }
      // остальные методы j0.java такие же, как и в главе 3.
}
```

Чтобы запустить программу, вам также придется скомпилировать модуль с именем *parser.java*, который генерируется *yacc* из нашего входного файла *j0gram.y*. Этот модуль предоставляет функцию `yyparse()` вместе с набором именованных констант, объявленных непосредственно, как короткие целые числа. Хотя в этой книге приводится *j0gram.y*, а не файл *parser.java*, который генерируется на его основе, вы можете запустить *yacc* и посмотреть на его результаты самостоятельно.

Существует также вспомогательный модуль *token.java*, который содержит класс *token*. Он идентичен тому, который был представлен в предыдущей главе, поэтому мы не дублируем его здесь.

Если вы любите планировать заранее, то, возможно, вам будет интересно узнать, что экземпляры класса *token* содержат именно ту информацию, которая вам нужна в листьях дерева синтаксиса, которое вы будете строить в следующей главе. Существуют различные способы, которыми можно включить эту лексическую информацию в листья дерева. Мы рассмотрим это в главе 5 «Деревья синтаксиса».

Запуск парсера Jzero

Вы можете запустить программу в Unicon или Java следующим образом. На этот раз давайте запустим программу на примере входного файла с именем *hello.java*:

```
public class hello {
      public static void main(String argv[]) {
              System.out.println("hello, jzero!");
      }
}
```

Помните, что для вашего парсера программа *hello.java* является последовательностью лексем, которые должны быть проверены на соответствие грамматике языка Jzero, приведенной ранее. Команды для компиляции и запуска парсера Jzero похожи на предыдущие примеры, с добавлением большего количества файлов. Команды Unicon выглядят следующим образом:

```
uflex javalex.l
iyacc -dd j0gram.y
unicon j0 javalex j0gram j0gram_tab yyerror
j0 hello.java
```

Машинно-сгенерированный код, выводимый uflex для *javalex.l*, содержит единственную функцию, достаточно большую, чтобы вызвать сбой стандартной версии генератора кода Unicon (*icont*) из-за переполнения стека разбора! Мне пришлось модифицировать грамматику *icont yacc*, чтобы использовать больший стек для запуска этого примера.

В предпоследней строке предыдущего списка команд компиляция исполняемого файла *j0* с одним вызовом для выполнения компиляции и линковки – это выбор ленивого представления в Unicon. В Java существует достаточно циклическая зависимость между лексическим анализатором (который использует целочисленные константы парсера) и парсером (который вызывает `yylex()`), для того чтобы вам пришлось постоянно прибегать к модели компиляции «с большим вдохом». Хотя это печально, но если это нужно для того, чтобы Java плавно объединил *jflex* и *BYACC/J*, давайте просто расслабимся и насладимся этим.

> **Модель «большого вдоха»**
>
> Все серьезные языки программирования, особенно объектно-ориентированные, позволяют компилировать модули отдельно, и фактически поощряют модули быть маленькими, так что сборка состоит из множества маленьких компиляций модулей. Когда некоторый код изменяется, только небольшая часть всей программы должна быть перекомпилирована. К сожалению, многие особенности языка программирования, в данном случае классы, использующие статические члены друг друга, могут привести к необходимости компиляции нескольких или многих модулей одновременно в Java… весьма иронично для языка, который избегает линковки. Иногда вы можете найти последовательность одиночных компиляций, которая будет работать в Java, а иногда нет. Когда вам нужно представить многие или все исходные файлы Java в командной строке одновременно, то, что было бы неразумным шагом для C/C++-программиста, становится обычным и необходимым для Java-программиста. Не парьтесь. Для этого и существуют быстрые многоядерные процессоры и сверхпроектированные IDE.

Команды Java для создания и запуска парсера `j0` следующие:

```
jflex javalex.l
yacc -Jclass=parser -Jpackage=ch4 -Jyylex=ch4.j0.yylex\
        -Jyyerror=ch4.yyerror.yyerror j0gram.y
javac parser.java Yylex.java j0.java parserVal.java \
             token.java yyerror.java
java ch4.j0 hello.java
```

Из Unicon- или Java-реализации вы должны увидеть результат, подобный следующему:

```
no errors, 26 tokens parsed
```

Не очень интересный результат. Парсер Jzero станет гораздо более полезным в следующей главе, когда вы научитесь строить структуру данных, представляющую собой запись полной синтаксической структуры исходной входной программы. Эта структура данных является фундаментальным скелетом, на котором основывается любая реализация интерпретатора или компилятора языка программирования. А что, если мы дадим исходный файл, в котором отсутствуют некоторые требуемые знаки пунктуации или используются некоторые конструкции Java, которых нет в Jzero? Мы ожидаем сообщение об ошибке. Следующий пример входного файла с именем *helloerror.java* служит для мотивировки нашего следующего раздела:

```
public class hello {
     public static void main(String argv[]) {
             System.out.println("hello, jzero!")
     }
}
```

Вы видите ошибку? Это самая старая и самая распространенная синтаксическая ошибка из всех. После оператора `println()` отсутствует точка с запятой. Основываясь на написанном до сих пор парсере, запуск `j0 helloerror.java` пока что выводит следующее сообщение об ошибке *yacc* по умолчанию и завершает работу:

```
parse error
```

Если *no errors* неинтересно, то говорить об ошибке разбора (*parse error*) при возникновении проблемы – это совсем неудобно для пользователя. Пришло время подумать о сообщениях о синтаксических ошибках и их устранении.

Улучшение сообщений об ошибках синтаксиса

Ранее мы немного познакомились с механизмом сообщений о синтаксических ошибках *yacc*. *Yacc* просто обращается к функции с именем `yyerror(s)`. Очень редко эта функция может быть вызвана для внутренней ошибки, например переполнения стека разбора, но обычно, когда она вызывается, ей передается

строка «*ошибка разбора*» или «*синтаксическая ошибка*» в качестве параметра. Ни то, ни другое не подходит для того, чтобы помочь программистам находить и исправлять ошибки в реальной жизни. Если вы напишете функцию под названием `yyerror()`, вы можете создать более качественное сообщение об ошибке. Главное – иметь дополнительную информацию, которую может использовать программист. Обычно эта дополнительная информация должна быть помещена в глобальную или публичную статическую переменную, чтобы функция `yyerror()` могла получить к ней доступ. Давайте рассмотрим, как написать улучшенную функцию `yyerror()` в Unicon, а затем в Java.

Добавление деталей в сообщения Unicon об ошибках синтаксиса

В разделе этой главы «*Создание игрушечного примера*» вы видели реализацию Unicon *yyerror(s)*, которая состояла только из вызова `stop(s)`. Легко сделать лучше, особенно если у нас есть глобальные переменные, такие как `yylineno`. В Unicon ваша функция `yyerror()` может выглядеть следующим образом:

```
procedure yyerror(s)
        write(&errout, "line ", yylineno, " column ", yycolno,
                       ", lexeme \"", yytext, "\": ", s)
end
```

Она выводит номера строки и столбца, а также текущую лексему на момент, когда была обнаружена синтаксическая ошибка. Поскольку `yylineno`, `yycolno` и `yytext` являются глобальными переменными, нет проблем с доступом к ним из вспомогательной процедуры `yyerror()`. Главное, что вы, возможно, захотите сделать еще лучше, чем это, – придумать, как выдать сообщение, которое будет более полезным, чем просто сообщение об ошибке разбора.

Добавление деталей в сообщения Java об ошибках синтаксиса

Соответствующая функция Java `yyerror()` приведена ниже. В BYACC/J вы можете поместить этот метод в раздел вспомогательных функций *j0gram.y*, где он будет включен в класс *Parser*, из которого он вызывается. К сожалению, если вы сделаете это, то откажетесь от переносимости Unicon/Java в файле спецификации *yacc*. Поэтому вместо этого мы помещаем `yyerror()` в свой собственный класс и свой собственный файл. Этот пример показывает степень пафоса, вызванного получистой объектно-ориентированной моделью Java, где все должно быть в классе, даже если это бессмысленно:

```
public class yyerror {
     public static void yyerror(String s) {
             System.err.println(" line "+ j0.yylineno +
                     " column "+ j0.yycolno +
                     ", lexeme \""+ j0.yytext()+ "\": "+ s);
     }
}
```

Как мы видели ранее в данной главе, использование этой `yyerror()` из другого файла внутри класса парсера, сгенерированного BYACC/J, требует объявления *import static*, для которого мы добавили опции командной строки *-Jyylex=...* и *-Jyyerror=...* в BYACC/J. В реализации Unicon или Java, когда вы подключите эту `yyerror()` в свой парсер `j0` и запустите `j0 helloerror.java`, то увидите результат, похожий на следующий:

```
line 4 column 1, lexeme «end»: parse error
```

До недавнего времени это удавалось сделать многим промышленным компиляторам, таким как **gcc**. Для опытного программиста этого вполне достаточно. Посмотрев до и после места ошибки, он увидит пропущенную точку с запятой. Но для новичка или программиста среднего уровня в неудачный день даже номера строки, столбца и токена, в котором обнаружена ошибка, недостаточно. Хорошие инструменты языка программирования должны быть способны выдавать лучшие сообщения об ошибках.

Использование Merr для создания лучших сообщений об ошибках синтаксиса

Как написать лучшее сообщение, четко указывающее на ошибку синтаксического разбора? Алгоритм разбора рассматривал два целых числа, когда понял, что произошла ошибка, – **состояние разбора** и **текущий входной символ**. Если вы можете сопоставить эти два целых числа с набором лучших сообщений об ошибках, вы выиграли. К сожалению, выяснить, что означают целочисленные состояния разбора, нетривиально. Вы можете сделать это путем мучительных проб и ошибок, но каждый раз, когда вы меняете грамматику, эти числа меняются.

Именно для решения этой проблемы был создан инструмент, названный **Merr** (для **Meta error**). Merr находится по адресу *http://unicon.org/merr*. В качестве входных данных он принимает имя вашего компилятора, *makefile* для его сборки и файл спецификации *meta.err*, который содержит список фрагментов ошибок и соответствующих им сообщений об ошибках. Чтобы сгенерировать `yyerror()`, Merr собирает ваш компилятор и запускает его в режиме, который заставляет его выводить состояние разбора и текущий входной токен при каждой ошибке фрагмента. Затем он выводит `yyerror()`, которая содержит таблицу, показывающую для каждого состояния разбора и фрагмента ошибки связанное с ними сообщение. Пример файла *meta.err* для нескольких ошибок, включая пропущенную точку с запятой, показанную ранее, выглядит следующим образом:

```
public {
::: class expected
public class {
::: missing class name
public class h public
::: { expected
public class h{public static void m(S a[]){S.o.p(«h»)}}
::: semi-colon expected
```

Вы вызываете инструмент *Merr*, сообщая ему имя компилятора, который вы собираете, он использует это имя в качестве целевого аргумента при вызове *make* для сборки вашего компилятора. Различные опции командной строки позволяют указать версию *yacc* и другие важные детали. Следующие командные строки вызывают *merr* на Unicon (слева) или Java (справа):

```
merr -u j0                    merr -j j0.class
```

Эта команда работает некоторое время, *merr* перестраивает ваш компилятор с модифицированной функцией `yyerror()`, чтобы сообщить о состоянии разбора и входном токене во время каждой ошибки. Затем Merr запускает ваш компилятор на каждом из фрагментов ошибок и записывает, в каком состоянии разбора они произошли. Наконец, *merr* выдаёт `yyerror()`, содержащую таблицу, отображающую состояния разбора с соответствующими сообщениями об ошибках.

Как вы видели на примере Unicon и Java, написать сообщение об ошибке, включающее номер строки или текущий входной символ, когда обнаружена синтаксическая ошибка, очень просто. С другой стороны, сказать что-то более полезное по этому поводу может быть непросто.

Заключение

В этой главе вы узнали о важнейших технических навыках и инструментах, используемых в языках программирования при парсинге последовательности лексем исходного кода программы для проверки его организации и структуры.

Вы научились писать бесконтекстные грамматики и использовать инструменты iyacc и BYACC/J для получения бесконтекстной грамматики и создания парсера для нее.

Когда вводимые данные не соответствуют правилам, вызывается функция сообщения об ошибках `yyerror()`. Вы узнали некоторые основы этого механизма обработки ошибок.

Вы узнали, как вызывать сгенерированный парсер из функции `main()`. Сгенерированный yacc парсер вызывается с помощью функции `yyparse()`.

Теперь вы готовы узнать, как построить структуру данных дерева синтаксиса, которая отражает структуру входного исходного кода. В следующей главе будет в деталях рассмотрено построение деревьев синтаксиса.

Вопросы

1. Что на самом деле значит сказать, что грамматический символ является терминальным? Он умирает или что-то в этом роде?
2. Парсеры YACC называются shift/reduce-парсерами. Что такое сдвиг? Что такое сокращение?
3. Выполняется ли код семантического действия в грамматике YACC, когда парсер выполняет сдвиг, или сокращение, или оба действия?
4. Как синтаксический анализ использует лексический анализ, описанный в предыдущей главе?

Глава 5

Деревья синтаксиса

Парсер, который мы построили в прошлой главе, может обнаруживать и сообщать о синтаксических ошибках, это большая и важная работа. Когда синтаксических ошибок нет, в процессе парсинга необходимо построить структуру данных, которая логически представляет всю программу. Эта структура данных основана на группировании различных лексем и больших частей программы. Дерево синтаксиса – это древовидная структура данных, которая записывает ветвящуюся структуру грамматических правил, используемых алгоритмом парсинга для проверки синтаксиса входного исходного файла. Ветвление происходит, когда два или более символов группируются в правой части грамматического правила для построения нетерминального символа.

В этой главе вы узнаете, как строить деревья синтаксиса, которые являются центральной структурой данных для реализации вашего языка программирования.

В данной главе рассматриваются следующие основные темы:

- знакомство с деревьями;
- создание листьев из терминальных символов;
- создание внутренних узлов из правил производства;
- формирование деревьев синтаксиса для языка Jzero;
- отладка и тестирование вашего дерева синтаксиса.

Пришло время узнать о древовидных структурах данных и о том, как их строить. Но сначала давайте узнаем о некоторых новых инструментах, которые облегчат построение языка в оставшейся части данной книги.

Технические требования

Для этой главы вам необходимо установить два следующих инструмента:

- **Dot** является частью пакета под названием **Graphviz**, который можно загрузить со страницы, расположенной по адресу *http://graphviz.org*. После успешной установки Graphviz в вашем пути должен появиться исполняемый файл с именем *dot* (или *dot.exe*);
- **GNU's Not Unix** (**GNU**) *make* – это инструмент, помогающий управлять большими проектами программирования, которые поддерживают как Unicon, так и Java. Он доступен для Windows по адресу *http://gnuwin32.*

sourceforge.net/packages/make.htm. Большинство программистов, вероятно, получают его вместе с компилятором C/C++ или с пакетом разработки, таким как MSYS2 или Cygwin. В Linux, как правило, вы получаете *make* из пакета для разработки на C, хотя часто это также отдельный пакет, который можно установить;

- примеры из этой книги можно скачать из нашего репозитория GitHub: *https://github.com/PacktPublishing/Build-Your-OwnProgramming-Language/tree/master/ch5*.

Видеоролик «Код в действии» (Code in Action) для этой главы можно найти здесь: *https://bit.ly/3DgRcgC*.

Прежде чем мы перейдем к основным темам данной главы, давайте изучим основы использования GNU *make* и то, почему он необходим для разработки вашего языка.

Использование GNU make

Командные строки становятся все длиннее и длиннее, и вы очень устанете от набора команд, необходимых для построения языка программирования. Мы уже используем Unicon, Java, uflex, jflex, iyacc и BYACC/J. Немногие инструменты для построения больших программ являются многоплатформенными и многоязычными в достаточной степени для такого набора инструментов. Мы будем использовать самый совершенный – GNU *make*.

Как только программа *make* будет установлена, вы сможете сохранить правила сборки для Unicon или Java, или обоих в файле с именем *makefile* (или *Makefile*), а затем просто запускать *make* всякий раз, когда вы изменили код и вам нужно пересобрать его. Полное описание *make* выходит за рамки этой книги, но вот основные моменты.

makefile похож на спецификацию *lex* или *yacc*, только вместо распознавания шаблонов строк он определяет граф **зависимостей сборки** между файлами. Для каждого файла *makefile* содержит исходные файлы, от которых он зависит, а также список одной или нескольких командных строк, необходимых для сборки этого файла. Заголовок *makefile* состоит только из макросов, определяемых строками *NAME= strings*, которые используются в последующих строках путем записи *$(NAME)* для замены имени его определением. Остальные строки *makefile* – это зависимости, записанные в следующем формате:

```
file: source_file(s)
  build rule
```

В первой строке `file` – это выходной файл, который вы хотите построить, также называемый целью. Первая строка указывает, что цель зависит от текущих версий исходного файла (файлов – `source file(s)`). Они необходимы для создания цели. `build rule` – это командная строка, которую вы выполняете для создания выходного файла из этого исходного файла (файлов).

Не забудьте про табуляцию!

Программа *make* поддерживает несколько строк правил сборки, при условии что строки начинаются с табуляции. Самая распространенная ошибка новичков при написании *makefile* – они забывают, что строка(и) правил сборки должна начинаться с символа американского стандартного кода для обмена информацией (**American Standard Code for Information Interchange – ASCII**) *Ctrl-I*, также известного как символ табуляции. В некоторых текстовых редакторах это совершенно не так. Если ваши правила сборки не начинаются с символа табуляции, *make*, вероятно, выдаст вам какое-то непонятное сообщение об ошибке. Используйте настоящий редактор кода и не забывайте о табуляции.

Следующий пример *makefile* соберет и Unicon, и Java, если вы просто запустите *make*. Если вы запустите *make unicon* или *make java*, то он соберет только один или другой. Для этой главы к командам из предыдущей главы добавлен новый модуль (*tree.icn* или *tree.java*). *makefile* представлен в двух половинах, для сборки Unicon и Java соответственно.

Цель с именем `all` определяет, что собирать, если *make* вызывается без аргумента, указывающего, что собирать. Остальная часть первой половины посвящена сборке Unicon. Макросы `U` (и `LYU` для iyacc) перечисляют модули Unicon, которые отдельно компилируются в формат машинного кода, называемый **ucode**. Необычная зависимость `%.u:%.icn` называется **суффиксным правилом**. Оно гласит, что все файлы `.u` собираются из файлов `.icn` путем выполнения команды `unicon -c` над файлом `.icn`. Исполняемый файл с именем `j0` собирается из файлов *ucode* путем запуска `unicon` на все файлы `.u`, чтобы связать их вместе. Файлы `javalex.icn` и `j0gram.icn` создаются с помощью `uflex` и `iyacc` соответственно. Давайте рассмотрим первую половину нашего `makefile` для этой главы, как показано ниже:

```
all: unicon java
LYU=javalex.u j0gram.u j0gram_tab.u
U=j0.u token.u tree.u serial.u yyerror.u $(LYU)
unicon: j0
%.u : %.icn
        unicon -c $<
j0: $(U)
        unicon $(U)
javalex.icn: javalex.l
        uflex javalex.l
j0gram.icn j0gram_tab.icn: j0gram.y
        iyacc -dd j0gram.y
```

Правила сборки Java занимают вторую половину нашего *makefile*. Макрос *JSRC* задает имена всех файлов Java, которые будут скомпилированы. Макросы *BYSRC* для генерируемых BYACC/J-источников, *BYJOPTS* для опций BYACC/J, *IMP* и *BYJIMPS* для статических импортов BYACC/J служат для сокращения последующих строк в *makefile*, чтобы они вписывались в ограничения форматирования данной книги. Мы тщательно придерживаемся *makefile*, который бу-

дет работать как под Windows, так и под Linux. Напоминаю, что правила Java для нашего *makefile* зависят от переменной среды `CLASSPATH`, синтаксис которой зависит от вашей операционной системы и синтаксиса ее командной строки (или оболочки). В Windows вы можете ввести следующее:

```
set CLASSPATH=».;c:\users\username\byopl"
```

Здесь `username` – это ваше имя пользователя, в то время как в Linux вместо этого вы можете использовать следующее:

```
export CLASSPATH=..
```

В любом случае, вот вторая половина нашего *makefile*:

```
BYSRC=parser.java parserVal.java Yylex.java
JSRC=j0.java tree.java token.java yyerror.java $(BYSRC)
BYJOPTS= -Jclass=parser -Jpackage=ch5
IMP=importstatic
BYJIMPS= -J$(IMP)=ch5.j0.yylex -J$(IMP)=ch5.yyerror.yyerror
j: java
     java ch5.j0 hello.java
     dot -Tpng foo.dot >foo.png
java: j0.class
j0.class: $(JSRC)
        javac $(JSRC)
parser.java parserVal.java: j0gram.y
        yacc $(BYJOPTS) $(BYJIMPS) j0gram.y
Yylex.java: javalex.l
jflex javalex.l
```

В дополнение к правилам компиляции кода Java Java-часть *makefile* имеет искусственную цель, *make j*, которая запускает компилятор и вызывает программу *dot* для создания изображения дерева синтаксиса в формате **Portable Network Graphic (PNG)**.

Если вы считаете файлы *makefile* странными и пугающе выглядящими, не волнуйтесь – вы в хорошей компании. Это момент «красной таблетки» и «синей таблетки»[1]. Вы можете закрыть глаза и просто набрать *make* в командной строке. Или же можете вникнуть и взять на себя ответственность за этот универсальный многоязычный инструмент сборки программного обеспечения. Если вы хотите узнать больше о *make*, вам стоит прочитать книгу Столлмана (Stallman) и Макграта (McGrath) «GNU Make: программа для управления компиляцией» или одну из других прекрасных книг по *make*. Теперь пришло время перейти к деревьям синтаксиса, но сначала нужно узнать, что такое дерево и как определить тип данных дерева для использования в языке программирования.

[1] Красная таблетка и ее противоположность, синяя таблетка – популярные символы выбора между мучительной правдой реальности (красная таблетка) и блаженной ложью иллюзии (синяя). Эти термины, популяризованные в культуре научной фантастики, происходят из фильма «Матрица» (1999). – *Прим. перев.*

Изучение деревьев

С математической точки зрения **дерево** – это разновидность структуры **графа**, оно состоит из **узлов** и ребер, которые соединяют эти узлы. Все узлы в дереве связаны между собой. Узел в вершине называется **корнем**. Узлы дерева могут иметь ноль или более дочерних узлов и не более одного родителя. Узел дерева с нулевым количеством дочерних узлов называется **листом**, большинство деревьев имеют много листьев. Узел дерева, который не является листом, имеет один или более дочерних узлов и называется **внутренним узлом**. На рис. 5.1 показан пример дерева с корнем, двумя дополнительными внутренними узлами и пятью листьями.

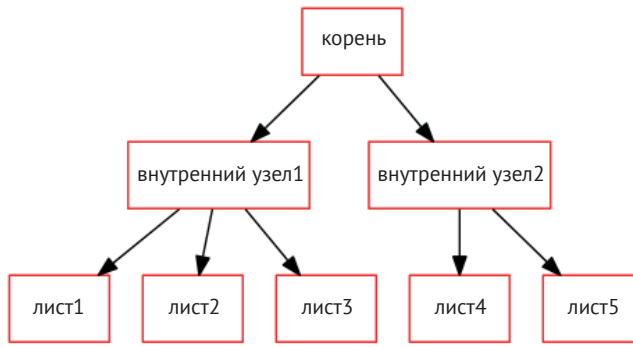


Рис. 5.1. Дерево с корнем, внутренними узлами и листьями

Деревья имеют свойство, называемое *арностью*, определяющее максимальное количество дочерних узлов, которое может иметь узел. При значении *арности*, равном 1, получается связанный список. Возможно, наиболее распространенными видами деревьев являются бинарные деревья (*арность* = 2). Нужный нам тип деревьев имеет столько дочерних узлов, сколько символов в правой части правил нашей грамматики, это так называемые **n-арные деревья**. Хотя для произвольных бесконтекстных грамматик не существует ограничения арности, для любой грамматики мы можем просто посмотреть, какое правило производства имеет больше всего символов в правой части, и при необходимости закодировать *арность* нашего дерева в соответствии с их числом. В *j0gram.y* из предыдущей главы *арность* Jzero равна 9, хотя большинство нелистовых узлов будут иметь от двух до четырех дочерних узлов. В следующих подразделах мы погрузимся глубже и узнаем, как определять деревья синтаксиса, и поймем разницу между деревом разбора и деревом синтаксиса.

Определение типа дерева синтаксиса

Каждый узел в дереве имеет несколько фрагментов информации, которые должны быть представлены в классе или типе данных, используемом для узлов дерева. Сюда входит следующая информация:

- метки или целочисленные коды, которые однозначно идентифицируют узел и его тип;
- полезная нагрузка данных, состоящая из любой информации, связанной с этим узлом;

- информация о дочерних узлах, в том числе о том, сколько у него дочерних узлов, и ссылки на эти дочерние узлы (если таковые имеются).

Мы используем класс для этой информации, чтобы максимально упростить отображение на языке Java. Здесь приведен набросок класса дерева с его полями и кодом конструктора, методы будут представлены в последующих разделах этой главы. Информация о дереве может быть представлена в Unicon в файле с именем *tree.icn* следующим образом:

```
class tree(id, sym, rule, nkids, tok, kids)
initially(s,r,x[])
     id := serial.getid(); sym := s; rule := r
     if type(x[1]) == "token__state" then {
             nkids:=0; tok := x[1]
     } else { nkids := *x; kids := x }
end
```

Класс дерева имеет следующие поля:

- поле *id* – это уникальный целочисленный идентификатор или порядковый номер, который используется для различения узлов дерева друг от друга. Оно инициализируется вызовом метода *getid()* в одноэлементном классе с именем *serial*, который будет представлен позже в этом разделе;
- строку *label* – это человекочитаемое описание для целей отладки;
- член с именем *rule* хранит правило производства (или, в случае листа, целочисленную категорию), которое представляет узел. Yacc не предоставляет числового кодирования для правил производства, поэтому вам придется создать свое собственное, независимо от того, будете ли вы просто считать правила, начиная с 1, или более сложное. Если вы начнете с 1,000 или будете использовать отрицательные числа, то никогда не спутаете номер правила производства с кодом терминального символа;
- член с именем *nkids* содержит количество дочерних узлов под этим узлом. Обычно это 0, что указывает на лист, или 2 или более, что указывает на внутренний узел;
- член с именем *tok* хранит лексические атрибуты листового узла, которые поступают к нам через функцию `yylex()`, устанавливающую переменную `yylval` парсера, как обсуждалось в главе 2 «Дизайн языка программирования»;
- член с именем *kids* представляет собой массив объектов дерева.

Соответствующий код Java выглядит как класс `tree` в файле с именем *tree.java*. Его члены соответствуют полям класса `tree` Unicon, приведенным ранее:

```
package ch5;
class tree {
     int id;
     String sym;
     int rule;
     int nkids;
     token tok;
     tree kids[];
```

Файл *tree.java* продолжается двумя конструкторами для класса `tree` – один для листьев, который принимает объект `token` в качестве аргумента, и один для внутренних узлов, который принимает дочерние узлы. Их можно увидеть в следующем фрагменте кода:

```
        public tree(String s, int r, token t) {
                id = serial.getid();
                sym = s; rule = r; tok = t; }
        public tree(String s, int r, tree[] t) {
                id = serial.getid();
                sym = s; rule = r; nkids = t.length;
                kids = t;
        }
}
```

Эта пара конструкторов инициализирует поля дерева очевидным образом. Вам может быть любопытно узнать об **идентификаторах (ID)**, инициализируемых из класса *serial*. Они используются для того, чтобы придать каждому узлу уникальную идентичность, необходимую инструменту, который рисует для нас деревья синтаксиса в графическом виде в конце этой главы. Прежде чем перейти к использованию этих конструкторов, давайте рассмотрим два различных подхода к проектируемым нами деревьям.

Деревья разбора в сравнении с деревьями синтаксиса

Дерево разбора – это то, что получается, когда вы выделяете внутренний узел для каждого правила производства, используемого при парсинге входных данных. Дерево разбора – это полная расшифровка того, как парсер сопоставил входные данные с помощью грамматики. Они слишком велики и громоздки, чтобы использовать их на практике. В реальных языках программирования существует много-много нетерминальных правил, которые строят нетерминал из одного нетерминала в правой части. Это приводит к появлению «плачущего» дерева. На рис. 5.2 показаны высота и форма дерева разбора для тривиальной программы «Hello World». Если вы построите полное дерево разбора, это существенно замедлит работу остальной части вашего компилятора.

Дерево синтаксиса имеет внутренний узел всякий раз, когда правило производства имеет два или более дочерних узлов с правой стороны и дерево должно разветвляться. На рис. 5.3 показано дерево синтаксиса для той же программы *hello.java*. Обратите внимание на различия в размере и форме по сравнению с деревом разбора, показанным на рис. 5.2.

Хотя дерево разбора может быть полезно для изучения или отладки алгоритма парсинга, в реализации языка программирования используется гораздо более простое дерево. Вы увидите это, в особенности когда мы представим правила построения узлов дерева для нашего примера языка в разделе «*Формирование деревьев синтаксиса для языка Jzero*».

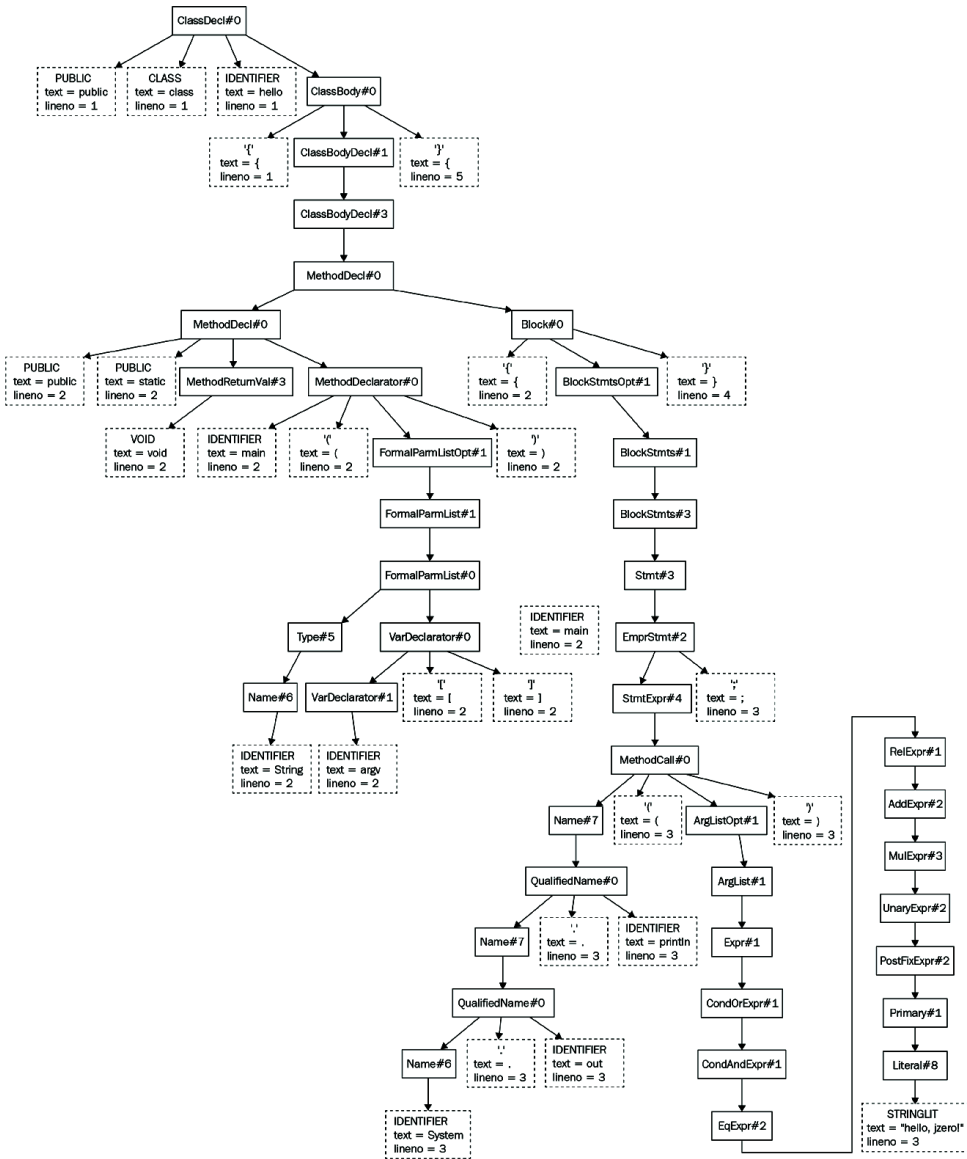


Рис. 5.2. Дерево разбора для программы «Hello World» (67 узлов, высота 27)

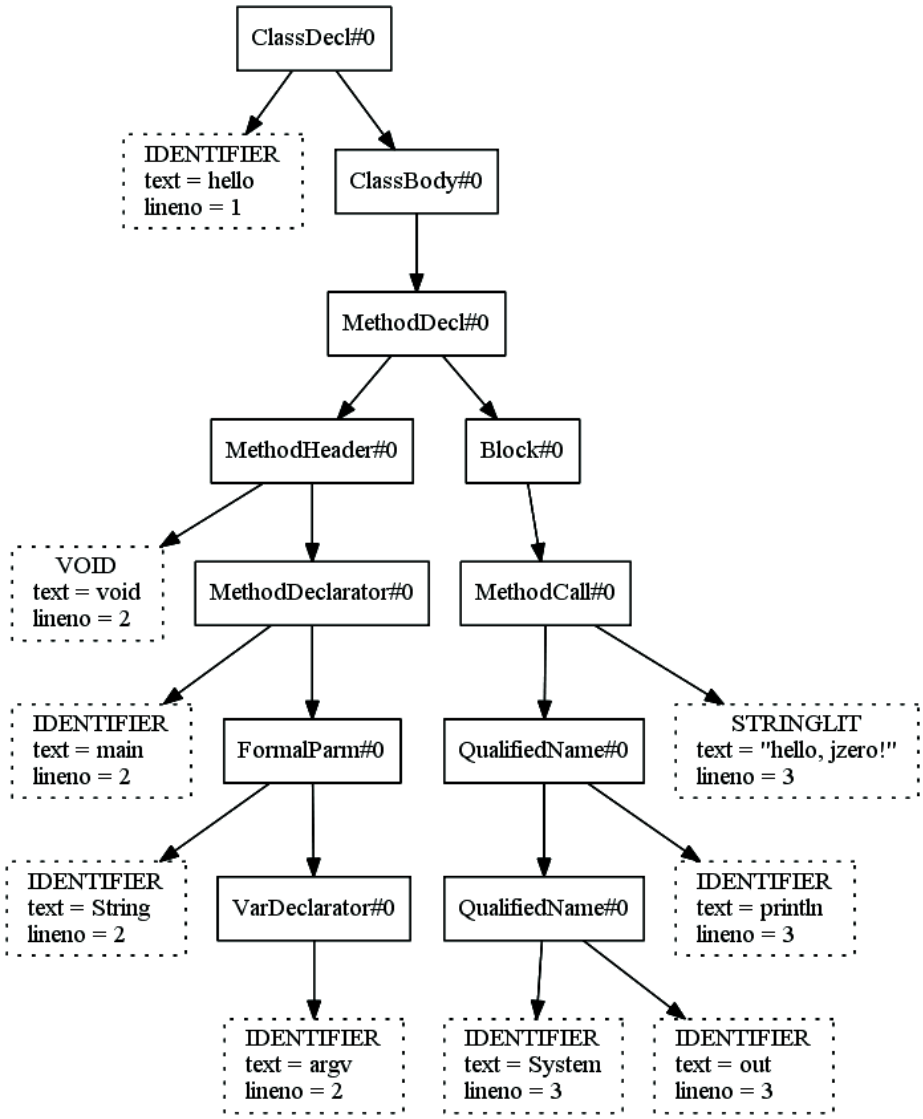


Рис. 5.3. Дерево синтаксиса для программы «Hello World» (20 узлов, высота 8)

Создание листьев из терминальных символов

Листья составляют большой процент узлов в дереве синтаксиса. Листья в дереве синтаксиса, построенном уасс, поступают из лексического анализатора. По этой причине в данном разделе обсуждаются модификации кода из главы 2 «*Дизайн языка программирования*». После создания листьев в лексическом анализаторе алгоритм парсинга должен их как-то подхватить и вставить в дерево, которое он строит. В данном разделе подробно описывается этот процесс. Во-первых, вы узнаете, как встраивать структуры лексем в листья дерева, а за-

тем изучите, как эти листья подхватываются парсером в его стеке значений. Для работы с Java вам потребуется знать о дополнительном типе, который необходим для работы со стеком значений. Наконец, в разделе даются некоторые рекомендации о том, какие листья действительно необходимы, а какие можно смело опустить. Вот как создать листья, содержащие информацию о токенах.

Обертывание токенов в листья

Представленный ранее тип дерева содержит поле, которое является ссылкой на тип токена, представленный в главе 2 «*Дизайн языка программирования*». Каждому листу будет соответствовать токен, и наоборот. Считайте это обертыванием токена в лист дерева. На рис. 5.4 показана диаграмма **унифицированного языка моделирования** (Unified Modeling Language –UML), которая изображает лист дерева, содержащий токен.

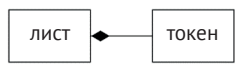


Рис. 5.4. Диаграмма листа, содержащего токен

Вместо этого можно добавить поля-члены типа токена непосредственно в тип дерева. Однако стратегия выделения объекта токена, а затем отдельного узла дерева, который содержит указатель на этот объект токена, достаточно чиста и проста для понимания. В Unicon код для создания листа выглядит следующим образом:

```
yylval := tree(«token»,0, token(cat, yytext, yylineno))
```

В Java создание листового узла, содержащего токен, выглядит почти как следующий код:

```
yylval = new tree(«token»,0,
        new token(cat, yytext(), yylineno));
```

Вы можете поместить этот код в метод *j0.scan()*, который вызывается для каждого токена лексического анализатора. В Unicon на этом этапе все в порядке. А в статически типизированных языках, таких как Java, каким типом данных является `yylval`? В главе 2 «*Дизайн языка программирования*» `yylval` был типом `token`, теперь он выглядит как тип `tree`. Но `yylval` объявляется в генерируемом парсере, и yacc ничего не знает о ваших типах `token` или `tree`. Для реализации на Java вы должны узнать тип данных, который код, сгенерированный yacc, использует для листьев, но сначала вам нужно узнать о стеке значений.

Работа со стеком значений YACC

BYACC/J не знает о вашем классе tree. По этой причине он генерирует свой стек значений как массив объектов, тип которых называется *parserVal*. Если вы переименуете класс парсера BYACC/J во что-то другое, например *myparse*, используя параметр командной строки *-Jclass=*, то класс стека значений также будет автоматически переименован в *myparseVal*.

Переменная `yylval` является частью публичного интерфейса уасс. Каждый раз, когда уасс перемещает следующий терминальный символ в свой стек разбора, он копирует содержимое `yylval` в стек, которым он управляет параллельно со стеком разбора, называемый стеком значений. BYACC/J объявляет элементы стека значений, а также `yylval` в классе парсера как элементы типа *parserVal*.

Поскольку стек разбора управляется параллельно со стеком значений, всякий раз, когда в стек разбора вставляется новое состояние, стек значений видит соответствующую вставку. То же самое относится и к операциям pop. Элементы стека значений, чье состояние разбора было получено операцией shift, содержат листья дерева. Элементы стека значений, чье состояние разбора было получено операцией reduce, содержат внутренние узлы дерева синтаксиса. На рис. 5.5 показан стек значений параллельно со стеком разбора.

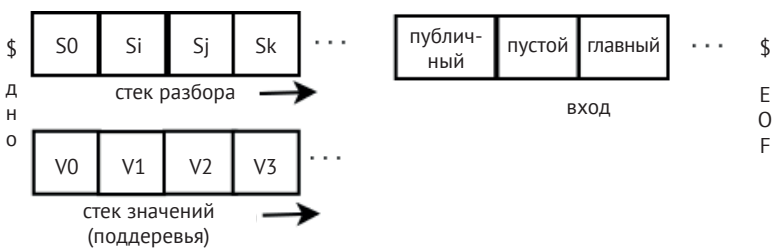


Рис. 5.5. Стек разбора и стек значений

На рис. 5.5 **$** слева представляет нижнюю часть двух стеков, которые растут вправо, когда значения вставляются в стек. Правая сторона диаграммы изображает последовательность терминальных символов, токены которых получаются в результате лексического анализа. Токены обрабатываются слева направо, при этом **$** справа представляет собой конец файла, также изображаемый как **EOF**. Многоточия (…) слева представляют собой пространство в двух стеках для обработки дополнительных операций вставки во время парсинга, в то время как многоточия с правой стороны представляют собой дополнительные входные символы, оставшиеся после изображенных на рисунке.

Тип `parserVal` был кратко упомянут в главе 4 «*Парсинг*». Для построения деревьев синтаксиса в BYACC/J мы должны остановиться на этом подробнее. Вот как тип `parserVal` определен в BYACC/J:

```
public class parserVal {
  public int ival;
  public double dval;
  public String sval;
  public Object obj;
  public parserVal() { }
  public parserVal(int val){ ival=val; }
  public parserVal(double val) { dval=val; }
  public parserVal(String val) { sval=val; }
  public parserVal(Object val) { obj=val; }
```

`parserVal` – это контейнер, который содержит *int*, *double*, *String* и *Object*, который может быть ссылкой на любой экземпляр класса вообще. Наличие здесь четырех полей является пустой тратой памяти, поскольку мы будем использовать только поле `obj`, но yacc – универсальный инструмент. В любом случае, давайте рассмотрим, как обернуть листья дерева в объект `parserVal`, чтобы поместить их в `yylval`.

Обертка листьев для стека значений парсера

С точки зрения механики, `parserVal` является третьим типом данных в коде, который строит наше дерево синтаксиса. BYACC/J требует, чтобы мы использовали этот тип для лексического анализатора с целью передачи токенов парсеру. По этой причине для реализации на Java класс этой главы `j0` имеет метод `scan()`, который выглядит следующим образом:

```
public static int scan(int cat) {
     ch5.j0.par.yylval =
             new parserVal(
                     new tree("token",0,
                             new token(cat, yytext(), yylineno)));
     return cat;
}
```

В Java каждый вызов `scan()` выделяет три объекта, как показано на рис. 5.6. В Unicon `scan()` выделяет два объекта, как показано на рис. 5.4.

Рис. 5.6. Три выделенных объекта – parserVal, лист и токен

Мы обернули токены в узлы дерева, чтобы представить информацию о листьях, и затем для Java мы обернули узлы листьев в `parserVal`, чтобы поместить их в стек значений. Давайте рассмотрим, как выглядит помещение листа в стек значений в замедленном режиме. Мы расскажем, как это происходит в Java, учитывая, что в Unicon все немного проще. Предположим, что вы находитесь в начале разбора и ваш первый токен – это зарезервированное слово *PUBLIC*. Сценарий показан на рис. 5.7. См. описание рис. 5.5, если вам нужно освежить в памяти, как организован этот рисунок.

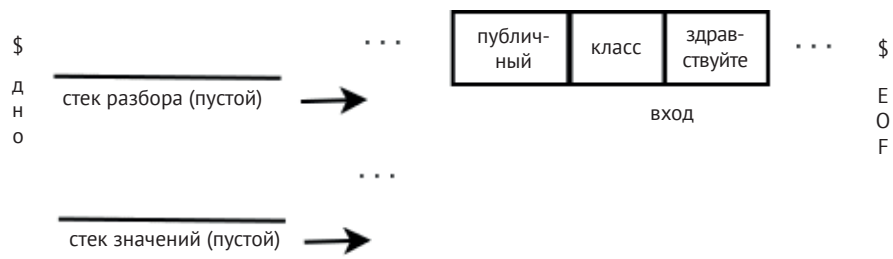


Рис. 5.7. Состояние стека разбора в начале парсинга

Первая операция – это сдвиг. Целочисленное состояние конечного автомата, которое кодирует тот факт, что мы видели *PUBLIC*, вставляется в стек. `yylex()` вызывает `scan()`, который выделяет лист, обернутый в экземпляр `parserVal`, и присваивает `yylval` ссылку на него, которую `yylex()` помещает в стек значений. Стеки находятся в замкнутом состоянии, как показано на рис. 5.8.

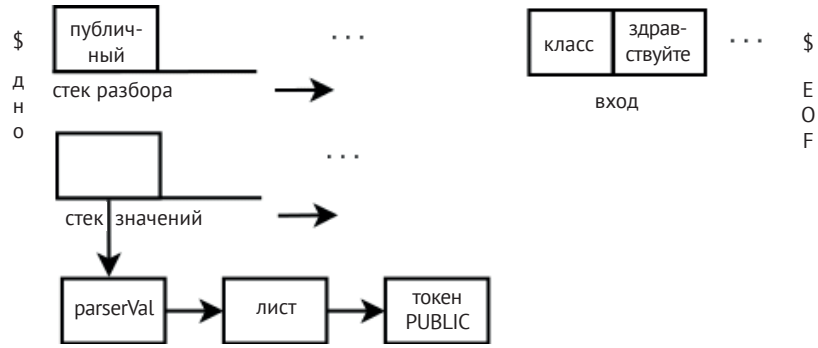


Рис. 5.8. Состояние стеков разбора и значений после операции сдвига

Следующий завернутый лист добавляется в стек значений каждый раз, когда происходит сдвиг. Теперь пришло время рассмотреть, как все эти листья помещаются во внутренние узлы и как внутренние узлы собираются в узлы более высокого уровня, пока вы не вернетесь к корню. Все это происходит по одному узлу за раз при соответствии правилу производства в грамматике.

Определение нужных вам листьев

В большинстве языков такие знаки препинания, как точка с запятой и круглые скобки, необходимы только для анализа синтаксиса. Возможно, они способствуют удобочитаемости, или устанавливают приоритет операторов, или делают грамматический разбор однозначным. Как только вы успешно разберете входные данные, вам больше никогда не понадобятся эти листья в дереве синтаксиса для последующего семантического анализа или генерации кода.

Вы можете опустить ненужные листья из дерева или оставить их, чтобы их информация о номере исходной строки и имени файла была в дереве на случай, если она понадобится для сообщения об ошибке. Я обычно опускаю их по умолчанию, но добавляю определенные листья со знаками препинания, если определяю, что они необходимы по какой-то причине.

Обратная сторона этого утверждения такова: любой лист, который содержит значение или имя, или другое семантическое значение какого-либо вида в языке, должен быть сохранен в дереве синтаксиса. Сюда входят литеральные константы, идентификаторы и другие зарезервированные слова или операторы. Теперь давайте рассмотрим, как и когда создавать внутренние узлы для вашего дерева синтаксиса.

Построение внутренних узлов из правил производства

В этом разделе мы научимся строить дерево по одному узлу за раз во время парсинга. Внутренние узлы вашего дерева синтаксиса, вплоть до корня, строятся снизу вверх, следуя последовательности операций сокращения, с помощью которых производственные правила распознаются во время разбора. Доступ к узлам дерева, используемым во время построения, осуществляется из стека значений.

Доступ к узлам дерева в стеке значений

Для каждого правила производства в грамматике есть возможность выполнить некоторый код, называемый **семантическим действием**, когда это правило производства используется во время разбора. Как вы видели в разделе «*Составление раздела бесконтекстной грамматики yacc*» главы 4 «*Парсинг*», код семантического действия появляется в конце грамматического правила, перед точкой с запятой или вертикальной чертой, которая заканчивает правило и начинает следующее.

Вы можете поместить в семантическое действие любой код, какой захотите. Для нас основной целью семантического действия является построение узла дерева синтаксиса. Используйте записи стека значений, соответствующие правой части правила производства, для построения узла дерева для символа, содержащегося в левой части правила производства. Нетерминал левой стороны, который был сопоставлен, получает новую запись в стек значений, которая может содержать только что построенный узел дерева.

Для этой цели *yacc* предоставляет макросы, которые ссылаются на каждую позицию в стеке значений во время операции сокращения. *$1, $2, ... $N* означают текущее содержимое стека значений, соответствующее правым символам грамматического правила от 1 до *N*. К моменту выполнения кода семантического действия эти символы уже были сопоставлены в какой-то момент в недавнем прошлом. Они являются верхними *N* символами в стеке значений, и во время операции сокращения они будут извлечены, а на их место будет вставлена новая запись стека значений. Новой записью стека значений будет то, что вы присвоите $$. По умолчанию это будет просто то, что находится в *$1*. Семантическое действие yacc по умолчанию – *$$=$1*, и это семантическое действие верно для правил производства с одним символом (терминалом или нетерминалом), который сокращается до нетерминала в левой части правила.

Все это можно долго распаковывать. Вот конкретный пример. Предположим, вы только что закончили разбор входного файла *hello.java*, представленного ранее, и дошли до того момента, когда пришло время сократить зарезервированные слова *PUBLIC*, *CLASS*, имя класса и его тело. Грамматическое правило, которое применяется в этот момент: *ClassDecl: PUBLIC CLASS IDENTIFIER ClassBody*.

Приведенное выше правило имеет четыре символа в правой части. Первые три являются терминальными, что означает, что в стеке значений их узлы дерева будут листьями. Четвертый символ в правой части – нетерминал, чья

запись в стеке значений будет внутренним узлом, поддеревом, которое в данном случае имеет три дочерних элемента. Когда приходит время сократить все это до правила производства *ClassDecl*, мы выделим новый внутренний узел. Поскольку мы заканчиваем парсинг, в данном случае это будет корень, но в любом случае он будет соответствовать найденному нами объявлению класса, и у него будет четыре дочерних узла. На рис. 5.9 представлено содержимое стека разбора и стека значений во время операции сокращения, когда весь класс, наконец, будет соединен в одно большое дерево.

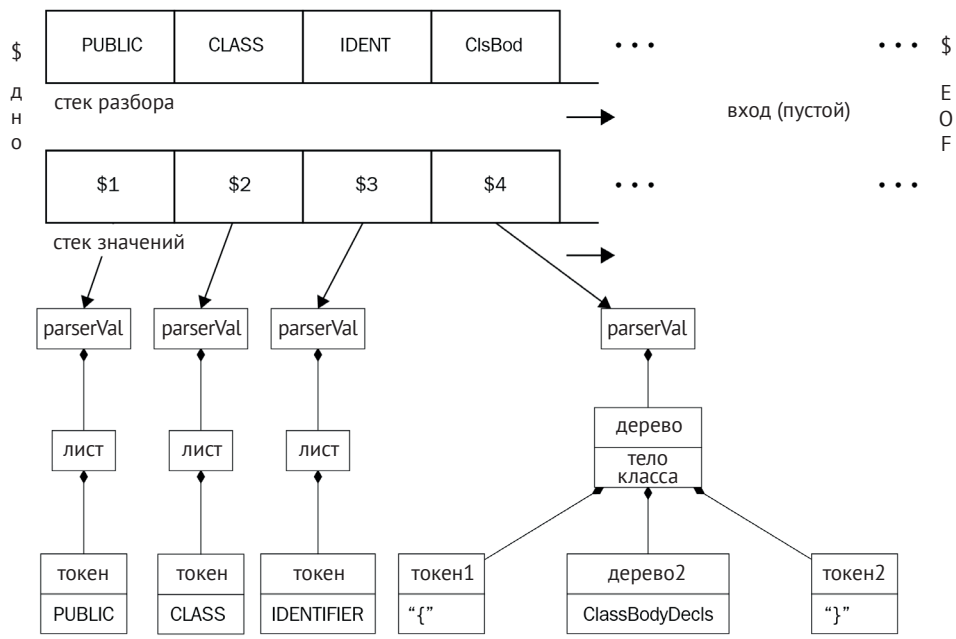


Рис. 5.9. Стек разбора и стек значений непосредственно перед операцией сокращения

Задача семантического действия для правила производства *ClassDecl* будет заключаться в создании нового узла, инициализации его четырех дочерних узлов *$1*, *$2*, *$3* и *$4* и присвоении ему значения *$$*. На рис. 5.10 показано, как это выглядит после создания правила *ClassDecl*.

Все дерево строится очень постепенно, по одному узлу за раз, и объекты `parserVal` удаляются в тот момент, когда дочерние объекты удаляются из стека значений и вставляются в родительский узел.

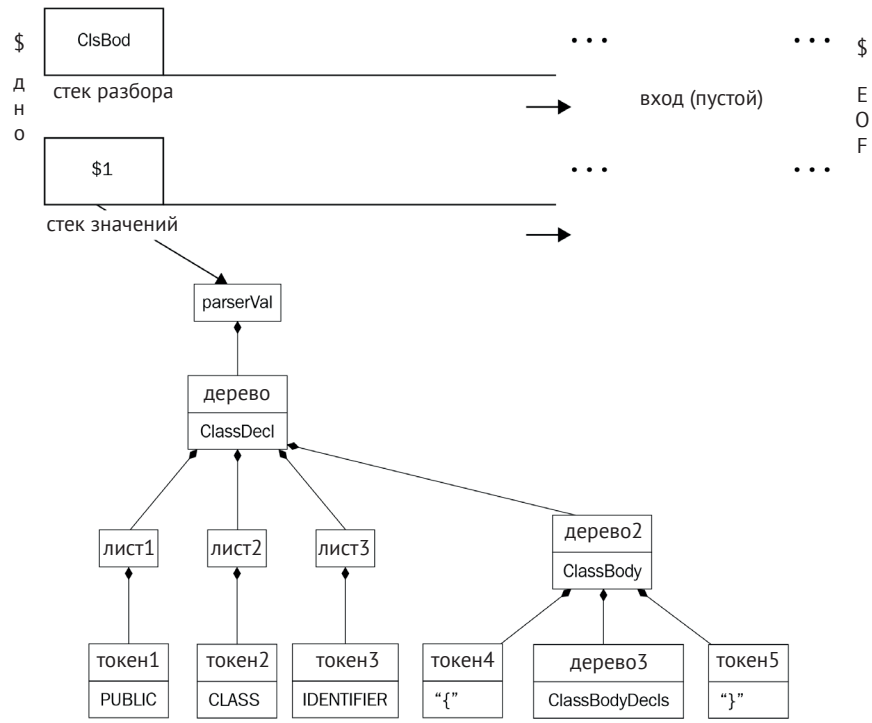


Рис. 5.10. Поддеревья объединяются в стеке значений во время операций сокращения

Использование фабричного метода узла дерева

Класс tree содержит важный фабричный метод под названием `node()`. Фабричный метод – это метод, который выделяет и возвращает объект. Он похож на конструктор, но выделяет объект другого типа, чем тот, в классе которого он находится. Фабричные методы используются в некоторых шаблонах проектирования. В нашем случае метод `node()` принимает метку, номер правила производства и любое количество дочерних элементов и возвращает внутренний узел, документирующий, что правило производства было сопоставлено. Код Unicon для метода `node()` показан далее:

```
method node(s,r,p[])
  return tree ! ([s,r] ||| p)
end
```

Код Java для метода `node()` является более сложным из-за обертывания и разворачивания типов `parserVal`. Завернуть только что созданный внутренний узел в объект `parserVal` достаточно легко с помощью вызова для создания нового объекта `parserVal`, но чтобы построить дочерние узлы дерева, они сначала разворачиваются с помощью отдельного вспомогательного метода под названием `unwrap()`. Код показан в следующем фрагменте:

```
    public static parserVal node(String s,
int
    r,parserVal...p) {
     tree[] t = new tree[p.length];
     for(int i = 0; i < t.length; i++)
     t[i] = (tree)(p[i].obj);
     return new parserVal((Object)new tree(s,r,t));
  }
```

Этот код Java принимает переменное количество аргументов, разворачивает их и передает в конструктор класса `tree`. Разворачивание заключается в выборе поля `obj` объекта `parserVal` и приведении его к типу `tree`.

Поскольку семантические действия для *iyacc* – это код Unicon, а для *BYACC/J* – код Java, это требует некоторого обмана. Семантическое действие будет законным как в Java, так и в Unicon только в том случае, если вы ограничите его общим синтаксисом, таким как вызов метода. Если вы начнете вставлять другие вещи в семантические действия, такие как операторы `if` и другой синтаксис, специфичный для конкретного языка, то ваша спецификация *yacc* станет специфичной только для одного языка, например Unicon или Java.

Однако в примерах из этой книги было не совсем возможно использовать один и тот же входной файл для iyacc и BYACC/J. Причина в том, что семантические действия в *yacc* обычно присваивают значение (ссылку на узел дерева разбора) специальной переменной с именем `$$`, а Unicon использует оператор `:=` для присваивания, в то время как Java использует `=`. Это было решено при написании данной книги путем модификации *iyacc* таким образом, что семантические действия, которые начинаются с `$$=`, воспринимаются как специальный оператор, который генерирует эквивалентное Unicon присваивание `$$:=`.

Стратегии, необходимые для построения внутренних узлов дерева синтаксиса, довольно просты – для каждого правила производства подсчитайте, сколько дочерних элементов является одним из следующих:

- нетерминалом;
- терминалом, не являющимся знаком препинания.

Если число таких элементов больше одного, вызовите фабричный метод `node()`, чтобы выделить узел дерева и назначить его в качестве элемента стека значений для правила производства. Теперь пришло время продемонстрировать построение дерева синтаксиса на нетривиальном примере – языке Jzero.

Формирование деревьев синтаксиса для языка Jzero

В этом разделе показано, как строить деревья синтаксиса для языка Jzero. Полный файл *j0gram.y* для этой главы доступен на сайте GitHub книги. Заголовок здесь опущен, поскольку объявления *%token* остались неизменными по сравнению с тем, как они представлены в разделе «*Спецификация yacc в Jzero*» предыдущей главы. Хотя мы снова представляем многие из правил грамматики, показанных в предыдущей главе, сейчас основное внимание уделяется построению новых узлов дерева, связанных с каждым правилом производства, если таковые имеются.

Как было описано ранее, внутренние узлы дерева строятся в семантических действиях, которые добавляются в конце правил производства. Для каждого правила производства, которое строит новый узел, узлу присваивается $$, значение yacc, соответствующее новому нетерминальному символу, построенному этим правилом производства.

Начальный нетерминал, который в случае Jzero представляет собой объявление класса, является точкой, от которой строится корень всего дерева. Его семантическое действие имеет дополнительную работу после присвоения построенному узлу значения $$. На этом верхнем уровне в этой главе код распечатывает дерево, вызывая метод `print()`, чтобы вы могли проверить, правильно ли оно построено. В последующих главах самый верхний узел дерева может быть присвоен глобальной переменной с именем `root` для последующей обработки или вызвать другой метод, чтобы перевести дерево в машинный код, либо для непосредственного выполнения программы путем интерпретации операторов в дереве.

Код показан в следующем фрагменте:

```
%%
ClassDecl: PUBLIC CLASS IDENTIFIER ClassBody {
   $$=j0.node("ClassDecl",1000,$3,$4);
   j0.print($$);
} ;
```

Нетерминал `ClassBody` либо содержит объявления (первое правило производства), либо пуст. В пустом случае возникает интересный вопрос, назначать ли явный листовой узел, указывающий на пустой `ClassBody`, как это сделано в следующем фрагменте кода, или просто сказать `$$=null`.

```
ClassBody: '{' ClassBodyDecls '}' {
              $$=j0.node("ClassBody",1010,$2); }
              | '{' '}' { $$=j0.node("ClassBody",1011); };
```

Нетерминал `ClassBodyDecls` объединяет в цепочку все поля, методы и конструкторы класса. Первое правило производства завершает рекурсии во втором правиле производства с помощью одного `ClassBodyDecl`. Поскольку в первом правиле производства нет семантического действия, оно выполняет `$$=$1`, поддерево для `ClassBodyDecl` продвигается вместо создания узла для родителя. Код показан в следующем фрагменте:

```
ClassBodyDecls: ClassBodyDecl
              | ClassBodyDecls ClassBodyDecl {
              $$=j0.node("ClassBodyDecls",1020,$1,$2); };
```

Существует три вида `ClassBodyDecl` на выбор. На этом уровне не выделяется дополнительный узел дерева, поскольку можно определить, к какому типу ClassBodyDecl относится каждое поддерево. Код представлен ниже:

```
ClassBodyDecl: FieldDecl | MethodDecl | ConstructorDecl ;
```

Поле, или переменная-член, объявляется с базовым типом, за которым следует список объявлений переменных, как показано в следующем фрагменте кода:

```
FieldDecl: Type VarDecls ';' {
             $$=j0.node("FieldDecl",1030,$1,$2); };
```

Типы в Jzero очень просты и включают четыре встроенных имени типов и общее правило для имен классов, как показано в следующем фрагменте кода. Ни одно правило производства не имеет двух дочерних элементов, поэтому на этом уровне нет необходимости в новых внутренних узлах. Возможно, строка (`String`) может быть обработана с помощью этого последнего правила и не должна быть особым случаем:

```
Type: INT | DOUBLE | BOOL | STRING | Name ;
```

Имя (`Name`) – это либо одиночный токен с именем `IDENTIFIER`, либо имя с одной или несколькими точками в нем, называемое `QualifiedName`, как показано в следующем фрагменте кода:

```
Name: IDENTIFIER | QualifiedName ;
QualifiedName: Name '.' IDENTIFIER {
             $$=j0.node("QualifiedName",1040,$1,$3);};
```

Объявления переменных представляют собой список из одного или нескольких деклараторов переменных (variable declarators), разделенных запятыми. В Jzero `VarDeclarator` – это просто `IDENTIFIER`, если за ним не следуют квадратные скобки, которые обозначают тип массива. Поскольку внутренний узел `VarDeclarator` подразумевает набор квадратных скобок, они не представлены в дереве в явном виде. Код показан в следующем фрагменте:

```
VarDecls: VarDeclarator | VarDecls ',' VarDeclarator {
             $$=j0.node("VarDecls",1050,$1,$3); };
VarDeclarator: IDENTIFIER | VarDeclarator '[' ']' {
             $$=j0.node("VarDeclarator",1060,$1); };
```

В Jzero метод может возвращать значение некоторого возвращаемого типа или `VOID`, как показано в следующем фрагменте кода:

```
MethodReturnVal : Type | VOID ;
```

Метод объявляется путем предоставления заголовка метода, за которым следует блок кода. Все методы являются публичными статическими методами. После возвращаемого значения заголовок метода, состоящий из имени и параметров метода, является MethodDeclarator, как показано в следующем фрагменте кода:

```
MethodDecl: MethodHeader Block {
             $$=j0.node("MethodDecl",1380,$1,$2); };
```

```
MethodHeader: PUBLIC STATIC MethodReturnVal
              MethodDeclarator {
                 $$=j0.node("MethodHeader",1070,$3,$4); };
MethodDeclarator: IDENTIFIER '(' FormalParmListOpt ')' {
              $$=j0.node("MethodDeclarator",1080,$1,$3); };
```

Необязательный список формальных параметров – это либо непустой FormalParmList, либо пустое правило производства, так называемое **правило эпсилона**, между вертикальной чертой и точкой с запятой. Список формальных параметров – это список формальных параметров, разделенных запятыми. Это непустой список, и рекурсия завершается единственным формальным параметром. Каждый формальный параметр имеет тип, за которым следует имя переменной, возможно, с квадратными скобками для типов массивов, как показано в следующем фрагменте кода:

```
FormalParmListOpt: FormalParmList | ;
FormalParmList: FormalParm | FormalParmList ',' FormalParm {
              $$=j0.node("FormalParmList",1090,$1,$3); };
FormalParm: Type VarDeclarator {
              $$=j0.node("FormalParm",1100,$1,$2); };
```

Конструкторы объявляются так же, как и методы, хотя у них нет возвращаемого типа, как показано в следующем фрагменте кода:

```
ConstructorDecl: MethodDeclarator Block {
              $$=j0.node("ConstructorDecl",1110,$1,$2); };
```

Блок – это последовательность из нуля или более высказываний. Хотя многие узлы дерева представляют собой разветвление двух или более дочерних узлов, несколько узлов дерева имеют только один дочерний узел, поэтому окружающая пунктуация не нужна. Такие узлы сами по себе могут быть ненужными, но они также могут облегчить понимание и обработку дерева. Вы можете увидеть пример в следующем фрагменте кода:

```
Block: '{' BlockStmtsOpt '}'{$$=j0.node("Block",1200,$2);};
BlockStmtsOpt: BlockStmts | ;
BlockStmts: BlockStmt | BlockStmts BlockStmt {
              $$=j0.node("BlockStmts",1130,$1,$2); };
BlockStmt: LocalVarDeclStmt | Stmt ;
```

Высказывания блока могут быть как объявлениями локальных переменных, так и высказываниями. Синтаксис `LocalVarDeclStmt` неотличим от правила `FieldDecl`. На самом деле, возможно, лучше устранить дублирование по умолчанию. Используете ли вы другой набор идентичных правил производства или факторизуете общие элементы грамматики, может зависеть от того, будет ли вам легче написать код, который делает правильные вещи с различными деревьями, если они имеют узнаваемо разные метки узлов дерева и номера правил производства, или будут ли эти различия распознаны и обработаны должным образом благодаря окружающему контексту дерева. Пример приведен в следующем фрагменте кода:

```
LocalVarDeclStmt: LocalVarDecl ';' ;
LocalVarDecl: Type VarDecls {
             $$=j0.node("LocalVarDecl",1140,$1,$2); };
```

В данном случае создается узел `LocalVarDecl`, что позволяет легко отличить локальные переменные от переменных-членов класса в дереве синтаксиса.

Многочисленные типы высказываний приводят к созданию собственных уникальных узлов дерева. Поскольку они являются правилами производства с одним дочерним элементом, введение еще одного узла дерева здесь излишне. Следующий фрагмент кода иллюстрирует это:

```
Stmt: Block | ';' | ExprStmt | BreakStmt | ReturnStmt |
      | IfThenStmt | IfThenElseStmt | IfThenElseIfStmt
      | WhileStmt | ForStmt ;
ExprStmt: StmtExpr ';' ;
StmtExpr: Assignment | MethodCall ;
```

В Jzero существует несколько нетерминалов, для того чтобы разрешить типичные вариации операторов *if*. Блоки требуются для тел условий и циклов в Jzero, для того чтобы избежать распространенной неоднозначности при их вложении, как показано в следующем фрагменте кода:

```
IfThenStmt: IF '(' Expr ')' Block {
     $$=j0.node("IfThenStmt",1150,$3,$5); };
IfThenElseStmt: IF '(' Expr ')' Block ELSE Block {
     $$=j0.node("IfThenElseStmt",1160,$3,$5,$7); };
IfThenElseIfStmt: IF '(' Expr ')' Block ElseIfSequence {
     $$=j0.node("IfThenElseIfStmt",1170,$3,$5,$6); }
| IF '(' Expr ')' Block ElseIfSequence ELSE Block {
     $$=j0.node("IfThenElseIfStmt",1171,$3,$5,$6,$8); };
ElseIfSequence: ElseIfStmt | ElseIfSequence ElseIfStmt {
     $$=j0.node("ElseIfSequence",1180,$1,$2); };
ElseIfStmt: ELSE IfThenStmt {
     $$=j0.node("ElseIfStmt",1190,$2); };
```

Узлы дерева обычно создаются для этих структур управления, и они обычно вводят ветвление в дерево. Хотя циклы *while* вводят только одно ответвление, узел для цикла *for* имеет четыре дочерних узла. Неужели разработчики языка сделали это специально? Вы можете увидеть пример в следующем фрагменте кода:

```
WhileStmt: WHILE '(' Expr ')' Stmt {
        $$=j0.node("WhileStmt",1210,$3,$5); };
ForStmt: FOR '(' ForInit ';' ExprOpt ';' ForUpdate ')'
Block {
        $$=j0.node("ForStmt",1220,$3,$5,$7,$9); };
ForInit: StmtExprList | LocalVarDecl | ;
ExprOpt: Expr | ;
ForUpdate: StmtExprList | ;
StmtExprList: StmtExpr | StmtExprList ',' StmtExpr {
        $$=j0.node("StmtExprList",1230,$1,$3); };
```

Оператор *break* адекватно представлен листом с надписью BREAK, как показано здесь:

```
BreakStmt: BREAK ';' ;
ReturnStmt: RETURN ExprOpt ';' {
        $$=j0.node("ReturnStmt",1250,$2); };
```

Оператор return требует нового узла, поскольку за ним следует необязательное выражение. Первичные выражения, включая литералы, не вводят дополнительный слой узлов дерева над содержимым их дочерних элементов. Единственное интересное действие здесь – для выражений с круглыми скобками, которое отбрасывает круглые скобки, использовавшиеся для приоритета операторов, и продвигает второго потомка без необходимости дополнительного узла дерева на данном уровне. Вот пример этого:

```
Primary: Literal | FieldAccess | MethodCall |
        '(' Expr ')' { $$=$2; };
Literal: INTLIT | DOUBLELIT | BOOLLIT | STRINGLIT | NULLVAL ;
```

Список аргументов – это одно или несколько выражений, разделенных запятыми. Чтобы разрешить ноль выражений, используется отдельный нетерминал, как показано в следующем фрагменте кода:

```
ArgList: Expr | ArgList ',' Expr {
        $$=j0.node("ArgList",1270,$1,$3); };
ArgListOpt: ArgList | ;
```

Доступы к полям могут быть соединены в цепочку, поскольку их левый дочерний элемент, Primary, может быть доступом к другому полю. Когда один нетерминал имеет правило производства, которое порождает другой нетерминал, который имеет правило производства, производящее первый нетерминал, такая ситуация называется **взаимной рекурсией** и является нормальной и здоровой. Вы можете увидеть пример этого в следующем фрагменте кода:

```
FieldAccess: Primary '.' IDENTIFIER {
        $$=j0.node("FieldAccess",1280,$1,$3); };
```

Вызов метода имеет синтаксис, состоящий из метода, за которым следует заключенный в круглые скобки список из нуля или более аргументов. Обычно это простой бинарный узел, в котором левый дочерний элемент довольно прост (имя метода), а правый дочерний элемент может содержать большое поддерево аргументов… или может быть пустым. Вот пример:

```
MethodCall: Name '(' ArgListOpt ')' {
                    $$=j0.node("MethodCall",1290,$1,$3); }
     | Primary '.' IDENTIFIER '(' ArgListOpt ')' {
            $$=j0.node("MethodCall",1291,$1,$3,$5); } ;
```

Как было показано в предыдущей главе, грамматика выражений в Jzero имеет множество рекурсивных уровней нетерминалов, которые не все показаны здесь. Вам следует обратиться к сайту книги, чтобы увидеть полную грамматику с описанием дерева синтаксиса. В следующем фрагменте кода каждый оператор вводит узел дерева. После того как дерево построено, простое прохождение по нему обеспечит корректное вычисление (или генерацию корректного кода) выражения:

```
PostFixExpr: Primary | Name ;
UnaryExpr: '-' UnaryExpr {$$=j0.node(«UnaryExpr»,1300,$1,$2);}
      | '!' UnaryExpr { $$=j0.node("UnaryExpr",1301,$1,$2); }
      | PostFixExpr ;
MulExpr: UnaryExpr
      | MulExpr '*' UnaryExpr {
             $$=j0.node("MulExpr",1310,$1,$3); }
      | MulExpr '/' UnaryExpr {
             $$=j0.node("MulExpr",1311,$1,$3); }
      | MulExpr '%' UnaryExpr {
             $$=j0.node("MulExpr",1312,$1,$3); };
AddExpr: MulExpr
      | AddExpr '+' MulExpr{$$=j0.node("AddExpr",1320,$1,$3); }
      | AddExpr '-' MulExpr{$$=j0.node("AddExpr",1321,$1,$3);
};
```

В классической грамматике языка C **операторы сравнения**, также называемые **операторами отношения**, являются просто еще одним уровнем приоритета для целочисленных выражений. Java и Jzero немного более интересны тем, что в них тип Boolean отделен от целочисленных типов, но об этом будет рассказано в последующих главах, посвященных семантическому анализу и проверке типов. Для кода, показанного в следующем фрагменте, существует четыре оператора отношения. `LESSTHANOREQUAL` – это целочисленный код, который лексический анализатор сообщает для <=, а `GREATERTHANOREQUAL` возвращается для >=. Для операторов < и > лексический анализатор возвращает их ASCII-коды:

```
RelOp: LESSTHANOREQUAL | GREATERTHANOREQUAL | '<' | '>' ;
```

Операторы отношения находятся на несколько более высоком уровне приоритета, чем сравнения того, равны или не равны значения друг другу:

```
RelExpr: AddExpr | RelExpr RelOp AddExpr {
        $$=j0.node("RelExpr",1330,$1,$2,$3); };
EqExpr: RelExpr
              | EqExpr ISEQUALTO RelExpr {
                     $$=j0.node("EqExpr",1340,$1,$3); }
              | EqExpr NOTEQUALTO RelExpr {
                     $$=j0.node("EqExpr",1341,$1,$3); };
```

Ниже операторов отношения и сравнения булевы операторы *&&* и || действуют на разных уровнях приоритета, как показано в следующем фрагменте кода:

```
CondAndExpr: EqExpr | CondAndExpr LOGICALAND EqExpr {
        $$=j0.node("CondAndExpr", 1350, $1, $3); };
CondOrExpr: CondAndExpr | CondOrExpr LOGICALOR CondAndExpr {
        $$=j0.node("CondOrExpr", 1360, $1, $3); };
```

Самым низким уровнем приоритета во многих языках, как и в Jzero, обладают операторы присваивания. В Jzero есть += и –=, но нет ++ и ––, которые считаются «банкой червей» для начинающих программистов и не имеют большой ценности для обучения построению компилятора. Примеры использования этих операторов приведены здесь:

```
Expr: CondOrExpr | Assignment ;
Assignment: LeftHandSide AssignOp Expr {
        $$=j0.node("Assignment",1370, $1, $2, $3); };
LeftHandSide: Name | FieldAccess ;
AssignOp: '=' | AUGINCR | AUGDECR ;
```

В этом разделе были представлены основные моменты построения дерева синтаксиса Jzero. Многие правила производства требуют построения нового внутреннего узла, который служит родителем нескольких дочерних узлов в правой части правила производства. Однако в грамматике имеется множество случаев, когда нетерминал строится только из одного символа в правой части, и в этом случае выделения дополнительного внутреннего узла обычно можно избежать. Теперь рассмотрим, как проверить ваше дерево, чтобы убедиться, что оно было собрано правильно.

Отладка и тестирование вашего дерева синтаксиса

Деревья, которые вы строите, должны быть прочными, как скала. Эта эффектная метафора означает следующее: если структура вашего дерева синтаксиса построена неправильно, вы не вправе рассчитывать на то, что сможете построить остальную часть вашего языка программирования. Самый прямой способ проверить, что дерево было построено правильно, – это вернуться назад и посмотреть на дерево, построенное вами. В этом разделе приведены два примера того, как это сделать. Сначала вы распечатаете свое дерево в человекочитаемом (более или менее) текстовом формате ASCII, а затем узнаете, как распечатать его в формате, который легко отображается графически с помощью популярного пакета с открытым исходным кодом Graphviz, доступ к которому обычно осуществляется через PlantUML или классический инструмент командной строки под названием *dot*. Сначала рассмотрим некоторые наиболее распространенные причины проблем в деревьях синтаксиса.

Предотвращение распространенных ошибок в дереве синтаксиса

Наиболее распространенные проблемы с деревьями синтаксиса приводят к сбоям в работе программы при распечатке дерева. Каждый узел дерева может содержать ссылки (указатели) на другие объекты, и когда эти ссылки непра-

вильно инициализируются – бах! Отладка проблем со ссылками сложна, даже в языках высокого уровня.

Первый важный момент заключается в следующем: создаются ли ваши листья и подхватываются ли они парсером? Предположим, у вас есть правило *lex*, подобное показанному здесь:

```
";"                          { return 59; }
```

Код ASCII является корректным. Разбор будет успешным, но ваше дерево синтаксиса будет нарушено. Вы должны создать лист и присвоить его *yylval* всякий раз, когда вы возвращаете целочисленный код в одном из ваших действий Flex. Если вы этого не сделаете, *yacc* будет иметь мусор, сидящий в *yylval*, когда *yyparse()* поместит его в стек значений для последующей вставки в ваше дерево. Вы должны убедиться, что каждое семантическое действие, которое возвращает целочисленный код в вашем файле *lex*, также выделяет новый лист и присваивает его *yylval*. Вы можете проверить каждый лист, чтобы убедиться, что он корректен на принимающей стороне, распечатав его содержимое при первом обращении к нему, как к правилу *$1* или *$2*, или к чему угодно, в семантических действиях для правил производства *yacc*.

Второй важный момент: правильно ли вы строите внутренние узлы для всех правил производства, которые имеют два или более дочерних значимых элементов (а не просто знаки препинания, например)? Если вы параноик, вы можете распечатать каждое поддерево, чтобы убедиться, что оно действительно, прежде чем создавать новое родительское, которое хранит указатели на дочерние поддеревья. Затем вы можете распечатать созданное родительское дерево, включая его дочерние, чтобы убедиться, что оно собрано корректно.

Один странный особый случай, который возникает при построении дерева синтаксиса, связан с правилами эпсилона – правилами производства, в которых нетерминал строится из пустой правой части. Примером может служить следующее правило из файла *j0gram.y*:

```
FormalParmListOpt: FormalParmList | ;
```

Для второго правила производства в этом примере нет дочерних элементов. Правило по умолчанию в yacc, `$$=$1`, выглядит не очень хорошо, поскольку не существует правила `$1`. Здесь можно построить новый лист, как в следующем решении:

```
FormalParmListOpt: FormalParmList | { $$=
                     j0.node("FormalParamListOpt",1095); }
```

Но этот лист отличается от обычного тем, что у него нет связанного токена. Коду, который впоследствии обходит дерево, лучше не предполагать, что все листья имеют токены. На практике некоторые могут просто использовать нулевой указатель для представления правила эпсилона. Если вы используете нулевой указатель, вам, возможно, придется добавить проверки на нулевые указатели везде в вашем последующем коде обхода дерева, включая распечат-

ки деревьев в следующих подразделах. Если вы выделяете лист для каждого правила эпсилона, ваше дерево станет больше, не добавляя никакой новой информации. Память дешева, поэтому если это упростит ваш код, то, вероятно, это можно сделать.

Подводя итог и в качестве последнего предупреждения: вы не сможете обнаружить фатальные ошибки в вашем коде построения дерева, если не напишете тестовые примеры, которые используют каждое правило производства в вашей грамматике! Такое покрытие грамматики может потребоваться в любом серьезном проекте по реализации языка. Итак, давайте рассмотрим реальные методы проверки корректности деревьев путем их распечатки.

Распечатка вашего дерева в текстовом формате

Один из способов проверить ваше дерево синтаксиса – распечатать структуру дерева в виде текста ASCII. Это делается с помощью обхода дерева, при котором каждый узел приводит к выводу одной или нескольких строк текста. Следующий метод `print()` в классе *j0* просто просит дерево распечатать себя:

```
method print(root)
      root.print()
end
```

Эквивалентный код на Java должен распаковать объект *parserVal* и преобразовать *Object* в дерево, чтобы попросить его напечатать себя, как показано в следующем фрагменте кода:

```
public static void print(parserVal root) {
      ((tree)root.obj).print();
}
```

Деревья обычно печатают себя рекурсивно. Лист просто печатает сам себя, а внутренний узел печатает себя, а затем просит свои дочерние узлы напечатать себя. При распечатке текста отступы используются для указания уровня вложенности или расстояния узла от корня. Значение уровня отступа передается как параметр и увеличивается для каждого более глубокого уровня в пределах дерева. Версия метода `print()` класса *tree* для Unicon показана в следующем фрагменте кода:

```
method print(level:0)
      writes(repl(" ",level))
      if \tok then
              write(id, " ", tok.text, " (",tok.cat,
                      "): ",tok.lineno)
      else write(id, " ", sym, " (", rule, "): ", nkids)
      every (!kids).print(level+1);
end
```

Этот метод отступает на заданное в параметре количество пробелов, а затем выводит строку текста, описывающую узел дерева. Затем он вызывает себя

рекурсивно, с уровнем вложенности больше на единицу, для каждого из дочерних узлов, если таковые имеются. Эквивалентный код Java для распечатки текста класса *tree* выглядит следующим образом:

```
public void print(int level) {
      int i;
      for(i=0;i<level;i++) System.out.print(" ");
      if (tok != null)
              System.out.println(id + " " + tok.text +
                                  " (" + tok.cat + "): "+tok.lineno);
      else
              System.out.println(id + " " + sym +
                      " (" + rule + "): "+nkids);
      for(i=0; i<nkids; i++)
              kids[i].print(level+1);
}
public void print() {
        print(0);
}
```

Когда вы выполняете команду *j0* с этой функцией печати дерева, она выдает следующие выходные данные:

```
63 ClassDecl (1000): 2
   6 hello (266): 1
   62 ClassBody (1010): 1
      59 MethodDecl (1380): 2
         32 MethodHeader (1070): 2
            14 void (264): 2
            31 MethodDeclarator (1080): 2
               16 main (266): 2
               30 FormalParm (1100): 2
                  20 String (266): 2
                  27 VarDeclarator (1060): 1
                     22 argv (266): 2
            58 Block (1200): 1
               53 MethodCall (1290): 2
                  46 QualifiedName (1040): 2
                     41 QualifiedName (1040): 2
                        36 System (266): 3
                        40 out (266): 3
                     45 println (266): 3
                  50 "hello, jzero!" (273): 3
no errors
```

Хотя древовидную структуру можно расшифровать, изучив этот вывод, она не совсем прозрачна. В следующем разделе показан графический способ изображения дерева.

Печать дерева с помощью dot

Забавный способ проверить ваше дерево синтаксиса – распечатать его в графическом виде. Как упоминалось в разделе «Технические требования», инструмент под названием *dot* может это сделать. Запись нашего дерева во входной формат *dot* осуществляется посредством другого обхода дерева, в котором каждый узел приводит к одной или нескольким строкам вывода текста. Чтобы нарисовать графическую версию дерева, измените метод *j0.print()*, дабы вызвать метод `print_graph()` класса *tree*. В Unicon это тривиально. Код показан в следующем фрагменте:

```
method print(root)
      root.print_graph(yyfilename || ".dot")
end
```

Эквивалентный код на Java должен распаковать объект `parserVal` и преобразовать *Object* в дерево, чтобы попросить его напечатать себя, как показано в следующем фрагменте:

```
public static void print(parserVal root) {
      ((tree)root.obj).print_graph(yyfilename + «.dot»);
}
```

Как и в случае печати текста, деревья печатают себя рекурсивно. Версия Unicon метода `print_graph()` класса *tree* показана в следующем фрагменте кода:

```
method print_graph(fw)
      if type(filename) == "string" then {
              fw := open(filename, "w") |
                      stop("can't open ", image(filename), " for writing")
              write(fw, "digraph {")
              print_graph(fw)
              write(fw, "}")
              close(fw)
      }
      else if \tok then print_leaf(fw)
      else {
              print_branch(fw)
              every i := 1 to nkids do
                      if \kids[i] then {
                              write(fw, "N",id," -> N",kids[i].id,";")
                              kids[i].print_graph(fw)
                      } else {
                              write(fw, "N",id," -> N",id,"_",j,";")
                              write(fw, "N", id, "_", j,
                                      " [label=\"Empty rule\"];")
                              j +:= 1
                      }
      }
end
```

Java-реализация функции `print_graph()` состоит из двух методов. Первый – это публичный метод, который принимает имя файла, открывает этот файл для записи и записывает весь граф в этот файл, как показано в следующем фрагменте кода:

```
void print_graph(String filename){
     try {
             PrintWriter pw = new PrintWriter(
                     new BufferedWriter(new FileWriter(filename)));
             pw.printf("digraph {\n");
             j = 0;
             print_graph(pw);
             pw.printf("}\n");
             pw.close();
             }
        catch (java.io.IOException ioException) {
             System.err.println("printgraph exception");
             System.exit(1);
             }
}
```

В Java перегрузка функций позволяет публичной и приватной частям `print_graph()` иметь одно и то же имя. Эти два метода отличаются различными параметрами. В публичной части `print_graph()` передает файл, который он открывает, в качестве параметра следующему методу. Эта версия `print_graph()` печатает одну или две строки о текущем узле и вызывает себя рекурсивно на каждом дочернем узле:

```
void print_graph(PrintWriter pw) {
int i;
     if (tok != null) {
             print_leaf(pw);
             return;
     }
     print_branch(pw);
     for(i=0; i<nkids; i++) {
             if (kids[i] != null) {
                     pw.printf("N%d -> N%d;\n", id, kids[i].id);
                     kids[i].print_graph(pw);
             } else {
                     pw.printf("N%d -> N%d%d;\n", id, kids[i].id, j);
                     pw.printf("N%d%d [label=\"%s\"];\n", id, j,
                             "Empty rule");
             j++;
             }
        }
}
```

Метод `print_graph()` вызывает пару вспомогательных функций – `print_leaf()` для листьев и `print_branch()` для внутренних узлов. Метод `print_leaf()`

печатает пунктирную рамку, содержащую характеристики терминального символа. Unicon-реализация `print_leaf()` показана здесь:

```
method print_leaf(pw)
      local s := parser.yyname[tok.cat]
      print_branch(pw)
      write(pw,"N",id,
              " [shape=box style=dotted label=\" ",s," \\n ")
      write(pw,"text = ",escape(tok.text)," \\l lineno = ",
                      tok.lineno," \\l\"];\n")
end
```

Целочисленный код терминального символа токена используется в качестве субскрипта в массиве строк в парсере с именем `yyname`. Это генерируется iyacc. Реализация `print_leaf()` на Java похожа на версию для Unicon, как показано в следующем фрагменте кода:

```
void print_leaf(PrintWriter pw) {
      String s = parser.yyname[tok.cat];
      print_branch(pw);
      pw.printf("N%d [shape=box style=dotted label=\" %s \\n",
              id, s);
      pw.printf("text = %s \\l lineno = %d \\l\"];\n",
                 escape(tok.text), tok.lineno);
}
```

Метод `print_branch()` выводит рамку для внутренних узлов, содержащую имя нетерминала, представленного этим узлом. Реализация `print_branch()` на Unicon показана здесь:

```
method print_branch(pw)
      write(pw, "N",id," [shape=box label=\"",
              pretty_print_name(),"\"];\n");
end
```

Реализация `print_branch()` в Java похожа на ее аналог в Unicon, как показано в следующем фрагменте кода:

```
void print_branch(PrintWriter pw) {
      pw.printf("N%d [shape=box label=\"%s\"];\n",
              id, pretty_print_name());
}
```

Метод `escape()` при необходимости добавляет escape-символы перед двойными кавычками, чтобы *dot* печатал двойные кавычки. Реализация метода `escape()` в Unicon состоит из следующего кода:

```
method escape(s)
      if s[1] == "\"" then
              return "\\" || s[1:-1] || "\\\""
      else return s
end
```

Реализация `escape()` в Java показана здесь:

```
public String escape(String s) {
      if (s.charAt(0) == '\"')
              return "\\"+s.substring(0, s.length()-1)+"\\\"";
      else return s;
}
```

Метод `pretty_print_name()` печатает наилучшее человекочитаемое имя для заданного узла. Для внутреннего узла это его строковая метка, а также порядковый номер, чтобы различать несколько повторений одной и той же метки. Для терминального символа оно включает лексему, которая была найдена. Код показан в следующем фрагменте:

```
method pretty_print_name() {
      if /tok then return sym || "#" || (rule%10)
      else return escape(tok.text) || ":" || tok.cat
end
```

Java-реализация `pretty_print_name()` выглядит аналогично предыдущему коду, как можно увидеть здесь:

```
public String pretty_print_name() {
      if (tok == null) return sym +"#"+(rule%10);
      else return escape(tok.text)+":"+tok.cat;
}
```

Запустите эту программу на примере входного файла `hello.java` с помощью следующей команды:

```
j0 hello.java                 java ch5.j0 hello.java
```

Программа *j0* выводит файл `hello.java.dot`, который является допустимым входом для программы *dot*. Для создания изображения PNG запустите программу *dot* с помощью следующей команды:

```
dot -Tpng hello.java.dot >hello.png
```

На рис. 5.11 показано дерево синтаксиса для `hello.java`, как оно записано в `hello.png`.

Если вы не напишете код построения дерева правильно, программа будет аварийно завершена в момент запуска или дерево будет явно ложным при просмотре изображения. Чтобы проверить ваш код на языке программирования,

вы должны запустить его на самых разных входных программах и внимательно изучить получившиеся деревья.

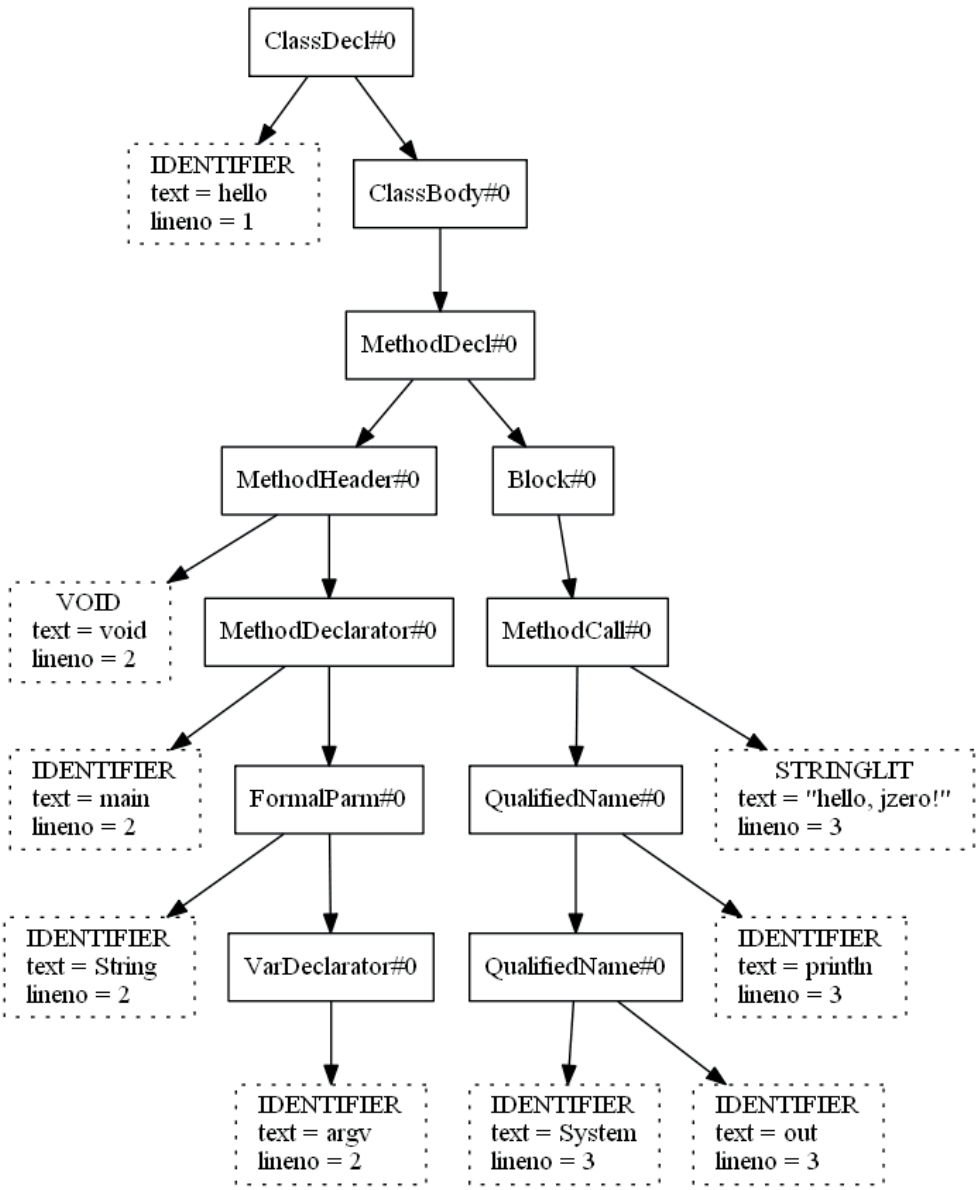


Рис. 5.11. Диаграмма дерева синтаксиса для hello.java

В этом разделе вы увидели, что всего несколько строк кода необходимы для создания текстовых и графических изображений ваших деревьев синтаксиса с помощью обходов деревьев. Графическое отображение обеспечивается внешним инструментом под названием *dot*. Обходы деревьев – это простая, но

мощная техника программирования, которая будет доминировать в следующих нескольких главах данной книги.

Заключение

В этой главе вы изучили важнейшие технические навыки и инструменты, используемые для построения дерева синтаксиса при разборе входной программы. Дерево синтаксиса – это основная структура данных, применяемая для внутреннего представления исходного кода компилятору или интерпретатору.

Вы узнали, как разработать код, идентифицирующий, какое правило производства было использовано для создания каждого внутреннего узла, чтобы в дальнейшем можно было определить, что именно мы рассматриваем. Вы узнали, как добавлять конструкторы узлов дерева для каждого правила в сканере. Вы узнали, как подключать листья из сканера в дерево, построенное в парсере. Вы узнали, как проверять свои деревья и решать общие проблемы их построения.

Вы закончили синтезировать исходный код в структуру данных, которую можно использовать. Теперь пришло время начать анализировать смысл исходного кода программы, чтобы вы могли определить, какие вычисления в нем указаны. Это делается путем прохождения по дереву разбора, используя обходы дерева для выполнения семантического анализа.

В следующей главе мы начнем путешествие по дереву, чтобы построить таблицы символов, которые позволят вам отследить все переменные в программе и выяснить, где они были объявлены.

Вопросы

1. Откуда берутся листья дерева синтаксиса?
2. Как создаются внутренние узлы дерева синтаксиса?
3. Где хранятся листья и внутренние узлы во время построения дерева?
4. Почему значения обертываются и разворачиваются, когда они вставляются и извлекаются из стека значений?

Часть II

Обходы дерева синтаксиса

Сердцем компилятора являются обходы деревьев. По завершении этой части вы получите компилятор, выполняющий семантический анализ и генерацию кода.

Эта часть включает в себя следующие главы:

- глава 6 «Таблицы символов»;
- глава 7 «Проверка базовых типов»;
- глава 8 «Проверка типов в массивах, вызовах методов и доступах к структурам»;
- глава 9 «Генерация промежуточного кода»;
- глава 10 «Раскрашивание синтаксиса в IDE».

Глава 6

Таблицы символов

Чтобы понять использование имен в исходном коде программы, ваш компилятор должен просмотреть каждое использование имени и определить, к чему относится это имя. Вы можете искать символы везде, где они используются, с помощью табличных структур данных, которые являются вспомогательными для дерева синтаксиса. Они называются **таблицами символов**. Выполнение операций по построению и последующему использованию таблицы символов – это первый шаг **семантического анализа**. Семантический анализ – это то, где вы изучаете значение входного исходного кода.

Бесконтекстные грамматики в главах синтаксиса этой книги имеют терминальные и нетерминальные символы, и они представлены в виде узлов дерева и структур токенов. Когда речь идет об исходном коде программы, слово «**символ**» используется по-другому. В этой и последующих главах символ означает имя переменной, функции, класса или пакета, например. В данной книге слова символ, имя, переменная и идентификатор используются как взаимозаменяемые.

В этой главе вы узнаете, как строить таблицы символов, вставлять в них символы и использовать таблицы символов для выявления двух видов семантических ошибок – необъявленных и незаконно переобъявленных переменных. В последующих главах вы будете использовать таблицы символов для проверки типов и генерирования кода входной программы.

Примеры в этой главе демонстрируют, как использовать таблицы символов, создавая их для подмножества Jzero языка Java. Таблицы символов важны для того, чтобы иметь возможность проверять типы и генерировать код для вашего языка программирования. В этой и нескольких следующих главах главным навыком, которому вы будете учиться, является искусство рекурсии путем написания множества избирательных и специализированных функций обхода деревьев.

В данной главе рассматриваются следующие основные темы:

- создание основы для таблиц символов;
- создание и заполнение таблиц символов для каждой области видимости;
- проверка наличия необъявленных переменных;
- поиск повторно объявленных переменных;
- управление областями действия классов – пример в Unicon.

Пришло время узнать о таблицах символов и о том, как их создавать. Однако сначала вам необходимо узнать о некоторых концептуальных основах, которые вы будете использовать для выполнения этой работы.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/.PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch6*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/3ccYTZv*.

Создание основы для таблиц символов

В разработке программного обеспечения вы должны пройти через анализ требований и проектирование, прежде чем приступить к кодированию. Точно так же, чтобы создать таблицы символов, необходимо понять, для чего они нужны и как написать обходы дерева синтаксиса, которые выполняют эту работу. Для начала вы должны проанализировать, какие виды информации должен хранить ваш компилятор, и вспомнить различные виды переменных. Информация в таблицах символов будет сохраняться из объявлений в программном коде, поэтому давайте рассмотрим их.

Объявления и области видимости

Смысл компьютерной программы сводится к смыслу вычисляемой информации и собственно вычислений, которые должны быть выполнены. Таблицы символов – это все, что касается первой части – определения того, какой информацией манипулирует программа. Мы начнем с определения используемых имен, того, на что они ссылаются и как используются.

Рассмотрим простой оператор присваивания, например следующий:

```
x = y + 5;
```

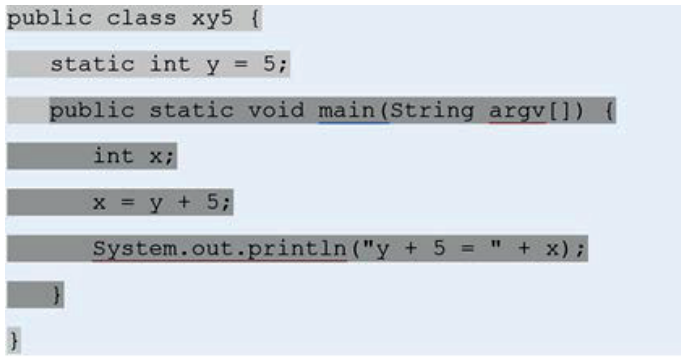
В большинстве языков такие имена, как *x* или *y*, должны быть объявлены до их использования. Объявление определяет имя, которое будет использоваться в программе, и обычно включает информацию о типе. Пример объявления для *x* может выглядеть следующим образом:

```
int x;
```

Каждое объявление переменной имеет **область видимости**, которая описывает область в программе, где эта переменная видима. В Jzero определяемые пользователем области видимости – это область видимости класса и локальная область видимости (метод). Jzero также должен поддерживать области видимости, связанные с несколькими предопределенными системными пакетами, что является небольшим подмножеством функциональности области видимости пакетов, требуемой для полноценного компилятора Java. Другие языки имеют дополнительные и различные виды областей видимости, с которыми приходится иметь дело.

Пример программы, показанный на рис. 6.1, который можно найти в файле *xy5.java* по адресу *https://github.com/PacktPublishing/Build-Your-Own-ProgrammingLanguage/tree/master/ch6*, расширяет предыдущий пример, иллюст-

рируя области видимости. Светло-серая область видимости класса окружает более темную серую локальную область видимости.



```
public class xy5 {
   static int y = 5;
   public static void main(String argv[]) {
      int x;
      x = y + 5;
      System.out.println("y + 5 = " + x);
   }
}
```

Рис. 6.1. Локальная область видимости, вложенная в область видимости класса

Один и тот же символ, например *x* или *y*, может быть объявлен в обеих областях видимости. Имя, объявленное во внутренней области видимости, перекрывает и скрывает то же имя, объявленное во внешней области видимости. Такая вложенная область видимости требует, чтобы язык программирования создавал несколько таблиц символов. Распространенной ошибкой новичков является попытка сделать весь язык с помощью только одной таблицы символов, потому что таблица символов кажется большой и страшной, а книги по компиляторам часто говорят о *таблице символов* вместо *таблиц символов*. Вы должны избегать этой ошибки. Планируйте поддержку нескольких таблиц символов и ищите символы, начиная с самой внутренней таблицы символов, затем переходите к окружающим таблицам. Теперь давайте подумаем о двух основных способах использования символов в программах для взаимодействия с памятью компьютера – присваивании и разыменовании.

Присваивание и разыменование переменных

Переменные – это имена для мест в памяти, а память можно читать или записывать. Запись значения в ячейку памяти называется **присваиванием**. Чтение значения из области памяти называется **разыменованием**. Большинство программистов имеют твердое понимание присваивания. Присваивание – одна из первых вещей, о которых они узнают в программировании, например высказывание `x=0` – это присваивание `x`. Многие программисты нечетко понимают, что такое разыменование. Программисты постоянно пишут код с разыменованием, но, возможно, они не слышали об этом термине раньше. Например, высказывание `y=x+1` разыменовывает `x`, чтобы получить его значение перед выполнением сложения. Аналогично передача параметра в вызове, таком как `System.out.println(x)`, разыменовывает `x`.

И присваивание, и разыменование – это действия, использующие адрес памяти. Они вступают в игру в семантическом анализе и генерации кода. Но при каких обстоятельствах присваивание и разыменование влияют на то, является ли использование переменной законным в данной ситуации? Присваивание не является законным для вещей, которые были объявлены как *const*,

включая имена методов. Существуют ли символы, которые нельзя разыменовывать? Конечно, необъявленные переменные, их также нельзя присваивать. Что-нибудь еще? Прежде чем мы сможем сгенерировать код для присваивания или разыменования, мы должны понять, какая область памяти используется, и является ли запрашиваемая операция законной и определенной в языке, который мы реализуем.

До сих пор мы рассматривали понятия присваивания и разыменования. Проверка законности каждого присваивания или разыменования требует хранения и извлечения информации об именах, используемых в программе, для чего и нужны таблицы символов. Вам нужна еще одна концептуальная основа, и тогда вы будете готовы к созданию своих таблиц символов. В этой и нескольких следующих главах вы будете выполнять множество функций обхода дерева синтаксиса. Давайте рассмотрим некоторые разновидности обхода дерева, имеющиеся в нашем распоряжении.

Выбор подходящего обхода дерева для работы

В предыдущей главе вы распечатывали деревья синтаксиса, используя обход деревьев, где работа на текущем узле была закончена, после чего рекурсивно вызывалась функция обхода на каждом дочернем узле. Это называется **прямым обходом**. Шаблон псевдокода для этого выглядит следующим образом:

```
method preorder()
      do_work_at_this_node()
      every child := !kids do child.preorder()
end
```

В некоторых примерах в этой главе вы посетите дочерние узлы и попросите их сначала выполнить свою работу, а затем использовать то, что они вычислили, для выполнения работы в текущем узле. Это называется **обратным обходом**. Шаблон псевдокода для обратного обхода выглядит следующим образом:

```
method postorder()
      every child := !kids do child.postorder()
      do_work_at_this_node()
end
```

Существуют и другие виды обхода, когда метод выполняет некоторую работу для текущего узла в промежутке между каждым вызовом дочернего узла, – это известно как **центрированный обход**. Наконец, часто можно написать обход дерева, состоящий из нескольких методов, которые работают вместе и вызывают друг друга по мере необходимости, возможно, по одному методу для каждого типа узла дерева. Хотя мы постараемся сделать наши обходы деревьев как можно более простыми, примеры в этой книге будут использовать лучший инструмент для этой работы.

В этом разделе вы узнали о нескольких важных понятиях, которые будут использоваться в примерах кода в этой и следующих главах. К ним относятся вложенные области видимости, присваивание и разыменование, а также различные виды обхода деревьев. Теперь пришло время использовать эти поня-

тия для создания таблиц символов. После этого вы можете рассмотреть, как заполнять таблицы символов, вставляя в них символы.

Создание и заполнение таблиц символов для каждой области видимости

Таблица символов содержит запись всех имен, которые объявлены для области видимости. Для каждой области видимости существует одна таблица символов. Таблица символов предоставляет возможность поиска символов по их имени, чтобы получить информацию о них. Если переменная была объявлена, то поиск по таблице символов возвращает запись со всей информацией, известной об этом символе: где он объявлен, каков его тип данных, является ли он *public* или *private* и так далее. Всю эту информацию можно найти в дереве синтаксиса. Если мы также поместим ее в таблицу, то цель состоит в том, чтобы получить доступ к информации напрямую для любого другого места, где эта информация необходима.

Традиционной реализацией таблицы символов является **хеш-таблица**, которая обеспечивает очень быстрый поиск информации. Ваш компилятор может использовать любую структуру данных, которая позволяет вам хранить или извлекать информацию, связанную с символом, даже связанный список[1]. Но хеш-таблицы лучше всего подходят для этого и являются стандартными в Unicon и Java. Поэтому в этой главе мы будем использовать хеш-таблицы.

Unicon предоставляет хеш-таблицы со встроенным типом данных, называемым **таблицей (table)**. Описание см. в *приложении «Основы Unicon»*. Вставка и поиск в таблице могут быть выполнены путем подписки, допустим доступ к элементам массива. Например, *symtable[sym]* ищет информацию, относящуюся к символу с именем *sym*, а *symtable[sym] := x* ассоциирует информацию об *x* с символом *sym*.

Java предоставляет хеш-таблицы в классах стандартных библиотек. Мы будем использовать класс библиотеки Java, известный как HashMap. Информация извлекается из *HashMap* с помощью вызова такого метода, как *symtable.get(sym)*, и сохраняется в HashMap с помощью *symtable.put(sym, x)*.

Как таблицы Unicon, так и Java HashMap отображают элементы из домена в связанный диапазон. В случае с таблицей символов домен будет содержать строковые имена символов в исходном коде программы. Для каждого символа в домене диапазон будет содержать соответствующий экземпляр класса *symtab_entry* – запись таблицы символов. В реализации Jzero, которую мы будем представлять, сами хеш-таблицы будут обернуты в класс, так что таблицы символов могут содержать дополнительную информацию обо всей области видимости, в дополнение к символам и записям таблицы символов.

Два главных вопроса: когда создаются таблицы символов для каждого диапазона и как именно в них вводится информация? Ответ на оба вопроса таков: во время **обхода дерева синтаксиса**. Но прежде чем мы перейдем к этому, вам необходимо узнать о семантических атрибутах.

---

[1] Связанный список – базовая динамическая структура данных в информатике, состоящая из узлов, каждый из которых содержит как собственно данные, так и одну или две ссылки на следующий и/или предыдущий узел списка. – *Прим. перев.*

Добавление семантических атрибутов к деревьям синтаксиса

Тип дерева в предыдущей главе был чистым и простым. Он содержал метку для печати, правило производства и несколько дочерних элементов. В реальной жизни язык программирования должен вычислять и хранить много дополнительной информации в различных узлах дерева. Эта информация хранится в дополнительных полях в узлах дерева, обычно называемых **семантическими атрибутами**. Значения этих полей иногда могут быть вычислены во время парсинга, когда мы строим узлы дерева. Чаще всего легче вычислить значения семантических атрибутов после построения всего дерева. В этом случае атрибуты строятся с помощью обхода дерева.

Существует два вида семантических атрибутов:

- синтезированные атрибуты – это атрибуты, значения которых для каждого узла могут быть построены из семантических атрибутов его дочерних узлов;
- унаследованные атрибуты – вычисляются с использованием информации, которая не поступает от дочерних узлов.

Единственный возможный путь для информации из других частей дерева – через родителя, поэтому атрибут называется наследуемым. На практике наследуемые атрибуты могут поступать от братьев и сестер либо издалека в дереве синтаксиса.

В этой главе мы добавим два атрибута к файлам *tree.icn* и *tree.java* из предыдущей главы. Первый атрибут, булев *isConst*, является синтезированным атрибутом, который сообщает о том, содержит ли данный узел дерева только постоянные значения, известные к моменту компиляции. На рис. 6.2 изображено выражение с именем *x+1*. *isConst* родительского узла (дополнение) вычисляется из значений *isConst* его дочерних узлов.

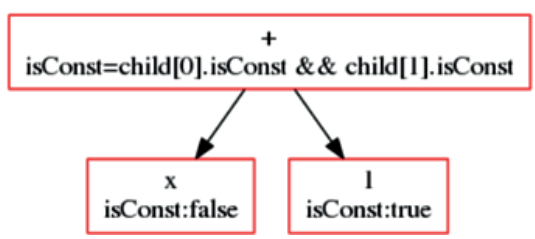


Рис. 6.2. Синтезированный атрибут вычисляет значение узла из своих дочерних узлов

На этом рисунке показан хороший пример вычисления синтезированного атрибута из его потомков. В этом примере листья для *x* и *1* уже имеют значения *isConst*, и эти значения должны откуда-то взяться. Легко догадаться, откуда берется значение *isConst* для токена *1*, – значения литеральных констант языка должны быть помечены как *isConst=true*.

Для такого имени, как *x*, не так очевидно, откуда берется значение *isConst*. Как было показано в предыдущих главах, язык Jzero не имеет ключевого слова Java *final*, которое обозначало бы неизменяемость данного символа. Ваши возможности заключаются в том, чтобы либо установить *isConst=false* для каждого

IDENTIFIER, либо расширить Jzero, чтобы разрешить ключевое слово *final*, по крайней мере для переменных. Если вы выберете последний вариант, то узнать, является *x* константой или нет, можно будет, просмотрев информацию об *x* в таблице символов. Только запись в таблице символов для *x* будет знать, является ли *x* константой, если мы поместим туда эту информацию.

Второй атрибут, *stab*, является наследуемым атрибутом, содержащим ссылку на таблицу символов для ближайшей охватывающей области видимости, содержащей данный узел дерева. Для большинства узлов значение *stab* просто копируется из его родителя. Узлы, в которых это не так, – это те узлы, в которых родитель определяет новую область видимости. На рис. 6.3 показано, как атрибут *stab* копируется из родительского узла в дочерний.

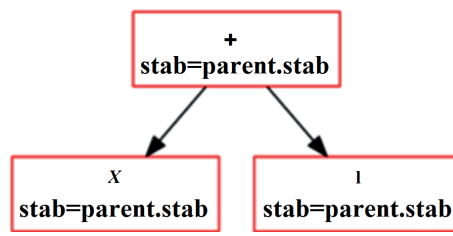


Рис. 6.3. Наследуемый атрибут вычисляет значение узла из информации родителя

Как получить атрибуты, перемещаемые к родителям от детей? Обходом деревьев. Как получить атрибуты, передаваемые детям от родителей? Обходом деревьев. Но сначала мы должны выделить место в узлах дерева для хранения этих атрибутов. Для включения этих атрибутов в данной главе заголовок класса дерева в Unicon был пересмотрен следующим образом:

```
class tree (id,sym,rule,nkids,tok,kids,isConst,stab)
```

Этот код ничего не делает, смысл в том, чтобы добавить два поля для семантических атрибутов в конце. В Java эти дополнения дерева классов приводят к следующему коду:

```
class tree {
      int id, rule, nkids;
      String sym;
      token tok;
      tree kids[];
      Boolean isConst;
      symtab stab;
}
```

К классу дерева в этой и последующих главах будет добавлено много методов, поскольку большинство аспектов семантического анализа и генерации кода для вашего языка будут представлены в виде обхода деревьев. Теперь давайте рассмотрим типы классов, необходимые для визуализации таблиц символов, и класс записи таблицы символов, который содержит информацию, хранящуюся в таблицах символов.

Определение классов для таблиц символов и записей в таблицах символов

Экземпляры класса *symtab* управляют символами для одной области видимости. Для каждой таблицы символов необходимо знать, с какой областью видимости она связана, а также какова охватывающая ее область видимости. Код Unicon для класса *symtab*, который можно найти в файле *symtab.icn*, выглядит следующим образом:

```
class symtab(scope, parent, t)
     method lookup(s)
            return \ (t[s])
     end
initially
        t := table()
end
```

Класс `symtab` – это почти обертка вокруг встроенного в Unicon табличного типа данных. В этом классе поле *scope* представляет собой строку, начинающуюся с «*class*» или «*local*» для объявленных пользователем областей видимости в Jzero. Важный метод этого класса *insert()* выдает семантическую ошибку, если символ уже находится в таблице. В противном случае *insert()* выделяет запись в таблице символов и вставляет ее. Метод *insert()* будет показан далее в этой главе, в разделе «*Поиск переопределенных переменных*». Соответствующий класс Java состоит из следующего кода в файле *symtab.java*:

```
package ch6;
import java.util.HashMap;
public class symtab {
     String scope;
     symtab parent;
     HashMap<String,symtab_entry> t;
     symtab(String sc, symtab p) {
            scope = sc; parent = p;
            t = new HashMap<String,symtab_entry>();
     }
     symtab_entry lookup(String s) {
            return t.get(s);
     }
}
```

Каждая таблица символов связывает имя с экземпляром класса `symtab_entry`. Класс `symtab_entry` будет хранить всю информацию, которую мы знаем о данной переменной. Реализацию `symtab_entry` в Unicon можно найти в файле *symtab_entry.icn*:

```
class symtab_entry(sym,parent_st,st,isConst)
end
```

Пока что класс `symtab_entry` не содержит никакого кода, он просто содержит несколько полей данных. Поле `sym` – это строка, содержащая символ, который

обозначает запись. Поле `parent_st` – это ссылка на окружающую таблицу символов. Поле `st` – это ссылка на новую таблицу символов, связанную с подобластью видимости этого символа, которая используется только для символов, имеющих подобласти видимости, такие как классы и методы. В следующих главах класс `symtab_entry` получит дополнительные поля как для семантического анализа, так и для целей генерации кода. Java-реализация `symtab_entry` в файле *symtab_entry.java* выглядит следующим образом:

```
package ch6;
public class symtab_entry {
 String sym;
 symtab parent_st, st;
 boolean isConst;
 symtab_entry(String s, symtab p, boolean iC) {
        sym = s; parent_st = p, isConst = iC; }
 symtab_entry(String s, symtab p, boolean iC, symtab t) {
        sym = s; parent_st = p; isConst = iC; st = t; }
}
```

Этот класс не содержит никакого кода, кроме двух конструкторов. Один из них предназначен для обычных переменных, а другой – для классов и методов. Классы и записи таблицы символов методов принимают в качестве параметра дочернюю таблицу символов, поскольку имеют подобласть видимости. Определив типы классов для таблиц символов и записей таблиц символов, пришло время рассмотреть, как создать таблицы символов для входной программы.

Создание таблиц символов

Вы можете создать таблицу символов для каждого класса и каждого метода, написав обход дерева. Каждый узел дерева синтаксиса должен знать, какой таблице символов он принадлежит. Представленный здесь подход грубой силы заключается в заполнении поля `stab` каждого узла дерева. Обычно это поле наследуется от родителя, но узлы, которые вводят новые области видимости, идут вперед и выделяют новую таблицу символов во время обхода. Следующий метод Unicon `mkSymTables()` создает таблицы символов. Он добавляется в класс *tree* в файле *tree.icn*:

```
method mkSymTables(curr)
      stab := curr
      case sym of {
              "ClassDecl": { curr := symtab("class",curr) }
              "MethodDecl": { curr := symtab("method",curr) }
      }
      every (!\kids).mkSymTables(curr)
end
```

Метод `mkSymTables()` принимает вложенную таблицу символов с именем `curr` как параметр. Соответствующий метод Java, `mkSymTables()` в файле *tree.java*, выглядит следующим образом:

```
void mkSymTables(symtab curr) {
      stab = curr;
      switch (sym) {
      case "ClassDecl": curr = new symtab("class", curr);
              break;
      case "MethodDecl": curr = new symtab("method", curr);
              break;
      }
      for (int i=0; i<nkids; i++) kid[i].mkSymTables(curr);
}
```

Корень всего дерева разбора начинается с глобальной таблицы символов с предопределенными символами, такими как *System* и *java*. В связи с этим возникает вопрос: когда и где вызывается `mkSymTables()`? Ответ: после построения корня дерева синтаксиса. Там, где в предыдущей главе вызывался `j0.print($$)`, теперь следует вызвать `j0.semantic($$)`, и весь семантический анализ будет выполняться в этом методе класса *j0*. Поэтому первое семантическое действие в *j0gram.y* становится следующим:

```
ClassDecl: PUBLIC CLASS IDENTIFIER ClassBody {
      $$=j0.node("ClassDecl",1000,$3,$4);
      j0.semantic($$);
} ;
```

Метод `semantic()` в *j0.icn* выглядит так:

```
method semantic(root)
local out_st, System_st
      global_st := symtab("global")
      out_st := symtab("class")
      System_st := symtab("class")
      out_st.insert("println", false)
      System_st.insert("out", false, out_st)
      global_st.insert("System", false, System_st)
      root.mkSymTables(global_st)
      root.populateSymTables()
      root.checkSymTables()
      global_st.print()
end
```

Этот код создает глобальную таблицу символов, а затем предопределяет символ для класса `System`. `System` имеет подобласть видимости, в которой объявлено имя `out` с подобластью видимости, в которой определено имя `println`. Соответствующий код Java для инициализации предопределенных символов выглядит следующим образом:

```
void semantic(tree root) {
symtab out_st, System_st;
      global_st = symtab("global");
      out_st = symtab("class");
```

```
        System_st = symtab("class");
        out_st.insert("println", false);
        System_st.insert("out", false, out_st);
        global_st.insert("System", false, System_st);
        root.mkSymTables(global_st);
        root.populateSymTables();
        root.checkSymTables();
        global_st.print();
}
```

Создание таблиц символов – это одно, а их использование – совсем другое. Давайте рассмотрим, как символы помещаются в таблицы символов. Затем мы можем начать говорить о том, как эти таблицы символов используются.

Заполнение таблиц символов

Заполнение (вставка символов) таблиц символов может быть выполнено во время того же обхода дерева, в котором создаются эти таблицы символов. Однако код более прост при отдельном обходе. Каждый узел знает, в какой таблице символов он находится. Задача состоит в том, чтобы определить, какие узлы вводят символы.

Для класса второй дочерний элемент FieldDecl содержит список символов для вставки. Первый дочерний элемент MethodDeclarator – это символ для вставки. Для метода второй потомок FormalParm вводит символ. Второй потомок LocalVarDecl имеет список символов для вставки. Эти действия показаны в следующем коде:

```
method populateSymTables()
      case sym of {
              "ClassDecl": {
                      stab.insert(kids[1].tok.text, , kids[1].stab)
                      }
              "FieldDecl" | "LocalVarDecl" : {
                      k := kids[2]
                      while \k & k.label=="VarDecls" do {
                              insert_vardeclarator(k.kids[2])
                              k := k.kids[1]
                              }
                      insert_vardeclarator(k); return
                      }
              "MethodDecl": {
                      stab.insert(kids[1].kids[2].kids[1].tok.text, ,
                                  kids[1].stab) }
              "FormalParm": { insert_vardeclarator(kids[2]); return }
      }
              every k := !\kids do k.populateSymTables()
end
```

Соответствующий код Java выглядит следующим образом:

```
void populateSymTables() {
      switch(sym) {
      case "ClassDecl": {
              stab.insert(kids[0].tok.text, false, kids[0].stab);
              break;
      }
      case "FieldDecl": case "LocalVarDecl": {
              tree k = kids[1];
              while ((k != null) && k.sym.equals("VarDecls")) {
                      insert_vardeclarator(k.kids[1]);
                      k = k.kids[0];
                      }
              insert_vardeclarator(k); return;
              }
      case "MethodDecl": {
              stab.insert(kids[0].kids[1].kids[0].tok.text, false,
                            kids[0].stab); }
      case "FormalParm": {
              insert_vardeclarator(kids[1]); return; }
              }
      for(int i = 0; i < nkids; i++) {
              tree k = kids[i];
              k.populateSymTables();
      }
}
```

Методу `insert_vardeclarator(n)` может быть передан один из двух вариантов – либо *IDENTIFIER*, содержащий символ, который нужно вставить, либо узел дерева *VarDeclarator*, который указывает на то, что объявляется массив. Реализация Unicon выглядит следующим образом:

```
method insert_vardeclarator(vd)
      if \vd.tok then stab.insert(vd.tok.text)
      else insert_vardeclarator(vd.kids[1])
end
```

Java-реализация этого кода выглядит так:

```
void insert_vardeclarator(tree vd) {
      if (vd.tok != null) stab.insert(vd.tok.text, false);
      else insert_vardeclarator(vd.kids[0]);
}
```

Заполнение таблиц символов необходимо для таких последующих аспектов реализации вашего языка программирования, как проверка типов и генерация кода. Они не смогут просто пропустить поддерево, пока не найдут IDENTIFIER. Даже такая формулировка уже хорошо подходит для проверки некоторых распространенных семантических ошибок, таких как необъявленные переменные. Теперь рассмотрим, как вычислить синтезированный атрибут – навык, который вы можете использовать как при заполнении таблиц символов информацией, так и в последующих частях семантического анализа и генерации кода.

Синтез атрибута isConst

`isConst` – это классический пример синтезированного атрибута. Правила его вычисления зависят от того, является узел листом (следуя базовому варианту) или внутренним узлом (используя шаг рекурсии):

- **базовый случай** – для лексем, литералов `isConst=true`, а для всего остального `isConst=false`;
- **шаг рекурсии** – для внутренних узлов `isConst` вычисляется из дочерних узлов, но только через грамматику выражений, где выражения имеют значения.

Если вам интересно, на какие правила производства ссылается грамматика выражений, то это в основном те правила, которые можно вывести из нетерминала с именем *Expr*. Реализация этого метода в Unicon – это еще один обход в *tree.icn*, как показано здесь:

```
method calc_isConst()
     case sym of {
             "INTLIT" | "DOUBLELIT" | "STRINGLIT" |
             "BOOLFALSE" | "BOOLTRUE": isConst := "true"
             "UnaryExpr": isConst := \kid[2].isConst
             "RelExpr": isConst := \kid[1].isConst &
                     \kid[3].isConst
             "CondOrExpr" | "CondAndExpr" | "EqExpr" |
             "MULEXPR"|
             "ADDEXPR": isConst := \kid[1].isConst &
                     \kid[2].isConst
             default: isConst := &null
     }
     every(!\kids).calc_isConst()
end
```

В приведенном выше коде есть несколько особых случаев. Являются ли бинарные реляционные операторы, такие как оператор «меньше чем» (<), постоянными, зависит от первого и третьего дочерних элементов. Большинство других бинарных операторов не помещают оператор в дерево как средний лист, они вычисляются из значений `isConst` первого и второго дочерних элементов. Java-реализация метода `calc_isConst()` выглядит следующим образом:

```
void calc_isConst() {
     switch(sym) {
     case "INTLIT": case "DOUBLELIT": case "STRINGLIT":
     case "BOOLFALSE": case "BOOLTRUE": isConst = true;
             break;
     case "UnaryExpr": isConst = kid[1].isConst; break;
     case "RelExpr":
             isConst = kid[0].isConst && kid[2].isConst; break;
     case "CondOrExpr": case "CondAndExpr":
     case "EqExpr": case "MULEXPR": case "ADDEXPR":
             isConst = kid[0].isConst && kid[1].isConst; break;
```

```
        default: isConst = false;
        }
        for(int i=0; i <nkids; i++)
                kids[i].calc_isConst();
}
```

Весь метод представляет собой переключатель для обработки базового случая и задания `isConst`, за которым следует обход нуля или более дочерних элементов. Java, возможно, так же хорош, как и Unicon, или немного лучше при вычислении синтезированного атрибута `isConst`.

На этом мы завершаем данный раздел о создании и заполнении таблиц символов. Основным навыком, в котором мы практиковались, был навык написания обходов деревьев, являющихся рекурсивными функциями. Обычный обход дерева посещает все дочерние элементы и обрабатывает их одинаково. Язык программирования может обходить дерево выборочно. Он может игнорировать некоторые дочерние элементы или делать разные вещи с разными потомками. Теперь рассмотрим пример того, как таблицы символов могут быть использованы для обнаружения необъявленных переменных.

Проверка наличия необъявленных переменных

Чтобы найти необъявленные переменные, проверьте таблицу символов для каждой переменной, которая используется для присвоения или разыменования. Эти операции чтения и записи из памяти происходят в исполняемых операциях и выражениях, значения которых вычисляются в этих операциях. Имея дерево синтаксиса, как их найти? Ответ заключается в использовании обхода дерева, который ищет токены *IDENTIFIER*, но только те, которые находятся в исполняемых операторах внутри блоков кода. Чтобы сделать это, начните сверху обход дерева, который просто найдет блоки кода. В Jzero это обход, который находит тела методов.

Идентификация тел методов

Метод `check_codeblocks()` обходит дерево сверху, чтобы найти все тела методов, в которых находится исполняемый код на Jzero. Для каждого найденного объявления метода он вызывает в теле этого метода другой метод `check_block()`:

```
method check_codeblocks()
      if sym == "MethodDecl" then { kids[2].check_block() }
      else every k := !\kids do
              if k.nkids>0 then k.check_codeblocks()
end
```

Соответствующая Java-реализация `check_codeblocks()` находится в файле *tree.java*:

```
void check_codeblocks() {
tree k;
      if (sym.equals("MethodDecl")) { kids[1].check_block(); }
      else {
              for(int i = 0; i<=nkids; i++){
                      k := kids[i];
                      if (k.nkids>0) k.check_codeblocks();
              }
      }
}
```

Приведенный выше метод демонстрирует шаблон поиска по дереву синтаксиса одного конкретного типа узла дерева. Он не вызывает себя рекурсивно на `MethodDecl`. Вместо этого он вызывает более специализированный метод `check_block()`, который реализует работу, выполняемую в случае, когда тело метода найдено. Этот метод знает, что он находится в теле метода, где идентификаторы, которые он находит, являются использованием переменных.

Выявление использования переменных в теле метода

В теле метода любой найденный IDENTIFIER заведомо находится внутри блока операторов исполняемого кода. Исключением является то, что новые переменные, вводимые с помощью объявлений локальной переменной, не могут быть необъявленными переменными:

```
method check_block()
      case sym of {
      "IDENTIFIER": {
              if not (stab.lookup(tok.text)) then
              j0.semerror("undeclared variable "||tok.text)
              }
      "FieldAccess" | "QualifiedName": kids[1].check_block()
      "MethodCall": {
              kids[1].check_block()
              if rule = 1290 then
              kids[2].check_block()
              else kids[3].check_block()
              }
      "LocalVarDecl": { } # skip
      default: {
              every k := !kids do {
                              k.check_block()
                      }
              }
      }
end
```

Приведенный выше метод `check_block()` обрабатывает несколько особых случаев формы дерева. Обратитесь к файлу грамматики *j0gram.y*, чтобы изучить использование идентификаторов (`IDENTIFIER`), которые не просматриваются в локальной таблице символов из-за их синтаксического контекста. В случае

FieldAccess или QualifiedName вторым дочерним элементом является IDENTIFIER, который является именем поля, а не именем переменной. Это можно будет узнать в следующих главах, где будет добавлена информация о типе. Аналогичным образом правило *rule 1291*, второе правило производства MethodCall, пропускает свой второй дочерний элемент. Соответствующий метод Java выглядит следующим образом:

```
void check_block() {
      switch (sym) {
      case "IDENTIFIER": {
              if (stab.lookup(tok.text) == null)
                      j0.semerror("undeclared variable " + tok.text);
              break;
              }
      case "FieldAccess": case "QualifiedName":
              kids[0].check_block();
              break;
      case "MethodCall": {
              kids[0].check_block()
              if (rule == 1290)
              kids[1].check_block();
              else kids[2].check_block();
              break;
              }
      case "LocalVarDecl": break;
      default:
              for(i=0;i<nkids;i++)
                      kids[i].check_block();
      }
}
```

Несмотря на операторы break, реализация Java эквивалентна версии Unicon, описанной ранее. Основная идея, которую вы узнали в этом разделе, заключалась в том, как разделить общую задачу обхода дерева на общий обход, который ищет интересующий узел, а затем специализированный обход, который работает с узлами, найденными в результате общего обхода. Теперь рассмотрим обнаружение семантической ошибки повторного объявления переменной, которая возникает, когда символы вставляются в таблицы символов.

Поиск повторно объявленных переменных

После объявления переменной большинство языков сообщают об ошибке, если та же переменная объявляется снова в той же области видимости. Причина этого заключается в том, что в данной области видимости имя должно иметь единственное, четко определенное значение. Попытка объявить новую переменную повлечет за собой выделение новой памяти, и с этого момента упоминание этого имени станет двусмысленным. Если переменная *x* определена дважды, то не ясно, к какому *x* относится то или иное использование. Вы можете выявить такие ошибки повторного объявления переменной, когда вставляете символы в таблицу символов.

Вставка символов в таблицу символов

Метод `insert()` в классе *symbol table* вызывает базовый интерфейс языка API хеш-таблицы. Метод принимает символ, булев флаг `isConst` и необязательную вложенную таблицу символов для символов, которые вводят новую (под)область видимости. Реализация Unicon-метода `insert()` таблицы символов показана здесь. Если вы перейдете на *https://github.com/PacktPublishing/Build-Your-Own-ProgrammingLanguage/tree/master/ch6*, ее можно найти в файле *symtab.icn* вместе с другими методами класса *symtab*.

```
method insert(s, isConst, sub)
     if \ (t[s]) then j0.semerror("redeclaration of "||s)
     else t[s] := symtab_entry(s, self, sub, isConst)
end
```

Перед вставкой выполняется поиск в таблице символов. Если символ уже присутствует, то сообщается об ошибке повторного объявления. Соответствующая Java-реализация методов insert() таблицы символов выглядит следующим образом:

```
void insert(String s, Boolean iC, symtab sub) {
     if (t.containsKey(s)) {
             j0.semerror("redeclaration of " + s);
     } else {
             sub.parent = this;
             t.put(s, new symtab_entry(s, this, iC, sub));
     }
}
void insert(String s, Boolean iC) {
     if (t.containsKey(s)) {
             j0.semerror("redeclaration of " + s);
     } else {
             t.put(s, new symtab_entry(s, this, iC));
     }
}
```

Этот код является грубым, но эффективным. Использование базового Java API хеш-таблицы является долгим, но читабельным. Теперь рассмотрим метод `semerror()`.

Сообщение о семантических ошибках

Метод `semerror()` в классе *j0* должен сообщать об ошибке пользователю, а также сделать пометку о том, что произошла ошибка, чтобы компилятор не пытался сгенерировать код. Код для сообщения о семантических ошибках аналогичен коду для сообщения о лексических или синтаксических ошибках, хотя иногда сложнее определить, какая строка в каком файле виновата. Пока можно рассматривать эти ошибки как фатальные и прекращать компиляцию при возникновении одной из них. В следующих главах вы сделаете эту ошибку нефатальной и будете сообщать о дополнительных семантических ошибках после

обнаружения одной из них. Код Unicon для метода `semerror()` класса *j0* выглядит следующим образом:

```
method semerror(s)
        stop("semantic error: ", s)
end
```

Код Java для метода `semerror()` класса *j0* показан здесь:

```
void semerror(String s) {
     System.out.println("semantic error: " + s);
     System.exit(1);
}
```

Идентификация ошибок повторного объявления происходит наиболее естественным образом в процессе заполнения таблицы символов, то есть при попытке вставить объявление. В отличие от ошибки необъявленного символа, при которой все вложенные таблицы символов должны быть проверены, прежде чем будет сообщено об ошибке, при ошибке повторного объявления сообщается немедленно, но только в том случае, если символ уже был объявлен в текущей внутренней области видимости. Теперь посмотрим, как реальный язык программирования решает другие проблемы с таблицами символов, которые не были затронуты в этом обсуждении.

Обработка пакетов и областей видимости классов в Unicon

При создании таблиц символов для Jzero учитываются две области видимости – класса и локальная. Поскольку Jzero не делает экземпляров, область видимости класса в Jzero статична и является лексической. Более крупный, реальный язык должен проделать больше работы для обработки областей видимости. Java, например, должен различать, когда символ, объявленный в области видимости класса, является ссылкой на переменную, общую для всех экземпляров класса, и когда символ является обычной переменной-членом, которая была выделена отдельно для каждого экземпляра класса. В случае Jzero булево значение *isMember* может быть добавлено к записям таблицы символов, чтобы отличать переменные-члены от переменных класса, подобно тому как это делает флаг *isConst*.

Реализация Unicon значительно отличается от реализации Jzero. Сводка его таблиц символов и областей видимости классов позволяет провести плодотворное сравнение. Все, что он делает аналогично Jzero, может быть и таким, как это делают другие языки. То, что Unicon делает иначе, чем Jzero, каждый язык может делать по-своему. То, как Unicon обрабатывает эти темы, представлено здесь ради его необычных идей из реального мира, а не потому, что он является каким-то образцовым или идеальным.

Одно из основных различий между Unicon и примером Jzero в этой главе заключается в том, что дерево синтаксиса Unicon представляет собой неоднородную смесь различных типов узлов дерева. Помимо общего типа узла дерева, существуют отдельные типы узлов дерева для представления классов, методов и некоторых других семантически значимых языковых конструкций. Общий тип узла дерева находится в файле с именем *tree.icn*, в то время как другие классы находятся в файле с именем *idol.icn*, который является потомком предшественника Unicon, языка под названием **Idol**. Теперь рассмотрим еще одно различие между Unicon и Jzero, которое проявляется в реализации пакетов. Это известно как искажение имен.

Искажение имен

Проверка области видимости может сообщить, что символ был найден в пакете. Многие языки программирования, и исторический C++ является тому ярким примером, используют искажение имен в генерируемом коде. В Unicon некоторые правила определения области видимости решаются с помощью объединения имен. Имя, такое как *foo*, если оно найдено в области видимости для пакета *bar*, записывается в сгенерированном коде как *bar__foo*.

Метод `mangle_sym(sym)` из реализации Unicon был представлен в его частичной форме здесь и был немного абстрагирован для удобства чтения. Этот метод принимает символ (строку) и *изменяет* его в соответствии с тем, к какому импортированному пакету он принадлежит, включая объявленный пакет текущего файла, который имеет приоритет перед любыми импортами:

```
procedure mangle_sym(sym)
…
      if member(package_level_syms, sym) then
              return package_mangled_symbol(sym)
      if member(imported, sym) then {
              L := imported[sym]
              if *L > 1 then
                      yyerror(sym || " is imported from multiple
                              packages")
              else return L[1] || "__" || sym
      }
      return sym
end
```

В методе `mangle_sym()` таблица Unicon с именем `package_level_syms` хранит записи для символов, объявленных в пакете, связанном с текущим файлом. Другая таблица, называемая `imported`, отслеживает все символы, определенные в других пакетах. Эта таблица возвращает список других пакетов, в которых встречается символ. Размер этого списка задается величиной `*L`. Если символ определен в двух или более импортированных пакетах, использование этого символа в данном файле будет неоднозначным и приведет к ошибке. Использование пакетов – это относительно простой механизм компиляции для создания отдельных пространств имен для различных областей видимости. Более сложные правила определения областей видимости должны быть обработаны

во время выполнения. Например, доступ к членам класса в Unicon требует от компилятора генерировать код, который использует ссылку на текущий объект с именем `self`.

Вставка self для ссылок на переменные-члены

Правила определения области видимости могут ответить, что символ является переменной-членом класса. В Unicon все методы не статичны, и вызовы методов всегда имеют неявный первый параметр с именем `self`, который является ссылкой на объект, для которого был вызван метод. Область видимости класса реализуется путем присоединения к имени оператора точки для ссылки на переменную внутри объекта `self`. Этот код, извлеченный из метода с именем *scopeck_expr()* в файле семантического анализа *idol.icn* компании Unicon, иллюстрирует, как `self` может быть префиксом для ссылок на переменные-члены:

```
«token»: {
     if node.tok = IDENT then {
            if not member(\local_vars, node.s) then {
                   if classfield_member(\self_vars, node.s)then
                          node.s := "self." || node.s
                   else
            node.s := mangle_sym(node.s)
            }
     }
}
```

Этот код модифицирует содержимое существующего поля дерева синтаксиса на месте. Использование префикса `self.` возможно потому, что код записывается в форме, подобной исходному коду, и далее компилируется в байт-код языка C или виртуальной машины последующим генератором кода. Использование `self` в качестве ссылки на текущий объект необходимо не только для доступа к переменным-членам объекта, но и для доступа к вызовам методов объекта. С другой стороны, давайте рассмотрим, как Unicon предоставляет переменную `self`, когда вызываются методы.

Вставка self в качестве первого параметра в вызовы методов

Когда идентификатор появляется перед круглыми скобками, синтаксис указывает, что это имя вызываемой функции или метода. В этом случае требуется дополнительная специальная обработка. Префикс вставки для метода должен искать имя метода во вспомогательной структуре, называемой вектором методов. Ссылка на вектор методов осуществляется через *self.__m*. Например, для метода с именем *meth*, вместо того чтобы стать *self.meth*, ссылка на метод становится *self.__m.meth*.

В дополнение к использованию вектора методов вызов метода *__m* требует, чтобы *self* был вставлен в качестве первого параметра в вызове. В предшественнике Unicon это было явно выражено в генерируемом коде. Такой вызов, как *meth(x)*, становился *self.__m.meth(self, x)*. В реализации Unicon эта вставка

объекта в список параметров вызова встроена в реализацию оператора *dot* в среде выполнения. Когда оператору предлагается выполнить *self.meth*, он просматривает *meth* на предмет того, является ли он обычной переменной-членом. Если это не так, он проверяет, существует ли *self.__m.meth*, и если да, то оператор *dot* ищет эту функцию и помещает *self* в стек как ее первый параметр.

Подведем итог: виртуальная машина Unicon была модифицирована, чтобы сделать генерацию кода для вызова методов более простой. Рассмотрим вызов `o.m()` в следующем примере. Семантика вызова `o.m(3,4)` эквивалентна `o.__m.m(o,3,4)`, но компилятор просто генерирует инструкции для `o.m(3,4)`, а оператор Unicon *dot* делает всю работу:

```
class C(…)
        method m(c,d); … end
end
procedure main()
     o := C(1,2)
     o.m(3,4)
end
```

Одна из приятных сторон создания языка программирования заключается в том, что вы можете заставить среду выполнения, которая запускает сгенерированный вами код, делать все, что вы захотите. Теперь давайте рассмотрим, как протестировать и отладить ваши таблицы символов, чтобы определить их корректность и работоспособность.

Тестирование и отладка таблиц символов

Вы можете протестировать свои таблицы символов, написав множество тестовых примеров и проверив, получают ли они ожидаемые сообщения об ошибках необъявленных или повторно объявленных переменных. Но ничто не придает такой уверенности, как реальное визуальное представление ваших таблиц символов. Если вы построили свои таблицы символов корректно, следуя указаниям этой главы, то должно получиться дерево таблиц символов. Вы можете распечатать таблицы символов, используя те же методы печати деревьев, которые использовались для проверки деревьев синтаксиса в предыдущей главе, применяя либо текстовое, либо графическое представление.

Обходы таблиц символов немного сложнее, чем обходы деревьев синтаксиса. Чтобы вывести таблицу символов, вам нужно вывести информацию для таблицы, а затем посетить все дочерние таблицы, а не просто найти одну из них по имени. Кроме того, здесь задействованы два класса – *symtab* и *symtab_entry*. Предположим, вы начинаете с корневой таблицы символов. Чтобы просмотреть в Unicon все таблицы символов, используйте следующий метод из файла *symtab.icn*:

```
method print(level:0)
     writes(repl(" ",level))
     write(scope, " - ", *t, " symbols")
     every (!t).print(level+1);
end
```

Обратите внимание, что хотя дочерние элементы вызываются одноименным методом, метод `print()` в `symtab_entry` – это другой метод, нежели метод в `symtab`. Код Java для метода `print()` таблицы символов выглядит следующим образом:

```
void print() { print(0); }
void print(int level) {
      for(int i=0;i<level;i++) System.out.print(" ");
      System.out.print(scope + " - " + t.size()+" symbols");
      for (symtab_entry : t.values()) se.print(level+1);
}
```

Методом `print()` `symtab_entry` выводится реальный символ. Если эта запись таблицы символов имеет подобласть видимости, то он распечатывается, и отступы делаются более глубокими, чтобы показать вложенность областей видимости:

```
method print(level:0)
      writes(repl(" ",level), sym)
      if \isConst then writes(" (const)")
      write()
      (\st).print(level+1);
end
```

Взаимно рекурсивный вызов для печати вложенной таблицы символов пропускается, если она пуста. В Java код длиннее, но более явный:

```
void print(level:0) {
      for(int i=0;i<level;i++) System.out.print(" ");
      System.out.print(sym);
      if (isConst) System.out.print(" (const)");
      System.out.println("");
      if (st != null) st.print(level+1);
}
```

Печать таблиц символов не занимает много строк кода. Вы можете решить, что для объявленных переменных стоит добавить дополнительную лексическую информацию, такую как имена файлов и номера строк. В следующих главах будет логично расширить эти методы, добавив в них информацию о типе.

Чтобы запустить компилятор Jzero с выводом таблицы символов, показанной в этой главе, загрузите код из репозитория GitHub этой книги, перейдите в подкаталог *ch6/* и постройте таблицу с помощью программы *make*. По умолчанию *make* построит как версию Unicon, так и версию Java. Если запустить команду *j0* с выводом таблицы символов, она выдаст результат, показанный на рис. 6.4. В данном случае представлена реализация Java.

```
C:\Users\clint\books\byopl\github\Build-Your-Own-Programming-Language\ch6>set CLASSPATH=".;C:\Users
\clint\books\byopl\github\Build-Your-Own-Programming-Language"

C:\Users\clint\books\byopl\github\Build-Your-Own-Programming-Language\ch6>java ch6.j0 hello.java
yyfilename hello.java
global - 2 symbols
 hello
  class - 2 symbols
   main
    method - 0 symbols
   System
 System
  class - 1 symbols
   out
    class - 1 symbols
     println
no errors

C:\Users\clint\books\byopl\github\Build-Your-Own-Programming-Language\ch6>
```

Рис. 6.4. Вывод таблицы символов из компилятора Jzero

Вы должны довольно внимательно прочитать входной файл *hello.java*, чтобы убедиться, что вывод этой таблицы символов является корректным и полным. Чем сложнее правила отображения и области видимости вашего языка, тем более сложным будет вывод таблицы символов. Например, в этом выводе нет данных о публичном и приватном статусе переменной, но для полноценного компилятора Java это необходимо. Когда вы убедитесь, что все символы присутствуют и учтены в корректных областях видимости, вы можете перейти к следующему этапу семантического анализа.

Заключение

В этой главе вы узнали о важнейших технических навыках и инструментах, используемых для построения таблиц символов, которые отслеживают все переменные во всех областях видимости входной программы. Вы создаете таблицу символов для каждой области видимости в программе и вставляете записи в нужную таблицу символов для каждой переменной. Все это выполняется с помощью обхода дерева синтаксиса.

Вы узнали, как писать обходы дерева, которые создают таблицы символов для каждой области видимости, а также как создать наследуемый атрибут для таблицы символов, связанной с текущей областью видимости, для каждого узла в дереве синтаксиса. Затем вы научились вставлять информацию о символах в таблицы символов, связанные с вашим деревом синтаксиса, и обнаруживать, когда один и тот же символ незаконно объявляется повторно. Вы узнали, как писать обходы дерева, которые ищут информацию в таблицах символов, и выявлять любые ошибки необъявленных переменных. Эти навыки позволили вам сделать первые шаги в обеспечении соблюдения семантических правил, связанных с вашим языком программирования. В остальной части вашего компилятора как семантический анализ, так и генерация кода основывались на таблицах символов, которые вы создали в этой главе, и добавлялись к ним.

Теперь, когда вы построили таблицы символов, пройдя по дереву разбора с помощью обходов дерева, пришло время подумать о том, как проверить использование типов данных в программе. В следующей главе мы начнем это путешествие с того, как проверить основные типы, такие как целые и вещественные числа.

Вопросы

1. Какая связь между различными таблицами символов, которые создаются в компиляторе, и деревом синтаксиса, созданным в предыдущей главе?
2. В чем разница между синтезированными и унаследованными семантическими атрибутами? Как они вычисляются и где хранятся?
3. Сколько таблиц символов необходимо в языке Jzero? Как организованы таблицы символов?
4. Предположим, что наш язык Jzero допускает несколько классов, компилируемых отдельно в отдельных исходных файлах. Как бы это повлияло на нашу реализацию таблиц символов в этой главе?

Глава 7

Проверка базовых типов

Это первая из двух глав о проверке типов. В большинстве основных языков программирования **проверка типов** является ключевым аспектом семантического анализа, который должен быть выполнен до того, как вы сможете генерировать код.

В этой главе вы узнаете, как выполнить простую проверку типов для базовых типов, включенных в подмножество Jzero языка Java. Побочным продуктом проверки типов является добавление информации о типах в дерево синтаксиса. Знание типов операндов в дереве синтаксиса позволяет генерировать правильные команды для различных операций.

В этой главе рассматриваются следующие основные темы:

- представление типов в компиляторе;
- присвоение информации о типе объявленным переменным;
- определение типа в каждом узле дерева синтаксиса;
- проверка типов во время выполнения и вывод типов – пример Unicon.

Пришло время узнать о проверке типов, начиная с базовых типов. Некоторые из вас могут задаться вопросом: зачем вообще нужна проверка типов? Если ваш компилятор не выполняет проверку типов, ему придется генерировать код, который работает независимо от того, какие типы операндов используются. Lisp, BASIC и Unicon являются примерами языков с таким подходом к проектированию. Часто это делает язык удобным для пользователя, но он работает медленнее. По этой причине мы рассмотрим проверку типов. Начнем с рассмотрения того, как представлять информацию о типах, которую вы извлекаете из исходного кода.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/.PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch7*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/3cgvkWT*.

Представление типов в компиляторе

Часто нашему компилятору приходится выполнять такие действия, как сравнение типов двух переменных, чтобы проверить, совместимы ли они. Исходный код программы представляет типы с помощью строковых данных, вклю-

ченных в наше дерево синтаксиса. В некоторых языках можно использовать небольшие поддеревья синтаксиса для представления типов, которые используются при проверке типов, но в общем случае информация о типе не совсем соответствует поддереву в нашем дереве синтаксиса. Это происходит потому, что часть информации о типе берется из другого места, например из другого типа. По этой причине нам нужен новый тип данных, который будет представлять информацию о типе, связанную с любым значением, которое объявляется или вычисляется в программе.

Было бы хорошо, если бы мы могли просто представлять типы с одним атомарным значением, таким как целочисленный код или строковое имя типа. Например, мы могли бы использовать 1 для целого числа, 2 для вещественного числа или 3 – для строки. Если язык имеет только небольшой, фиксированный набор встроенных типов, то достаточно будет использовать атомарное значение. Однако типы реального языка сложнее. Представление составных типов, таких как массив, класс или метод, является более сложным. Вы можете начать с базового класса, способного представлять атомарные типы.

Определение базового класса для представления типов

Информация о типе, связанная с любым именем или значением в вашем языке, может быть представлена в новом классе с именем **typeinfo**. Класс *typeinfo* не называется типом, потому что некоторые языки программирования используют его как зарезервированное слово или встроенное имя. В Unicon это имя встроенной функции, поэтому объявлять класс с таким именем было бы плохо.

Класс *typeinfo* имеет член *basetype* для хранения того, какой тип данных представлен. Сложные типы содержат дополнительную информацию по мере необходимости. Например, тип, *basetype* которого указывает на то, что он является массивом, имеет дополнительный *element_type*. С помощью этой дополнительной информации мы сможем отличить массив целых чисел от массива строк или массива какого-либо типа класса. В некоторых языках типы массивов также имеют явный размер или начальный и конечный индексы.

Есть много способов, которыми вы можете справиться с этой разницей в информации, необходимой для различных типов. Классическим объектно-ориентированным представлением этих различий является использование подклассов. Для Jzero мы добавим *arraytype*, *methodtype* и *classtype* в качестве подклассов *typeinfo*. Во-первых, есть сам суперкласс, который можно найти в файле *typeinfo.icn*, как показано в следующем коде:

```
class typeinfo(basetype)
      method str()
              return string(basetype)|"unknown"
      end
end
```

В дополнение к члену `basetype` класс `typeinfo` имеет методы, облегчающие отладку. Типы должны иметь возможность выводить информацию о себе в человекочитаемом формате. Java-версия в файле *typeinfo.java* выглядит следующим образом:

```
public class typeinfo {
      String basetype;
      public typeinfo() { basetype = "unknown"; }
      public typeinfo(String s) { basetype = s; }
      public String str() { return basetype; }
}
```

Дополнительный конструктор, не принимающий аргументов, необходим для того, чтобы подклассы компилировались правильно в Java. Наличие класса, а не просто целого числа для кодирования информации о типе позволяет нам представлять более сложные типы путем создания подклассов базового класса.

Подклассификация базового класса для сложных типов

Код Unicon для подклассов `typeinfo` также хранится в *typeinfo.icn*, поскольку они короткие и тесно связаны. В Jzero класс `arraytype` имеет только `element_type`. В других языках тип массива может требовать дополнительных полей для хранения размера массива или типа и диапазона допустимых индексов. Представление Unicon типа массива в Jzero выглядит следующим образом:

```
class arraytype : typeinfo(element_type)
initially
        basetype := "array"
end
```

Файл *arraytype.java* содержит соответствующую Java-реализацию класса `arraytype`:

```
public class arraytype extends typeinfo {
      typeinfo element_type;
      public arraytype(typeinfo t) {
              basetype = "array"; element_type = t; }
}
```

Представление для методов, также называемых **функциями-членами класса**, включает сигнатуру, состоящую из их параметров и возвращаемого типа. На данный момент это позволяет идентифицировать методы как таковые. Реализация класса `methodtype` в Unicon выглядит следующим образом:

```
class methodtype : typeinfo(parameters,return_type)
initially
        basetype := "method"
end
```

Типы методов содержат список из нуля или более параметров и возвращаемый тип, они будут использоваться в следующей главе для проверки типов при вызове методов (функций). Java-представление методов выглядит следующим образом и может быть найдено в файле *methodtype.java*:

```
public class methodtype extends typeinfo {
      parameter [] parameters;
      typeinfo return_type;
      methodtype(parameter [] p, typeinfo rt){
              parameters = p; return_type = rt;
      }
}
```

Параметрами (`parameters`) может быть массив `typeinfo`. Для параметров определен отдельный класс, чтобы позволить языкам включать имена параметров вместе с их типами для представления методов. Реализация Unicon выглядит следующим образом:

```
class parameter(name, param_type)
end
```

Некоторые из этих классов кажутся довольно пустыми. Они являются заполнителями, которые будут включать больше кода в последующих главах или потребуют более существенной обработки на других языках. Соответствующая Java-реализация класса `parameter` в файле *parameter.java* показана здесь:

```
public class parameter {
      String name;
      typeinfo param_type;
      parameter(String s, typeinfo t) { name=s; param_type=t; }
}
```

Класс для представления классов включает имя класса, связанную с ним таблицу символов и списки из нуля или более полей, методов и конструкторов. В некоторых языках это может быть сложнее, чем в Jzero, например включать суперклассы. Реализация в Unicon показана здесь:

```
class classtype : typeinfo(name, st, fields, methods,
              constrs)
      method str()
              return name
      end
initially
        basetype := "class"
end
```

Вам может быть интересно поле `st`, в котором хранится таблица символов. В *главе 6 «Таблицы символов»* таблицы символов были построены и сохранены в узлах дерева синтаксиса, где они образуют логическое дерево, соответствующее объявленным областям видимости программы. Ссылки на эти же таблицы символов должны быть помещены в типы, чтобы мы могли вычислить тип, полученный в результате использования оператора *dot*, который ссылается на область видимости, не связанную с деревом синтаксиса. Файл *classtype.java* содержит реализацию Java-класса `classtype`. Следующий код показывает пример этого:

```
public class classtype extends typeinfo {
      String name;
      symtab st;
      parameter [] methods;
      parameter [] fields;
      typeinfo [] constrs;
}
```

Учитывая класс `typeinfo`, целесообразно добавить поле-член этого типа в класс `tree` и класс `symtab_entry`, чтобы информация о типе могла быть представлена для выражений и переменных. В обоих классах мы будем называть его `typ`:

```
class tree (id,sym,rule,nkids,tok,kids,isConst,stab,typ)
class symtab_entry(sym,parent_st,st,isConst,typ)
```

Мы не повторяем здесь классы полностью; код для этого можно найти в подкаталоге *ch7/* на сайте *https://github.com/PacktPublishing/Build-YourOwn-Programming-Language*. В Java соответствующие классы изменяются следующим образом:

```
class tree {   .   .   .
        typeinfo typ;   .   .   .   }
class symtab_entry {   .   .   .
      typeinfo typ;   .   .   .   }
```

Учитывая поле `typ`, можно написать обходы мини-дерева, необходимые для размещения информации о типе в таблицах символов с переменными по мере их объявления. Давайте рассмотрим присвоение информации о типе объявленным переменным.

Присвоение информации о типе объявленным переменным

Информация о типе строится во время обхода дерева и затем хранится вместе с ассоциированными с ней переменными в таблице символов. Обычно это является частью обхода, которая заполняет таблицу символов, как это было представлено в предыдущей главе. В этом разделе мы будем обходить дерево синтаксиса в поисках объявлений переменных, как мы это делали ранее, но на этот раз нам необходимо распространить информацию о типе с помощью синтезированных и/или наследуемых атрибутов.

Чтобы информация о типе была доступна в тот момент, когда мы вставляем переменные в таблицу символов, эта информация должна быть вычислена в какой-то предшествующий момент времени. Информация о типе вычисляется либо при предыдущем обходе дерева, либо при парсинге, когда строится дерево синтаксиса. Рассмотрим следующее грамматическое правило и семантическое действие из главы 5 «Деревья синтаксиса»:

```
FieldDecl: Type VarDecls ';' {
        $$=j0.node("FieldDecl",1030,$1,$2); };
```

Семантическое действие строит новый узел дерева с именем *FieldDecl*, соединяющий *Type* с *VarDecls*. Ваш компилятор должен синтезировать информацию о типе из *Type* и наследовать ее в *VarDecls*. Информацию, поступающую вверх из левого поддерева и уходящую вниз в правое поддерево, можно увидеть на рис. 7.1.

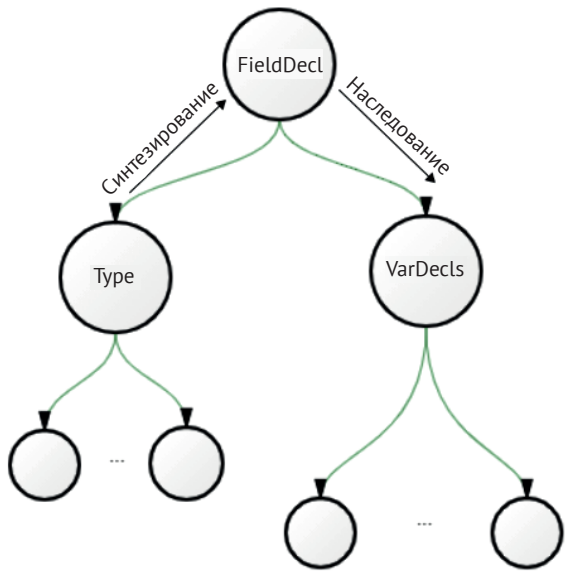


Рис. 7.1. Поток информации о типе в объявлениях переменных

Мы можем включить это в процесс построения дерева синтаксиса с помощью мини-обходов поддеревьев. Следующий код добавляет вызов из файла *j0gram.y* метода с именем `calctype()`, в котором будет проводиться семантический анализ, как показано в предыдущем примере:

```
FieldDecl: Type VarDecls ';' {
      $$=j0.node("FieldDecl",1030,$1,$2);
      j0.calctype($$);
};
```

Изучая грамматику, вы можете заметить, что аналогичный вызов `calctype()` необходим для нетерминала *FormalParm* и что в грамматике есть несколько дополнительных мест, где тип ассоциируется с идентификатором или списком идентификаторов. Метод `calctype()` класса `j0` разворачивается и вызывает два обхода дерева на двух дочерних элементах `FieldDecl`. Версия этого метода для Unicon в файле *j0.icn* выглядит следующим образом:

```
method calctype(t)
      t.kids[1].calctype()
      t.kids[2].assigntype(t.kids[1].typ)
end
```

Метод `calctype()` класса `j0` вызывает метод `calctype()` класса `tree`, который вычисляет синтезированный атрибут `typ` в левом дочернем элементе. Затем тип передается вниз, как наследуемый атрибут, в правый дочерний элемент. Java-версия этого метода в файле *j0.java* выглядит следующим образом:

```
void calctype(parserVal pv){
      tree t = (tree)pv.obj;
      t.kids[0].calctype();
      t.kids[1].assigntype(t.kids[0].typ);
}
```

По сравнению с обходом деревьев, который мы рассматривали в предыдущих главах, методы класса дерева, включая `calctype()` и `assigntype()`, являются особыми случаями, для которых форма дерева и виды возможных узлов ограничены. Чтобы использовать это преимущество, код обхода может быть изменен. Начнем с рассмотрения метода `calctype()`.

Синтез типов из зарезервированных слов

Метод `calctype()` вычисляет синтезированный атрибут `typ`. Сначала выполняется рекурсивная работа по вычислению значения для дочерних узлов, а затем вычисляется значение для текущего узла. Такая форма обхода называется **обратным обходом** и часто встречается в компиляторах. В Unicon метод `calctype()` в классе *tree* файла *tree.icn* выглядит следующим образом:

```
method calctype()
      every (!\kids).calctype()
      case sym of {
              "FieldDecl": typ := kids[1].typ
              "token": {
                      case tok.cat of {
                              parser.IDENTIFIER:{return typ :=
                                      classtype(tok.text) }
                              parser.INT:{ return typ := typeinfo(tok.text) }
                              default:
                                      stop("can't grok the type of ", image(tok.
                              text))
                              }
                      }
              default:
                      stop("can't grok the type of ", image(sym)) }
end
```

Этот код строит значение `typ` текущего узла дерева, используя информацию из его дочерних узлов, в данном случае путем прямого доступа к пуб-

личному полю typ дочернего узла. Другой способ получения информации от дочернего узла – это вызов метода, который возвращает тип дочернего узла в качестве возвращаемого значения, например возвращаемое значение calctype(). В этом коде количество ветвей невелико, поскольку грамматика Jzero для нетерминала *Type* минимальна. В других языках она была бы более богатой. Соответствующий Java-код показан в методе calctype() из файла *tree.java*:

```
typeinfo calctype() {
     for(int i=0; i<nkids; i++) kids[i].calctype()
     switch (sym) {
             case "FieldDecl": return typ = kids[0].typ;
             case "token": {
                     switch (tok.cat) {
                             case parser.IDENTIFIER:{
                                     return typ=new classtype(tok.text); }
                             case parser.INT: { return typ=new
                                     typeinfo(tok.text); }
                             default:
                                     j0.semerror("can't grok the type of " +
                                             tok.text);
                     }
             }
              default:
                     j0.semerror("don't know how to calctype " + sym);}
}
```

Пусть мы синтезировали тип из левого дочернего узла FieldDecl. Рассмотрим, как унаследовать этот тип в узлах переменных в правом дочернем поддереве FieldDecl.

Наследование типов в списке переменных

Передача информации о типе в поддерево выполняется в методе assigntype(t). Наследуемые атрибуты обычно кодируются с помощью **прямого обхода**, в котором текущий узел выполняет свою работу, а затем вызывает дочерние узлы с информацией, которую они наследуют. Реализация метода assigntype(t) в Unicon выглядит следующим образом:

```
method assigntype(t)
     typ := t
     case sym of {
     "VarDeclarator": {
             kids[1].assigntype(arraytype(t))
             return
             }
     "token": {
             case tok.cat of {
                     parser.IDENTIFIER:{
                             return
                             }
```

```
                    default: stop("eh? ", image(tok.cat))
                    }
            }
      default:
                    stop("don't know how to assign the type of ", image(sym))
            }
      every (!\kids).assigntype(t)
end
```

Поскольку информация передается от родителя к потомкам, естественно передать эту информацию в качестве параметра потомку, который затем присваивает ее своему типу через `typ := t`. Также можно скопировать ее через явное присвоение публичному полю потомка. Соответствующая Java-реализация метода `assigntype(t)` показана здесь:

```
void assigntype(typeinfo t) {
      typ = t;
      switch (sym) {
      case "VarDeclarator": {
              kids[0].assigntype(new arraytype(t));
              return;
      }
      case "token": {
              switch (tok.cat) {
                      case parser.IDENTIFIER:{ return; }
                      default: j0.semerror("eh? " + tok.cat);
              }
      }
      default:
              j0.semerror("don't know how to assigntype " + sym);
      }
      for(tree k : kids) k.assigntype(t);
}
```

Прикрепление информации о типе к именам переменных там, где они объявлены, очень важно, и это не слишком сложно, особенно для такого простого языка, как Jzero. Теперь пришло время рассмотреть основную задачу этой главы – как вычислять и проверять информацию о типе в выражениях, составляющих исполняемый код в теле функций и методов.

Определение типа в каждом узле дерева синтаксиса

В дереве синтаксиса узлы, связанные с фактическими выражениями кода в теле метода, имеют тип, связанный со значением, которое вычисляет выражение. Например, если узел дерева соответствует сумме сложения двух чисел, тип узла дерева определяется типами операндов и правилами языка для оператора сложения. Цель этого раздела – объяснить, как можно вычислить эту информацию о типе.

Как вы видели в разделе «*Представление типов в компиляторе*», класс для узлов дерева синтаксиса имеет атрибут для хранения типа узла, если таковой имеется. Атрибут типа вычисляется снизу вверх во время обратного обхода дерева. Здесь есть сходство с проверкой на наличие необъявленных переменных, которую мы делали в предыдущей главе, в том, что проверка типа выражений происходит только в телах функций. Вызов обхода дерева с проверкой типов, начиная с корня дерева синтаксиса, добавляется в конце метода *semantic()* в классе *j0*. В Unicon вызов следующий:

```
root.checktype()
```

Здесь нет параметра, но это то же самое, что передать значение null или false. В Java добавляется следующее утверждение:

```
root.checktype(false);
```

В обоих случаях параметр указывает, находится ли данный узел в теле исполняемой инструкции. В корне ответ – false. Он станет true, когда обход дерева достигнет тел методов, содержащих код. Чтобы выполнить обход дерева, вы должны подумать, что делать с листьями дерева.

Определение типа в листьях

В листьях типы литеральных константных значений очевидны из их лексической категории. Для начала мы должны добавить поле `typ` к токену класса. Для литералов мы должны инициализировать `typ` в конструкторе. В Unicon первая строка и начальная часть файла *token.icn* становятся следующими:

```
class token(cat, text, lineno, colno, ival, dval, sval,typ)
        .   .   .
initially
     case cat of {
     parser.INTLIT:{ ival := integer(text);
             typ:=typeinfo("int")}
     parser.DOUBLELIT:{dval:=real(text);
             typ:=typeinfo("double")}
     parser.STRINGLIT:{
             sval:=deEscape(text); typ := typeinfo("String") }
     parser.BOOLLIT: { typ := typeinfo("boolean") }
     parser.NULLVAL: { typ := typeinfo("null") }
     ord("="|"+"|"-"): { typ := typeinfo("unknown") }
     }
end
```

Приведенный выше код присваивает операторам тип «`unknown`» («неизвестный»; код для вычисления типов выражений, использующих эти операторы, из типов их операндов будет показан позже в разделе «*Вычисление и проверка типов во внутренних узлах*»). В Java соответствующее изменение в классе `token` для литеральных типов выглядит следующим образом:

```
package ch7;
public class token {
        .   .   .
      public typeinfo typ;
      public token(int c, String s, int l) {
              cat = c; text = s; lineno = l;
              id = serial.getid();
              switch (cat) {
              case parser.INTLIT: typ = new typeinfo("int"); break;
              case parser.DOUBLELIT:typ = new typeinfo("double");
                      break;
              case parser.STRINGLIT: typ= new typeinfo("String");
                      break;
              case parser.BOOLLIT: typ = new typeinfo("boolean");
                      break;
              case parser.NULLVAL: typ = new typeinfo("null"); break;
              case '=': case '+': case '-':
                      typ = new typeinfo("unknown"); break;
              }
      }
```

Типы переменных ищутся в таблице символов. Это подразумевает, что таблица символов должна быть заполнена до проверки типов. Поиск в таблице символов выполняется методом `type()` и добавляется в класс `token` в файле *token.icn*. Он принимает таблицу символов, в которой токен находится в качестве параметра:

```
method type(stab)
      if \typ then return typ
      if cat === parser.IDENTIFIER then
              if rv := stab.lookup(text) then return typ := rv.typ
      stop("cannot check the type of ",image(text))
end
```

Первая строка этого метода возвращает тип для данного токена немедленно, если он был определен ранее. Если нет, то остальная часть метода просто проверяет, есть ли у нас идентификатор, и если да, то ищет его в таблице символов. Соответствующее дополнение к файлу *token.java* выглядит так:

```
public typeinfo type(symtab stab) {
      symtab_entry rv;
      if (typ != null) return typ;
      if (cat == parser.IDENTIFIER)
              if ((rv = stab.lookup(text)) != null)
                      return typ=rv.typ;
      j0.semerror("cannot check the type of " + text);
}
```

После ознакомления с кодом для вычисления типа листьев дерева синтаксиса пришло время рассмотреть, как проверять типы во внутренних узлах. Это основная функция проверки типов.

Вычисление и проверка типов во внутренних узлах

Внутренние узлы проверяются только в пределах исполняемых инструкций и выражений в коде программы. Это прямой обход, при котором сначала выполняется работа в дочерних узлах, а затем в родительском узле. Процесс посещения дочерних узлов, который делегируется вспомогательной функции `checkkids()`, варьируется в зависимости от узла дерева, а работа, выполняемая в родительском узле, зависит от того, находится ли он в блоке кода:

```
method checktype(in_codeblock)
      if checkkids(in_codeblock) then return
      if /in_codeblock then return
      case sym of {
              "Assignment": typ := check_types(kids[1].typ,
                                               kids[3].typ)
              "AddExpr": typ := check_types(kids[1].typ, kids[2].typ)
              "Block" | "BlockStmts": { typ := &null }
              "MethodCall": { }
              "QualifiedName": {
                      if type(kids[1].typ) == "classtype__state" then {
                              typ := (kids[1].typ.st.lookup(
                                      kids[2].tok.text)).typ
                              } else stop("illegal . on ",kids[1].typ.str())
              }
              "token": typ := tok.type(stab)
              default: stop("cannot check type of ", image(sym))
              }
end
```

В дополнение к вспомогательному методу `checkkids()` этот код полагается на вспомогательную функцию под названием `check_types()`, которая определяет тип результата, учитывая операнды. Соответствующая Java-реализация `checktype()` показана здесь:

```
void checktype(boolean in_codeblock) {
      if (checkkids(in_codeblock)) return;
      if (! in_codeblock) return;
      switch (sym) {
      case "Assignment":
              typ = check_types(kids[0].typ, kids[2].typ); break;
      case "AddExpr":
              typ = check_types(kids[0].typ, kids[1].typ); break;
      case "Block": case "BlockStmts": typ = null; break;
      case "MethodCall": break;
      case "QualifiedName": {
              if (kids[0].typ instanceof classtype) {
                      classtype ct = (classtype)(kids[0].typ);
                      typ = (ct.st.lookup(kids[1].tok.text)).typ;
              } else j0.semerr("illegal . on " + kids[0].typ.str());
              break;
              }
```

```
      case "token": typ = tok.type(stab); break;
      default: j0.semerror("cannot check type of " + sym);
      }
}
```

По умолчанию вспомогательная функция `checkkids()` вызывает `checktype()` для каждого потомка, но в некоторых случаях она этого не делает. Например, при объявлении метода его заголовок не содержит никаких исполняемых кодовых выражений и пропускается, посещается только блок кода, и в этом блоке булев параметр `in_codeblock` устанавливается в *true*. Аналогично в блоке кода, где встречается объявление локальной переменной, посещается только список переменных, и в этом списке параметр `in_codeblock` выключен (только для того, чтобы быть снова включенным в инициализаторах). В качестве другого примера идентификаторы в правой части оператора точки не ищутся в обычной таблице символов. Вместо этого они ищутся относительно вида выражения в левой части оператора точки и поэтому требуют особого обращения. Реализация `checkkids()` в Unicon показана здесь:

```
method checkkids(in_codeblock)
      case sym of {
              "MethodDecl": { kids[2].checktype(1); return }
              "LocalVarDecl": { kids[2].checktype(); return }
              "FieldAccess": { kids[1].checktype(in_codeblock);
                               return }
              "QualifiedName": {
                      kids[1].checktype(in_codeblock);
                      }
              default: { every (!\kids).checktype(in_codeblock) }
              }
end
```

Соответствующая Java-реализация этой вспомогательной функции показана ниже:

```
public boolean checkkids(boolean in_codeblock) {
      switch (sym) {
      case "MethodDecl": kids[1].checktype(true); return true;
      case "LocalVarDecl": kids[1].checktype(false);
                           return true;
      case "FieldAccess": kids[0].checktype(in_codeblock);
                           return true;
      case "QualifiedName":
                          kids[0].checktype(in_codeblock);
                          break;
      default: for (tree k : kids) k.checktype(in_codeblock);
      }
         return false;
}
```

Вспомогательный метод check_types() вычисляет тип текущего узла из типов до двух операндов. Его расчет варьируется в зависимости от того, какой оператор выполняется, а также от правил языка. Ответом может быть то, что тип такой же, как у одного или обоих операндов, или же это может быть какой-то новый тип либо ошибка. В Unicon реализация check_types() в файле *tree.icn* выглядит следующим образом:

```
method check_types(op1, op2)
      operator := get_op()
      case operator of {
              "="|"+"|"-" : {
                      if tok := findatoken() then
                              writes("line ", tok.tok.lineno, ": ")
                      if op1.basetype === op2.basetype === "int" then {
                              write("typecheck ",operator," on a ",
                                      op2.str(), " and a ", op1.str(), " -> OK")
                              return op1
                              }
                      else stop("typecheck ",operator," on a ",
                              op2.str(), " and a ", op1.str(),
                              " -> FAIL")
                      }
              default: stop("cannot check ", image(operator))
      }
end
```

Этот метод опирается на два вспомогательных метода. Метод get_op() сообщает, какой выполняется оператор. Метод findatoken() ищет первый токен в исходном коде, представленном заданным узлом дерева синтаксиса; он используется для сообщения номера строки. Соответствующая Java-реализация check_types() показана здесь:

```
public typeinfo check_types(typeinfo op1, typeinfo op2) {
      String operator = get_op();
      switch (operator) {
      case "=": case "+": case"-": {
              tree tk;
              if ((tk = findatoken())!=null)
                      System.out.print("line " + tk.tok.lineno + ": ");
              if ((op1.basetype.equals(op2.basetype)) &&
                  (op1.basetype.equals("int"))) {
                      System.out.println("typecheck "+operator+" on a "+
                              op2.str() + " and a "+ op1.str()+
                              " -> OK");
              return op1;
      }
              else j0.semerror("typecheck "+operator+" on a "+
                      op2.str()+ " and a "+ op1.str()+
                      " -> FAIL");
              }
```

```
      default: j0.semerror("cannot check " + operator);
      }
return null;
}
```

Оператор, который представляет текущий узел синтаксиса, обычно может быть определен из соответствующего нетерминального символа узла. В некоторых случаях также должно быть использовано фактическое правило производства. Здесь показана реализация `get_op()` в Unicon:

```
method get_op()
      return case sym of {
              "Assignment" : "="
              "AddExpr": if rule=1320 then "+" else "-"
              default: fail
      }
end
```

Unicon позволяет возвращать результат, полученный выражением `case`. Аддитивные выражения, обозначаемые `"AddExpr"`, включают в себя как сложение, так и вычитание. Правило производства используется для устранения неоднозначности. Соответствующая Java-реализация `get_op()` аналогична приведенной ниже:

```
public String get_op() {
      switch (sym) {
      case "Assignment" : return "=";
      case "AddExpr": if (rule==1320) return "+";
                      else return "-";
      }
      return sym;
}
```

Метод `findatoken()` используется из внутреннего узла дерева синтаксиса для поиска одного из его листьев. Он рекурсивно погружается в дочерние узлы, пока не найдет токен. Unicon-реализация `findatoken()` выглядит следующим образом:

```
method findatoken()
if sym==="token" then return self
return (!kids).findatoken()
end
```

Соответствующая Java-реализация `findatoken()` показана здесь:

```
public tree findatoken() {
      tree rv;
      if (sym.equals("token")) return this;
      for (tree t : kids)
              if ((rv=t.findatoken()) != null) return rv;
      return null;
}
```

Даже основы проверки типов, показанные в этом разделе, потребовали от вас изучения множества новых способов обхода деревьев. Дело в том, что создание языка программирования или написание компилятора – это большая, сложная работа, и если бы мы показали полный вариант для какого-нибудь основного языка, эта книга была бы толще, чем позволяет наш лимит страниц.

В этой главе было рассказано о том, как добавить в Jzero примерно половину средств проверки типов. Запуск j0 с этими дополнениями не очень эффектен, он просто позволяет увидеть, как обнаруживаются простые ошибки типов и сообщается о них. Если вы хотите увидеть это, скачайте код с сайта данной книги на GitHub, перейдите в подкаталог *Chapter07/* и постройте код с помощью программы *make*. По умолчанию *make* будет создавать версии как для Unicon, так и для Java. Когда вы запустите команду *j0* с предварительной проверкой типов, она выдаст результат, подобный приведенному на рис. 7.2. На выходе из программы проверки типов получаем `OK` или `FAIL` для различных операторов. В данном случае показана реализация Unicon.

```
D:\Users\Clinton Jeffery\books\byopl\github\Build-Your-Own-Programming-Language\ch7>type hello.java

public class hello {
   public static void main(String argv[]) {
      int x;
      x = 0;
      x = x + "hello";
      System.out.println("hello, jzero!");
   }
}

D:\Users\Clinton Jeffery\books\byopl\github\Build-Your-Own-Programming-Language\ch7>j0 hello.java
line 4: typecheck = on a int and a int -> OK
line 5: typecheck + on a String and a int -> FAIL

D:\Users\Clinton Jeffery\books\byopl\github\Build-Your-Own-Programming-Language\ch7>
```

Рис. 7.2. Результат работы программы проверки типов

Конечно, если в программе нет ошибок типа, вы не увидите ничего, кроме строк, заканчивающихся `OK`. Теперь рассмотрим аспект проверки типов, который встречается при реализации некоторых языков программирования, включая Unicon-проверку типов во время выполнения.

Проверка типов во время выполнения и вывод типов в Unicon

Язык Unicon работает с типами во многом иначе, чем система типов Jzero, описанная в этой главе. В Unicon типы связаны не с объявлением, а с фактическими значениями. Генератор кода виртуальной машины Unicon не помещает информацию о типах в таблицах символов и не выполняет проверку типов во время компиляции. Вместо этого типы представляются в явном виде во время выполнения и проверяются везде перед использованием значения. Явное представление информации о типах во время выполнения является обычным в интерпретируемых и объектно-ориентированных языках и необязательным в некоторых полуобъектно-ориентированных языках, таких как C++.

Рассмотрим функцию *write()* языка Unicon. Каждый аргумент *write()*, не являющийся файлом и указывающий место записи, должен быть строкой или иметь возможность быть преобразованным в строку. В виртуальной машине Unicon информация о типе создается и проверяется во время выполнения по мере необходимости. Псевдокод для функции Unicon *write()* выглядит следующим образом:

```
for (n = 0; n < nargs; n++) {
     if (is:file(x[n])) {
             set the current output file
     } else if (cnv:string(x[n])) {
             output the string to the output file
     } else runtime_error("string or file expected")
}
```

Для каждого аргумента *write()* предыдущий код предлагает либо установить текущий файл, преобразовать аргумент в строку и записать ее, либо остановиться с ошибкой выполнения. Проверка во время выполнения обеспечивает дополнительную гибкость, но замедляет выполнение. Хранение информации о типе во время выполнения также потребляет память – потенциально много памяти. Чтобы выполнить проверку типов во время выполнения, каждое значение в языке Unicon хранится в дескрипторе. Дескриптор – это структура, содержащая значение плюс дополнительное слово памяти, которое кодирует его тип, называемое **d-словом**. Булево выражение, такое как *is:file(x)* для некоторого значения Unicon *x*, сводится к выполнению проверки того, говорит ли d-слово, что значение имеет тип файла.

Unicon также имеет оптимизирующий компилятор, который генерирует код на языке C. Оптимизирующий компилятор выполняет **вывод типа**, который определяет уникальный тип более чем в 90 % случаев, устраняя необходимость в большинстве проверок типов во время выполнения. Рассмотрим следующую тривиальную программу Unicon:

```
procedure main()
     s := "hello" || read()
     write(s)
end
```

Оптимизирующий компилятор знает, что `"hello"` – это строка, а `read()` возвращает только строки. Он может сделать вывод, что переменная `s` содержит только строковые значения, поэтому в данном конкретном вызове `write()` передается значение, которое уже является строкой и не нуждается в проверке или преобразовании. Вывод типов выходит за рамки данной книги, но полезно знать, что он существует и что для некоторых языков он является важным мостом, который позволяет гибким языкам высокого уровня работать со скоростью, сравнимой со скоростью компилируемых языков более низкого уровня.

Заключение

В этой главе вы узнали, как представлять базовые типы и проверять безопасность типов обычных операций, таких как предотвращение добавления целого числа к функции. Все это можно выполнить путем обхода дерева синтаксиса.

Вы узнали, как представлять типы в структуре данных и добавлять атрибут к узлам дерева синтаксиса для хранения этой информации. Вы также узнали, как писать обходы дерева, которые извлекают информацию о типе переменных и хранят ее в записях таблицы символов. Затем вы познакомились с тем, как вычислять правильный тип в каждом узле дерева, проверяя, корректно ли используются типы. Наконец, вы узнали, как сообщать о найденных ошибках типа.

Процесс проверки типов может показаться неблагодарной работой, которая просто приводит к большому количеству сообщений об ошибках, но на самом деле информация о типе, которую вы вычисляете при каждом из вызовов операторов и функций в дереве синтаксиса, будет играть важную роль в определении того, какие машинные инструкции генерировать для этих узлов дерева. Теперь, когда вы построили представление типов и реализовали простые проверки типов, пришло время рассмотреть некоторые более сложные операции, необходимые для проверки составных типов, такие как вызовы функций и классы. Вы сделаете это в следующей главе.

Вопросы

1. Для какой цели служит проверка типов, кроме как для того, чтобы расстраивать уставших программистов?
2. Почему структурный тип (в нашем случае класс) необходим для представления информации о типе? Почему мы не можем просто использовать целое число для представления каждого типа?
3. Код в этой главе выводит строки, которые сообщают о каждой успешной проверке типа символом *OK*. Это очень обнадеживает. Почему другие компиляторы не сообщают об успешной проверке типа подобным образом?
4. Java более придирчив к типам, чем его предок, язык программирования C. Каковы преимущества более придирчивого отношения к типам вместо автоматического их преобразования по требованию?

Глава 8

Проверка типов в массивах, вызовах методов и доступах к структурам

Это вторая из двух глав, посвященных проверке типов. В предыдущей главе была представлена проверка типов для встроенных атомарных типов. Для сравнения в этой главе будут рассмотрены операции проверки более сложных типов.

Здесь вы узнаете, как выполнять проверку типов для массивов, параметров и возвращать типы вызовов методов в подмножестве Jzero языка Java. Кроме того, она включает в себя проверку составных типов, таких как классы.

В этой главе мы рассмотрим следующие основные темы:

- проверку типов массивов;
- проверку вызовов методов;
- проверку доступа к структурированным типам.

К концу главы вы сможете писать более сложные обходы деревьев для проверки типов, которые сами содержат один или несколько других типов. Возможность поддержки таких составных типов в вашем языке программирования необходима для того, чтобы выйти за рамки игрушечных языков программирования и перейти в сферу языков, полезных в реальном мире. Пришло время узнать больше о проверке типов. Мы начнем с самого простого составного типа – массивов.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/.PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch8*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/30w1V8I*.

Операции проверки типов массивов

Массив – это последовательность элементов, которые имеют одинаковый тип. До этого момента язык Jzero реально не поддерживал типы массивов,

кроме как для того, чтобы обеспечить синтаксис, достаточный для объявления *main()* своего массива параметров *String*. Теперь пришло время добавить поддержку для оставшихся операций с массивами Jzero, которые являются небольшим подмножеством того, что могут делать массивы Java. Массивы Jzero ограничены одноразмерными массивами, созданными без инициализаторов. Чтобы правильно проверять операции с массивами, мы изменим код из предыдущих глав так, чтобы можно было распознавать переменные массива, когда они объявлены, а затем проверять все операции с этими массивами, чтобы разрешить только законные операции. Начнем с объявления переменных массивов.

Управление объявлениями переменных в массивах

Понятие о том, что переменная будет содержать ссылку на массив, прикреплено к типу переменной в правиле рекурсивной грамматики для нетерминала `VarDeclarator` в файле *j0gram.y*. Речь идет о втором правиле производства, которое появляется после вертикальной черты следующим образом:

```
VarDeclarator: IDENTIFIER | VarDeclarator '[' ']' {
        $$=j0.node("VarDeclarator",1060,$1); };
```

Для этого правила соответствующий код в методе `assigntype()` дерева класса добавляет `arraytype()` поверх наследуемого типа, так как `assigntype()` обращается рекурсивно к дочернему узлу `VarDeclarator`. Для этого код Unicon в файле *tree.icn* выглядит следующим образом:

```
method assigntype(t)
      .   .   .
              "VarDeclarator": {
                      kids[1].assigntype(arraytype(t))
                      return
              }
```

Наследуемый тип `t` не отбрасывается. Он становится типом элемента типа массива, который строится здесь. Соответствующий Java-код в файле *tree.java* практически аналогичен:

```
void assigntype(typeinfo t) {
      .   .   .
      case "VarDeclarator": {
              kids[0].assigntype(new arraytype(t));
              return;
      }
```

Поскольку этот код является рекурсивным, он работает для многомерных массивов, представленных цепочкой узлов `VarDeclarator` в дереве синтаксиса, хотя для краткости остальная часть Jzero не будет работать. Даже для одномерных массивов все становится интересным, когда вы рассматриваете проверку информации о типах при использовании массивов в исполняемом коде. Пер-

вое место в коде, где вам потребуется проверить типы массивов, – это момент создания массива.

Проверка типов при создании массива

Массивы в Java создаются с помощью выражения *new*. Это то, что до сих пор было упущено в Jzero. Для этого в файл *javalex.l* добавляется новый токен для зарезервированного слова `new`, как показано в следующем коде:

```
"new"        { return j0.scan(parser.NEW); }
```

Кроме того, это влечет за собой новый вид первичного выражения, называемый выражением `ArrayCreation`. Оно добавляется в грамматику в файле *j0gram.y*, как показано в следующем коде:

```
Primary: Literal | FieldAccess | MethodCall |
        '(' Expr ')' { $$=$2;} | ArrayCreation ;
ArrayCreation: NEW Type '[' Expression ']' {
        $$=j0.node("ArrayCreation", 1260, $2, $4); };
```

После добавления нового зарезервированного слова и определения узла дерева для него пришло время рассмотреть, как назначается тип для этого выражения. Рассмотрим создание массива в выражении Java `new int` [3]. Токен `int` используется в исполняемом выражении впервые, и первоначально код, создающий токен `int` в файле *token.icn*, должен выделить его тип следующим образом:

```
class token(cat, text, lineno, colno, ival, dval, sval,
     typ)
     .   .   .
initially
     case cat of {
            parser.INT:     typ := typeinfo("int")
            parser.DOUBLE:  typ := typeinfo("double")
            parser.BOOLEAN: typ := typeinfo("boolean")
            parser.VOID:    typ := typeinfo("void")
```

Как вы видите, такие же дополнения необходимы и для других атомарных скалярных типов. Соответствующий Java-код в конструкторе в файле *token.java* показан ниже:

```
case parser.INT: typ = new typeinfo(«int»); break;
case parser.DOUBLE: typ = new typeinfo(«double»);
  break;
case parser.BOOLEAN: typ = new typeinfo("boolean");
  break;
case parser.VOID: typ = new typeinfo("void"); break;
```

Эти дополнения к токену класса заботятся о листьях, которые предоставляют наши базовые типы. Тип узла `ArrayCreation` вычисляется с помощью до-

полнения к методу checktype(). Дополнение к checktype() в файле *tree.icn*, состоящее в основном из обращения к arraytype(), показано здесь:

```
class tree (id,sym,rule,nkids,tok,kids,isConst,stab,typ)
      .   .   .
      method checktype(in_codeblock)
      .   .   .
              "ArrayCreation": typ := arraytype(kids[1].typ)
```

Соответствующий этому Java-код в файле *tree.java* выглядит следующим образом:

```
case «ArrayCreation»:
        typ = new arraytype(kids[0].typ); break;
```

Поэтому, когда используется вновь созданный массив, как правило, при присваивании его тип массива должен соответствовать типу, который допускается окружающим выражением. Например, в следующих двух строках оператор присваивания во второй строке должен разрешать массивы, когда проверяется его тип:

```
int x[];
x = new int[3];
```

Код, позволяющий присваивать переменную массива из значения массива, добавляется в метод check_types() файла *tree.icn*, как показано здесь:

```
method check_types(op1, op2)
      .   .   .
      else if (op1.basetype===op2.basetype==="array") &
               operator==="=" &
               check_types(op1.element_type,
                      op2.element_type) then {
               return op1
               }
```

Код проверяет, что op1 и op2 являются массивами, что мы выполняем присваивание и что типы элементов ОК. Здесь оператор write() в части then может быть полезным для целей тестирования кода этой главы. Однако в компиляторе будут показаны только ошибки типа. Соответствующее дополнение Java к методу check_types() в файле *tree.java* выглядит следующим образом:

```
      else if (op1.basetype.equals(«array») &&
                      op2.basetype.equals("array") &&
                      operator.equals("=") &&
                      (check_types(((arraytype)op1).element_type,
                              ((arraytype)op2).element_type) !=
                                      null)) {
                      return op1;
              }
```

Из примеров, приведенных в этом разделе, может показаться, что проверка типов – это просто придирчивое внимание к деталям. Рекурсивный вызов `check_types()` для типов элементов массивов предотвращает, например, случайное присвоение программой типа *array of string* переменной типа *array of int*. Теперь пришло время рассмотреть проверку типов для доступа к элементам массива.

Проверка типов при обращении к массиву

Обращение к массиву состоит из операций чтения и записи элементов массива с использованием оператора индекса. Здесь нам необходимо добавить поддержку синтаксиса для этих операций и построить узлы дерева синтаксиса, прежде чем мы сможем выполнить проверку типов. Добавление обращения к массивам в грамматику заключается в добавлении нетерминала `ArrayAccess`, а затем в добавлении двух правил производства, которые используют этот нетерминальный символ:

- одно для присваиваний, которые сохраняют значение в элементе массива;
- одно для выражений, которые извлекают значение из элемента массива.

Изменения в файле *j0gram.y* будут выглядеть следующим образом. Их грамматика была переупорядочена для ясности:

```
ArrayAccess: Name '[' Expr ']' {
        $$=j0.node("ArrayAccess",1390,$1,$3); };
LeftHandSide: Name | FieldAccess | ArrayAccess ;
Primary: Literal | FieldAccess | MethodCall | ArrayAccess
             |'(' Expr ')' { $$=$2;} | ArrayCreation ;
```

Оператор в квадратных скобках, который используется для доступа к элементам массива, должен проверять типы своих операндов и использовать их для вычисления типа результата. Результат индекса массива удаляет один уровень массива из типа левого операнда, тем самым создавая тип его элемента. Дополнение к методу `checktype()` в файле *tree.icn* выглядит следующим образом:

```
method checktype(in_codeblock)
     .   .   .
             "ArrayAccess": {
                     if match("array ", kids[1].typ.str()) then {
                             if kids[2].typ.str()=="int" then
                                     typ := kids[1].typ.element_type
                             else stop("subscripting array with ",
                                       kids[2].typ.str())
                             }
                     else stop("illegal subscript on type ",
                               kids[1].typ.str())
                     }
```

Приведенный выше код проверяет, что тип `kids[1]` является типом массива, а тип `kids[2]` является типом целого числа. Если все в порядке, то значением, присвоенным `typ` этого узла, будет `element_type` массива. Соответствующее Java-дополнение к методу `checktype()` в файле *tree.java* показано здесь:

```
case «ArrayAccess»:
if (kids[0].typ.str().startsWith(«array «)) {
     if (kids[1].typ.str().equals("int"))
     typ = ((arraytype)(kids[0].typ)).element_type;
     else j0.semerror("subscripting array with " +
                      kids[1].typ.str());
     }
else j0.semerror("illegal subscript on type " +
                 kids[0].typ.str());
break;
```

В этом разделе мы показали, как проверять типы массивов. К счастью, нетерминальные символы в грамматике, а значит, и в дереве синтаксиса, позволяют легко найти места, где необходима такая форма проверки типов. Теперь пришло время рассмотреть, возможно, самую сложную часть проверки типов. Далее мы узнаем, как проверять параметры и возвращаемые типы вызовов методов.

Проверка вызовов методов

Вызов функции – это фундаментальный строительный блок парадигм как императивного, так и функционального программирования. В объектно-ориентированных языках функции называются **методами**, но они могут играть все те же роли, что и функции. Кроме того, набор методов обеспечивает публичный интерфейс объекта. Чтобы проверить тип вызова метода, необходимо проверить как количество, так и тип параметров вместе с возвращаемым типом.

Вычисление параметров и информации о возвращаемом типе

Представление типа, введенное в предыдущей *главе 7 «Проверка базовых типов»*, включало класс *methodtype*, который имел поля для параметров и возвращаемого типа. Однако мы еще не представили код для извлечения этой информации из дерева синтаксиса и внесения ее в тип. Параметры и возвращаемый тип метода называются его **сигнатурой**. Правило грамматики, в котором объявляется сигнатура метода, строит узел `MethodHeader`. Чтобы вычислить возвращаемый тип, нам нужно синтезировать его из узла `MethodReturnVal`. Чтобы вычислить параметры, нам нужно перейти к поддереву *FormalParmList* в `MethodDeclarator`. Это можно сделать, добавив вызов `j0.calctype()` в правило грамматики для `MethodHeader`. Это аналогично тому, что мы добавили ранее для объявлений переменных:

```
MethodHeader: PUBLIC STATIC MethodReturnVal
              MethodDeclarator {
      $$=j0.node("MethodHeader",1070,$3,$4);
      j0.calctype($$);
      };
```

Метод `calctype()` в классе `j0` не был изменен, но методы, которые он вызывает в *tree.icn*, были расширены для добавления дополнительной информации о типе, по мере необходимости, для обработки сигнатур методов. Метод `calctype()` в классе *tree* получил небольшое усовершенствование, чтобы синтезировать тип листа из содержащегося в нем типа токена, если таковой имеется. В Unicon это следующая строка, добавленная в файл *tree.icn*, которая присваивает тип из `tok.typ`:

```
method calctype()
      .   .   .
      "token": {
              if typ := \ (tok.typ) then return
```

Соответствующее Java-дополнение к функции `calctype()` в файле *tree.java* показано здесь:

```
              if ((typ = tok.typ) != null) return;
```

Модификации метода `assigntype()` для построения сигнатур методов являются более существенными. Для объявлений переменных вы просто передаете тип как наследуемый атрибут вниз по списку к идентификаторам листьев отдельных переменных. Для метода тип, который будет связан с идентификатором, строится из наследуемого атрибута, который является возвращаемым типом, плюс остаток сигнатуры метода, полученный из поддерева, связанного со списком параметров:

```
method assigntype(t)
      case sym of {
      .   .   .
              "MethodDeclarator": {
                      parmList := (\ (kids[2]).mksig()) | []
                      kids[1].typ := typ := methodtype(parmList , t)
              return
      }
```

В этом коде список параметров `parmList` строится как список типов. Если список параметров не пуст, он строится путем вызова метода `mksignature()` на непустом узле дерева. Если список параметров пуст, то `parmList` инициализируется как пустой список `[]`. Список параметров и возвращаемый тип `t` передаются для построения типа метода, который присваивается узлу `MethodDeclarator` и его первому дочернему узлу, то есть идентификатор имени метода будет вставлен в таблицу символов класса. Соответствующее дополнение Java к методу `assigntype()` в файле *tree.java* показано здесь:

```
Case «MethodDeclarator»:
typeinfo parmList[];
      if (kids[1] != null) parmList = kids[1].mksig();
      else parmList = new typeinfo [0];
      kids[0].typ = typ = new methodtype(parmList , t);
      return;
```

Метод `mksig()` строит список типов параметров метода. Этот метод является примером очень специализированного метода дерева. Он представляет собой обход поддерева, который обходит только очень узкое подмножество всех узлов дерева. Он вызывается только для формального списка параметров и должен рассматривать по мере продвижения по списку параметров лишь узлы `FormalParmList` и `FormalParm`, подбирая типы каждого параметра. Код Unicon для `mksig()` в файле *tree.icn* выглядит следующим образом:

```
method mksig()
      case sym of {
              "FormalParm": return [ kids[1].typ ]
              "FormalParmList":
                      return kids[1].mksig() ||| kids[2].mksig()
              }
end
```

Случай `FormalParm` возвращает список размером 1. Случай `FormalParmList` возвращает конкатенацию двух рекурсивных вызовов его дочерних элементов. Соответствующий Java-код в файле *tree.java* показан здесь:

```
typeinfo [] mksig() {
      switch (sym) {
      case "FormalParm": return new typeinfo[]{kids[0].typ};
      case "FormalParmList":
              typeinfo ta1[] = kids[0].mksig();
              typeinfo ta2[] = kids[1].mksig();
              typeinfo ta[] = new typeinfo[ta1.length +
                              ta2.length];
              for(int i=0; i<ta1.length; i++) ta[i]=ta1[i];
              for(int j=0; j<ta2.length; j++)
                      ta[ta1.length+j]=ta2[j];
              return ta;
      }
      return null;
}
```

В реализации Java используются массивы. Большая часть приведенного выше кода конкатенирует два массива, возвращенных после вызова `mksig()` для дочерних элементов. Это объединение может быть выполнено путем импорта *java.util.Arrays* и использования методов утилиты, но код *Arrays* не будет намного короче или понятнее. Есть еще одно последнее изменение в коде, которое необходимо, чтобы связать всю эту информацию о типах методов и сделать ее пригодной для использования. Когда метод вставляется в таблицу

символов в методе `populateSymTables()`, в нем должна быть сохранена информация о его типе. В Unicon изменение в файле *tree.icn* показано здесь:

```
method populateSymTables()
      case sym of {
      .   .   .
      "MethodDecl": {
              stab.insert(kids[1].kids[2].kids[1].tok.text, ,
                          kids[1].stab, kids[1].kids[2].typ)
```

По сравнению с предыдущими главами, добавление информации о типе – это всего лишь один дополнительный параметр, передаваемый в метод `insert()` таблицы символов. Соответствующий Java-код в файле *tree.java* показан здесь:

```
stab.insert(s, false, kids[0].stab, kids[0].kids[1].typ);
```

Мы создали информацию о типе для методов при их объявлении и сделали эту информацию доступной в таблице символов. Теперь давайте рассмотрим, как использовать информацию о типах различных методов для проверки типов фактических параметров, когда они вызываются.

Проверка типов в каждом месте вызова метода

Места вызова методов можно найти в дереве синтаксиса, отыскав два правила производства, которые создают нетерминал `MethodCall`. Правило, в котором вызов метода представляет собой имя, за которым следует заключенный в круглые скобки список из нуля или более параметров, показано ниже. Оно включает классический синтаксис функции, который в основном используется для вызова методов внутри одного класса, а также квалифицированные имена с синтаксисом *object.function* для вызова метода внутри другого класса. Этот раздел посвящен проверке типов для классического синтаксиса функций. Синтаксис *object.function* рассматривается в разделе *«Проверка доступа к структурированным типам»*. Приведенный здесь код был изменен в этом разделе.

Код для проверки типов вызовов методов добавлен в метод `checktype()`. Дополнения Unicon к файлу *tree.icn* выглядят следующим образом:

```
Method checktype(in_codeblock)
      .   .   .
      "MethodCall": {
              if rule = 1290 then {
                      if kids[1].sym ~== "token" then
                              stop("can't check type of Name ", kids[1].sym)
                      if kids[1].tok.cat == parser.IDENTIFIER then {
                              if (\(rv:=stab.lookup(kids[1].tok.text))) then {
                                      rv := rv.typ
                                      if not match("method ", rv.str()) then
                                              stop("method expected, got ",
```

```
                                        rv.str())
                                        cksig(rv)
                                }
                        }
                else stop("can't typecheck token ", kids[1].tok.cat)
        }
        else stop("Jzero does not handle complex calls")
}
.   .   .
```

В приведенном выше коде метод ищется в таблице символов, а его тип извлекается. Если параметров нет, тип проверяется на то, что его список параметров пуст. Если в вызове есть фактические параметры, они проверяются на соответствие формальным параметрам посредством вызова метода `cksig()`. Если проверка прошла успешно, то полю `typ` для этого узла присваивается тип *return_type*, который был указан для вызванного метода. Соответствующий Java-код в файле *tree.java* показан здесь:

```
case «MethodCall»:
     if (rule == 1290) {
             symtab_entry rve;
             methodtype rv;
             if (!kids[0].sym.equals("token"))
                     j0.semerror("can't check type of " + kids[0].sym);
             if (kids[0].tok.cat == parser.IDENTIFIER) {
                     if ((rve = stab.lookup(kids[0].tok.text)) != null){
                     if (! (rve.typ instanceof methodtype))
                             j0.semerror("method expected, got " +
                                         rv.str());
                     rv = (methodtype)rve.typ;
                     cksig(rv);
             }
     }
     else j0.semerror("can't typecheck " + kids[0].tok.cat);
     }
else j0.semerror("Jzero does not handle complex calls");
break;
```

Метод, который используется для проверки сигнатуры функции и применения ее возвращаемого типа, – это метод `cksig()`. Unicon-реализация метода `cksig()` в файле *tree.icn* показана здесь:

```
method cksig(sig)
local i:=*sig.parameters, nactual := 1, t := kids[2]
     if /t then {
             if i ~= 0 then stop("0 parameters, expected ", i)
             }
     else {
             while t.sym == "ArgList" do {
                     nactual +:= 1; t:=t.kids[1] }
```

```
                if nactual ~= i then
                        stop(nactual " parameters, expected ", i)
                t := kids[2]
                while t.sym == "ArgList" do {
                        check_types(t.kids[-1].typ, sig.parameters[i])
                        t := t.kids[1]; i-:=1
                        }
                check_types(t.typ, sig.parameters[1])
        }
        typ := sig.return_type
end
```

Этот метод сначала обрабатывает нулевые параметры как особый случай, и кроме того, он проверяет по одному параметру за раз в цикле `while`. Для каждого параметра он вызывает *ckarg()* для проверки формального и фактического типов. Из-за способа построения дерева синтаксиса параметры во время обхода дерева встречаются в обратном порядке. Первый параметр встречается, когда вы попадаете на узел дерева, который не является `ArgList`. После обработки аргументов `cksig()` устанавливает тип узла *MethodCall* в тип, возвращаемый методом. Соответствующий Java-код в файле *tree.java* выглядит следующим образом:

```
void cksig(methodtype sig) {
        int i = sig.parameters.length, nactual = 1;
        tree t = kids[1];
        if (t == null) {
                if (i != 0) j0.semerror("0 params, expected ",i);
        }
        else {
                while (t.sym.equals("ArgList")){nactual++;
                t=t.kids[0];}
                if (nactual != i)
                        j0.semerror(nactual + " parameters, expected "+ i);
                t = kids[1];
                i--;
                while (t.sym.equals("ArgList")) {
                        check_types(t.kids[1].typ, sig.parameters[i--]);
                        t = t.kids[0];
                }
        check_types(t.typ, sig.parameters[0]);
}
          typ = sig.return_type;
}
```

Метод `check_types()` и его вспомогательный метод `get_op()` должны быть изменены для управления проверкой типов параметров. Реализация этих изменений в Unicon выглядит следующим образом:

```
method get_op()
      return case sym of { …
              "MethodCall" : "param"
.   .   .
method check_types(op1, op2)
      operator := get_op()
      case operator of {
              "param"|"return"|"="|"+"|"-" : {
              .   .   .
```

Соответствующие изменения в Java для `get_op()` и `check_types()` в файле *tree.java* следующие:

```
public String get_op() {
      switch (sym) {
      case "MethodCall" : return "param";
      .   .   .
public typeinfo check_types(typeinfo op1, typeinfo op2) {
      String operator = get_op();
      switch (operator) {
      case "param": case "return": case "=": case "+":
      case"-":
```

Итак, вы узнали, как проверять типы параметров, передаваемых в вызовы методов, что является одним из самых сложных аспектов проверки типов. Теперь пришло время проверить возвращаемые типы, которые выводятся из вызова функции с помощью ее операторов возврата.

Проверка типов в операторах возврата

Тип выражений в операторах возврата метода должен соответствовать объявленному возвращаемому типу. Эти два элемента находятся на довольно большом расстоянии друг от друга в дереве синтаксиса. Существует множество различных способов, которыми вы можете их соединить. Например, можно добавить атрибут return_type ко всем узлам дерева и унаследовать тип заголовка метода (*MethodHeader*) в блок (*Block*) и далее по коду в операторы возврата. Однако такой подход является пустой тратой времени для относительно редко используемой части информации. Таблица символов – самый удобный способ соединения удаленных объектов. Мы можем вставить фиктивный символ в таблицу символов, который будет содержать тип возвращаемого значения функции. Этот фиктивный символ можно искать и сверять с типом в каждом операторе возврата. Идеальным вариантом является фиктивный символ с именем `return`. Его легко запомнить, и это зарезервированное слово, которое никогда не будет конфликтовать с реальным символом в пользовательском коде. Код для вставки типа `return` в таблицу символов метода является дополнением к методу `populateSymTables()`. Реализация Unicon в файле *tree.icn* выглядит следующим образом:

```
method populateSymTables()
case sym of {
.   .   .
     "MethodDecl": {
            stab.insert(kids[1].kids[2].kids[1].tok.text, ,
                        kids[1].stab, kids[1].kids[2].typ)
            kids[1].stab.insert("return", , ,
                    kids[1].kids[1].typ)
            }
```

В этом коде `kids[1]` является узлом *MethodHeader*. Его поле `stab` является локальной таблицей символов, вставляемой в качестве подобласти видимости внутрь охватывающей области видимости класса. Выражение `kids[1].kids[1]` – это узел *MethodReturnVal*, который обычно представляет собой просто токен, обозначающий возвращаемый тип. Пара пробелов, разделенных запятыми, между `"return"` и типом – это пустые значения. Они передаются во второй и третий параметры таблицы символов `insert()`. Соответствующий код Java, который добавляется в метод `populateSymTables()` файла *tree.java*, выглядит следующим образом:

```
Kids[0].stab.insert(«return», false, null,
                    kids[0].kids[0].typ);
```

Код проверки типа, который использует эту информацию о возвращаемом типе в операторе возврата, добавляется к методу `checktype()`, который также находится в классе дерева. Реализация Unicon в файле *tree.icn* выглядит следующим образом:

```
Method checktype(in_codeblock)
.   .   .
     case sym of {
     "ReturnStmt": {
            if not (rt := ( \ ( \ stab).lookup("return")).typ)
                        then
                    stop("stab did not find a returntype")
            if \ (kids[1].typ) then
                    typ := check_types(rt, kids[1].typ)
            else {
                    if rt.str() ~== "void" then
                            stop("void return from non-void method")
                    typ = rt;
                    }
     }
```

Соответствующий Java-код представлен здесь:

```
Case «ReturnStmt»:
     symtab_entry ste;
     if ((ste=stab.lookup("return")) == null)
            j0.semerror("stab did not find a returntype");
```

```
typeinfo rt = ste.typ;
if (kids[0].typ != null)
        typ = check_types(rt, kids[0].typ);
else {
        if (!rt.str().equals("void"))
        j0.semerror("void return from non-void method");
typ = rt;
        }
break;
```

Итак, вы узнали, как проверять операторы *return*. Теперь пришло время узнать, как проверять обращения к полям и методам экземпляра класса.

Проверка обращений к структурированным типам

В этой книге термин «**структурированный тип**» будет обозначать составные объекты, которые могут содержать смесь типов, доступ к элементам которых осуществляется по имени. Это отличается от массивов, доступ к элементам которых осуществляется по их позиции и элементы которых относятся к одному типу. В некоторых языках для такого рода данных существуют типы структура (struct) или запись (record). В Jzero и большинстве объектно-ориентированных языков в качестве основного структурированного типа используются классы.

В этом разделе рассматриваются аспекты того, как проверять типы для операций над классами и, более конкретно, экземплярами класса. Эта организация отражает представление типов массивов в начале данной главы, начиная с того, что необходимо для обработки объявлений переменных класса.

Первоначальным замыслом языка Jzero была поддержка крошечного подмножества Java, которое было бы в некоторой степени сравнимо с языком PL/0 Вирта. Такой язык не требует экземпляров класса или объектно-ориентированного подхода, и ограниченное пространство не позволяет охватить многие нюансы, необходимые для многофункционального объектно-ориентированного языка, такого как Java или C++. Тем не менее мы представим некоторые из них. Первое, что необходимо рассмотреть, – это как объявлять переменные экземпляра для типов классов.

Обработка объявлений переменных экземпляра

Переменные типа класса объявляются путем задания имени класса и затем разделенного запятыми списка из одного или нескольких идентификаторов. Например, наш компилятор должен обрабатывать объявления, которые похожи на следующее объявление трех строк:

```
String a, b, c;
```

Jzero должен был обрабатывать такие объявления с самого начала, поскольку процедура *main()* принимает массив строк. Хотя наш компилятор Jzero уже поддерживает объявления переменных класса, необходимо принять во внимание несколько дополнительных соображений.

Во многих объектно-ориентированных языках объявления переменных имеют сопутствующие правила видимости, такие как *public* и *private*. В Jzero все методы являются публичными (`public`), а все переменные – приватными (`private`), но вы можете пойти дальше и реализовать атрибут *isPublic* в любом случае. Аналогичное соображение относится к статическим (`static`) переменным. В Jzero нет статических переменных, но вы можете реализовать атрибут *isStatic*, если решите, что они вам нужны. Расширение нашего примера с учетом этих двух соображений будет выглядеть следующим образом:

```
private static String a, b, c;
```

Для поддержки этих атрибутов Java можно добавить их к токенам, узлам дерева и типу записи таблицы символов. Вы можете распространить их из зарезервированного слова туда, где объявлены переменные, точно так же, как мы это делали для информации о типе.

Проверка типов при создании экземпляра

Объекты, также называемые **экземплярами классов**, создаются с помощью зарезервированного слова *new*, как и для массивов, о чем мы говорили в разделе «*Операции проверки типов массивов*». Дополнения к грамматике в файле *j0gram.y* показаны здесь:

```
Primary: Literal | FieldAccess | MethodCall | ArrayAccess
             |'(' Expr ')' { $$=$2;} | ArrayCreation |
                    InstanceCreation;
InstanceCreation: NEW Name '(' ArgListOpt ')' {
       $$=j0.node("InstanceCreation", 1261, $2, $4); };
```

Этот добавленный синтаксис позволяет создавать экземпляры. Для вычисления типа выражения, чтобы его можно было проверить, нам нужно найти тип класса в таблице символов. Чтобы это сработало, в более ранний момент времени мы должны создать соответствующий объект *classtype* и связать его с именем класса во вложенной таблице символов.

Вместо того чтобы встраивать код для построения типа класса при обходе поддеревьев во время парсинга, как мы делали в предыдущих разделах для построения сигнатуры метода, для класса проще подождать до окончания парсинга и заполнения таблицы символов, то есть делать это непосредственно перед проверкой типов. Таким образом, вся информация для построения типа класса будет готова в таблице символов класса. Вызов нового метода `mkcls()` добавлен после обработки таблицы символов и перед проверкой типов к методу `semantic()` в файле *j0.icn* следующим образом:

```
method semantic(root)
.   .   .
     root.checkSymTables()
     root.mkcls()
     root.checktype()
```

Соответствующее Java-дополнение в файл *j0.java* показано здесь:

```
root.mkcls();
```

Метод `mkcls()` означает «создать класс» (*make class*). Когда он видит объявление класса, он ищет имя класса и просматривает таблицу символов класса, помещая записи в нужную категорию. Есть один список для полей, один для методов и один для конструкторов. Здесь показана Unicon- реализация `mkcls()` в файле *tree.icn*:

```
method mkcls()
      if sym == "ClassDecl" then {
              rv := stab.lookup(kids[1].tok.text)
              flds := []; methds := []; constrs := []
              every k := key(rv.st.t) do
                      if match("method ", rv.st.t[k].typ.str()) then
                              put(methds, [k, rv.st.t[k].typ])
                      else put(flds, [k, rv.st.t[k].typ])
              /(rv.typ) := classtype(kids[1].tok.text, rv.st,
                                      flds, methds, constrs)
      }
      else every k := !kids do
              if k.nkids>0 then k.mkcls()
end
```

Когда этот обход встречает объявление класса, он просматривает имя класса и получает таблицу символов для этого класса. Каждый символ проверяется, и если это метод, то он попадает в список методов с именем `methds`, в противном случае он переходит в список полей с именем `flds`. Типу класса в записи таблицы символов класса присваивается экземпляр типа класса, который содержит всю эту информацию. Вы можете заметить, что конструкторы не идентифицируются и не помещаются в список конструкторов. Это нормально для Jzero – не поддерживать конструкторы, но более крупное подмножество языка Java будет поддерживать по крайней мере один конструктор для каждого класса. Соответствующая Java-версия для любого случая показана далее:

```
void mkcls() {
      symtab_entry rv;
      if (sym.equals("ClassDecl")) {
              int ms=0, fs=0;
              rv = stab.lookup(kids[0].tok.text);
              for(String k : rv.st.t.keySet()) {
                      symtab_entry ste = rv.st.t.get(k);
                      if ((ste.typ.str()).startsWith("method ")) ms++;
                      else fs++;
              }
              parameter flds[] = new parameter[fs];
              parameter methds[] = new parameter[ms];
              fs=0; ms=0;
              for(String k : rv.st.t.keySet()) {
```

```
                symtab_entry ste = rv.st.t.get(k);
                if ((ste.typ.str()).startsWith("method "))
                        methds[ms++] = new parameter(k, ste.typ);
                else flds[fs++] = new parameter(k, ste.typ);
                }
                rv.typ = new classtype(kids[0].tok.text,
                                rv.st, flds, methds, new typeinfo[0]);
        }
        else for(int i = 0; i<nkids; i++)
                if (kids[i].nkids>0) kids[i].mkcls();
}
```

Есть еще один фрагмент кода, который необходим для завершения обработки создания экземпляра. Поле типа должно быть установлено для узлов `InstanceCreation` в методе `checktype()`. После всей работы по размещению информации о классе в таблице символов это – простой поиск. Реализация Unicon в файле *tree.icn* выглядит следующим образом:

```
method checktype(in_codeblock)
.   .   .
        "InstanceCreation": {
                if not (rv := stab.lookup(kids[1].tok.text)) then
                        stop("unknown type ",kids[1].tok.text)
                if not (typ := \ (rv.typ)) then
                        stop(kids[1].tok.text, " has unknown type")
        }
```

Приведенный выше код – это просто *поиск в таблице символов*, который включает в себя выборку типа из записи в таблице символов плюс множество проверок на ошибки. Соответствующие дополнения Java в файле *tree.java* выглядят следующим образом:

```
case «InstanceCreation»:
        symtab_entry rv;
        if ((rv = stab.lookup(kids[0].tok.text))==null)
                j0.semerror("unknown type " + kids[0].tok.text);
        if ((typ = rv.typ) == null)
                j0.semerror(kids[0].tok.text + " has unknown type");
        break;
```

Итак, вы узнали, как создавать информацию о типе для классов и использовать ее для получения корректного типа при создании экземпляра. Теперь рассмотрим, что потребуется для поддержки обращения к именам, определенным внутри экземпляра.

Проверка типов при обращении к экземпляру

Обращения к экземпляру относятся к ссылкам на поля и методы объекта. Существуют неявные обращения, когда на поле или метод текущего объекта ссылаются непосредственно по имени, и явные обращения, когда оператор точки

используется для доступа к объекту через его публичный интерфейс. Неявные обращения обрабатываются обычным поиском в таблице символов текущей области видимости, которая автоматически пытается охватить области видимости, включая область видимости класса, где находятся методы и переменные класса текущего объекта. Этот раздел посвящен явным обращениям с использованием оператора точка. В грамматике *j0gram.y* они называются узлами квалифицированных имен (`QualifiedName`). Добавление поддержки квалифицированных имен начинается с изменения кода `MethodCall` в методе `checktype()` дерева класса. Код, представленный ранее в этой главе для проверки сигнатуры метода на простых именах, помещается в предложение `else`. Реализация Unicon в файле *tree.icn* добавляет следующие строки:

```
method checktype(in_codeblock)
      .   .   .
              "MethodCall": {
                      if rule = 1290 then {
                              if kids[1].sym == "QualifiedName" then {
                                      rv := kids[1].dequalify()
                                      cksig(rv)
                                      }
                              else {
                                      if kids[1].sym ~== "token" then
                                      …
                                      else stop("can't check type of ",
                                                  kids[1].tok.cat)
                              }
```

Код в `checktype()` распознает квалифицированные имена, когда они используются в качестве имени вызываемого метода, и вызывает метод `dequalify()`, чтобы получить тип точечного имени. Затем он использует метод проверки сигнатуры `cksig()`, представленный ранее, для проверки типов при вызове. Соответствующий Java-код в файле *tree.java* выглядит следующим образом:

```
if (kids[0].sym.equals(«QualifiedName»)) {
        rv = (methodtype)(kids[0].dequalify());
        cksig(rv);
        }
```

`kids[0]` – это узел дерева с двумя дочерними элементами. Тип левого дочернего узла содержит таблицу символов, в которой мы ищем правого потомка, чтобы найти тип его метода. Метод `dequalify()` делает эту грязную работу. Реализация Unicon в файле *tree.icn* выглядит следующим образом:

```
method dequalify()
local rv, ste
     if kids[1].sym == "QualifiedName" then
             rv := kids[1].dequalify()
     else if kids[1].sym=="token" &
              kids[1].tok.cat=parser.IDENTIFIER then {
```

```
                if not \ (rv := stab.lookup(kids[1].tok.text)) then
                        stop("unknown symbol ", kids[1].tok.text)
                rv := rv.typ
        }
        else stop("can't dequalify ", sym)
        if rv.basetype ~== "class" then
                stop("can't dequality ", rv.basetype)
        if \ (ste := rv.st.lookup(kids[2].tok.text)) then
                return ste.typ
        else stop(kids[2].tok.text, " is not in ", rv.str())
end
```

Этот метод сначала вычисляет тип для операнда левой стороны. Это требует рекурсии, если левый операнд является другим квалифицированным именем. В противном случае левый операнд должен быть идентификатором, который можно найти в таблице символов. В любом случае проверяется, является ли классом тип левого операнда, и если да, то идентификатор справа от точки ищется в этом классе, и возвращается его тип. Соответствующая Java-реализация показана здесь:

```
public typeinfo dequalify() {
      typeinfo rv = null;
      symtab_entry ste;
      if (kids[0].sym.equals("QualifiedName"))
              rv = kids[0].dequalify();
      else if (kids[0].sym.equals("token") &
           (kids[0].tok.cat==parser.IDENTIFIER)) {
      if ((ste = stab.lookup(kids[0].tok.text)) != null)
              j0.semerror("unknown symbol " + kids[0].tok.text);
      rv = ste.typ;
      }
      else j0.semerror("can't dequalify " + sym);
      if (!rv.basetype.equals("class"))
              j0.semerror("can't dequalify " + rv.basetype);
      ste = ((classtype)rv).st.lookup(kids[1].tok.text);
      if (ste != null) return ste.typ;
      j0.semerror("couldn't lookup " + kids[1].tok.text +
              " in " + rv.str());
      return null;
}
```

В этом разделе вы узнали, как обрабатывать обращения к структурам. Мы рассмотрели проверку типов, в которой объявляются и создаются переменные типа класса. Затем вы узнали, как вычислять типы квалифицированных имен внутри объектов.

После всей этой проверки типов вывод, опять же, немного разочаровывающий. Вы можете скачать код с сайта книги на Github, перейти в подкаталог *Chapter08/* и собрать его с помощью программы *make*. Это позволит создать как версию для Unicon, так и версию для Java. Напоминаю, что вам придется настроить установленное программное обеспечение и/или установить свой

путь к каталогу, куда вы распаковали примеры из книги, как это обсуждалось в главах от второй, «*Проектирование языка программирования*», до пятой, «*Деревья синтаксиса*». Когда вы выполните команду *j0* с установленной проверкой типов, она выдаст результат, похожий на представленный на рис. 8.1.

```
> type funtest.java
public class funtest {
   public static int foo(int x, int y, String z) {
       return 0;
   }
   public static void main(String argv[]) {
      int x;
      x = foo(0,1,"howdy");
      x = x + 1;
      System.out.println("hello, jzero!");
   }
}

> java ch8.j0 funtest.java
line 3: typecheck return on a int and a int -> OK
checking the type of a call to foo
line 7: typecheck param on a String and a String -> OK
line 7: typecheck param on a int and a int -> OK
line 7: typecheck param on a int and a int -> OK
line 7: typecheck = on a int and a int -> OK
line 8: typecheck + on a int and a int -> OK
line 8: typecheck = on a int and a int -> OK
line 9: typecheck param on a String and a String -> OK
no errors
```

Рис. 8.1. Проверка типов параметров и возвращаемых типов

Если в программе нет ошибок типов, все строки будут заканчиваться символом **OK**. В последующих главах Jzero не будет беспокоиться о выводе данных при успешной проверке типов, так что это будет последний раз, когда вы видите такие строки.

Заключение

Эта глава была второй из двух глав, посвященных различным аспектам проверки типов. Вы узнали, как представлять составные типы. Например, вы узнали, как строить сигнатуры методов и использовать их для проверки вызовов методов. Это достигается с помощью обходов дерева синтаксиса, и большая часть этого включает в себя добавление небольших расширений к функциям, представленным в предыдущей главе.

В данной главе вы также узнали, как распознавать объявления массивов и создавать соответствующие представления типов для них. Вы узнали, как проверить, правильные ли типы используются для создания массива и обращения к нему, а также как строить сигнатуры типов для объявлений методов. Вы также узнали, как проверить, что для вызова методов и возврата используются корректные типы.

Хотя написание более сложных функций обхода дерева является ценным навыком само по себе, представление информации о типе и распространение ее по дереву синтаксиса туда, где она необходима, также является отличной тренировкой навыков, которые понадобятся вам на следующих этапах работы с компилятором. Теперь, когда вы реализовали проверку типов, вы готовы перейти к генерации кода. Это означает среднюю точку в реализации вашего языка программирования. До сих пор вы собирали информацию о программе. В следующей главе начинается работа над преобразованным выводом входной программы, начиная с генерации промежуточного кода.

Вопросы

1. Каковы основные различия между проверкой типов обращений к массивам и проверкой типов обращений к членам структур или классов?
2. Откуда операторы возврата функции знают, какой тип они возвращают? Они часто находятся довольно далеко в дереве от того места, где объявлен возвращаемый функцией тип.
3. Как проверяются типы во время вызова функции? Как это соотносится с проверкой типов операторов, таких как плюс и минус?
4. Помимо доступа через операторы «*[ ]*» и «.», какие еще формы проверки типов необходимы для проверки типов массивов, структур или классов?

Глава 9

Генерация промежуточного кода

После завершения семантического анализа вы можете подумать о том, как выполнить программу. Для компиляторов следующим шагом является создание последовательности независимых от машины инструкций, называемых **промежуточным кодом**. За этим обычно следует этап оптимизации и генерация окончательного кода для целевой машины. В этой главе вы узнаете, как генерировать промежуточный код, на примере языка Jzero. После нескольких глав, из которых вы узнали, как писать обходы деревьев, которые анализируют и добавляют информацию в дерево синтаксиса, построенное на основе входных данных, эта глава интересна тем, что обходы деревьев в ней начинают процесс построения выходных данных компилятора.

В этой главе рассматриваются следующие основные темы:

- подготовка к генерированию кода;
- определение набора инструкций промежуточного кода;
- генерация кода для выражений;
- генерация кода для потока управления.

Начнем с понимания того, почему промежуточный код так полезен. Генерацию промежуточного кода можно рассматривать как процесс подготовки к генерации окончательного кода.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/.PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch9*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/ 30t3gNQ*.

Подготовка к генерации кода

Генерация промежуточного кода позволяет получить достаточно информации для выполнения задачи генерации окончательного кода, который может быть запущен. Как и многие другие вещи в жизни, решение сложной задачи становится возможным, когда вы хорошо подготовитесь. Нетерпеливые разработчики могут захотеть пропустить этот этап и сразу перейти к генерации

окончательного кода. Поэтому давайте рассмотрим, почему генерация промежуточного кода так полезна. Генерация окончательного машинного кода – сложная задача, и большинство компиляторов используют промежуточный код, чтобы разбить работу на этапы для ее успешного завершения. В этом разделе мы подробно расскажем, что к чему и почему, начиная с некоторых конкретных технических мотивов для генерации промежуточного кода.

Зачем генерировать промежуточный код?

Цель этой стадии работы вашего компилятора – создать список машинно независимых инструкций для каждого метода в программе. Генерация предварительного машинно-нейтрального кода в качестве промежуточного представления инструкций программы имеет следующие преимущества:

- это позволяет вам определить области памяти и смещения байтов, в которых будут храниться переменные, прежде чем беспокоиться о специфических для машины деталях, таких как регистры и режимы адресации;
- это позволяет вам проработать большинство деталей потока управления, таких как определение того, где потребуются метки и инструкции go-to;
- включение промежуточного кода в компилятор уменьшает размер и область видимости кода, специфичного для процессора, улучшая переносимость компилятора на новые архитектуры;
- она позволяет вам проверить свою работу до этого момента и обеспечивает вывод в человекочитаемом формате, прежде чем мы увязнем в низкоуровневом машинном коде;
- генерация промежуточного кода позволяет применить широкий спектр оптимизаций перед генерацией окончательного кода, специфичного для конкретной машины. Оптимизации, проводимые в промежуточном коде, приносят пользу всем генераторам конечного кода, которые вы используете после этого момента.

Теперь давайте рассмотрим некоторые дополнения к структуре данных, которые помогут сгенерировать промежуточный код.

Изучение областей памяти в созданной программе

В интерпретаторе адрес в пользовательской программе ссылается на память в адресном пространстве интерпретатора, и им можно управлять напрямую. Компилятор сталкивается с более сложной задачей управления адресами, которые являются абстракциями областей памяти в будущих исполнениях созданной программы. Во время компиляции адресное пространство пользовательской программы еще не существует, но когда оно появится, то будет организовано примерно так, как показано на рис. 9.1.

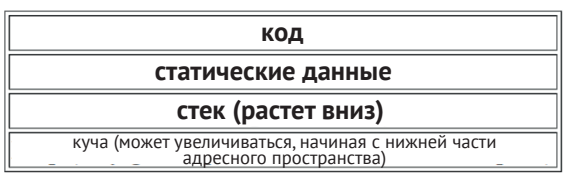


Рис. 9.1. Области оперативной памяти

Некоторые адреса будут находиться в статической памяти, некоторые – в стеке, некоторые – в куче, и некоторые – в коде. В конечном коде способ доступа к этим областям различен, но для адресов промежуточного кода нам просто нужен способ сказать, в какой области находится каждый адрес. Мы могли бы использовать целочисленные коды для представления этих различных областей памяти, но в Unicon и Java строковое имя – это прямой человекочитаемый способ обозначить их. Пусть будет так. Области, которые мы будем использовать, и их интерпретации показаны здесь:

- «loc». В локальной области смещение задается относительно вершины стека. Например, доступ к нему, вероятно, будет осуществляться относительно регистра указателя стекового фрейма;
- «glob». В глобальной области хранятся статически выделенные переменные. Смещение указывается относительно начала некоторой области данных, которая фиксируется во время загрузки. В зависимости от вашего окончательного кода оно может быть преобразовано в абсолютный адрес;
- «const». Область констант содержит статически выделенные значения, доступные только для чтения. За ислючением того, что она доступна лишь для чтения, ее свойства аналогичны свойствам глобальной области. Обычно в ней хранятся строки и другие постоянные структурированные данные. Небольшие константы относятся к псевдообласти «imm»;
- «lab». Уникальная целочисленная метка используется для абстрагирования смещения относительно начала области кода, которая обычно является статической областью, доступной только для чтения. Метки преобразуются в абсолютный адрес в конечном коде, но мы позволяем ассемблеру выполнить работу по вычислению смещения байтов. В промежуточном коде, как и в коде ассемблера, метки – это просто имена для машинных инструкций;
- «obj». Смещение производится относительно начала некоторого объекта, выделенного из кучи, что означает, что доступ к нему будет осуществляться относительно другого адреса. Например, объектно-ориентированный язык может адресовать переменные экземпляра как смещение относительно указателя *self* или *this*;
- «imm». Псевдообласть для непосредственных значений означает, что смещение является фактическим значением, а не адресом.

Области не представляют особой сложности, если к ним привыкнуть. Теперь рассмотрим, как они применяются в структуре данных, которую компилятор использует для представления адресов в сгенерированном коде.

Представление типов данных для промежуточного кода

Наиболее распространенной формой промежуточного кода, используемой в компиляторах, является **трехадресный код**. Каждая инструкция содержит опкод и от нуля до трех операндов, которые являются значениями, используемыми данной инструкцией, обычно это адреса. Для компилятора Jzero мы определим класс под названием `address`, который представляет адрес как область и смещение. Unicon-реализация класса `address` начинается в файле *address.icn*, как показано здесь:

```
class address(region, offset)
end
```

Соответствующая версия Java требует, чтобы мы решили, какие типы использовать для области и смещения. Мы используем строки для представления областей, а смещение обычно представляет собой расстояние в байтах от начала области, поэтому оно может быть представлено целым числом. Java-реализация класса `address` в файле *address.java* выглядит следующим образом:

```
public class address {
      public String region;
      public int offset;
      address(String s, int x) { region = s; offset = x; }
}
```

В дальнейшем мы будем добавлять методы к этому классу по мере необходимости. Учитывая это представление адресов, мы можем определить наш трехадресный код в классе с именем `tac`, который состоит из опкода и до трех адресов. Не все опкоды будут использовать все три адреса. Unicon-реализация класса `tac` в файле *tac.icn* показана здесь:

```
class tac(op, op1, op2, op3)
end
```

Один интересный вопрос, который приходит на ум в этот момент, заключается в том, использовать ли встроенный в Unicon тип списка и класс ArrayList в Java или реализовать явное представление связанного списка. Явное представление связанного списка обеспечит более тесную синхронизацию кода Unicon и Java и облегчит совместное использование подсписков. К тому же, если честно, мне немного стыдно при мысли об использовании в Java *ArrayList get()* и *set(), length* по сравнению с *length()*, *size()* и так далее.

С другой стороны, если мы будем создавать собственные связанные списки, мы будем тратить место и время на относительно низкоуровневый код для основных операций, которые должен обеспечить язык реализации. Итак, мы будем использовать встроенный тип списка в Unicon и *ArrayList* в Java и посмотрим, насколько хорошо они работают. Соответствующая Java-реализация в файле *tac.java* выглядит следующим образом:

```
public class tac {
      String op;
      address op1, op2, op3;
              tac(String s) { op = s; }
              tac(String s, address x) { op = s; op1 = x; }
              tac(String s, address o1, address o2) {
                      op = s; op1 = o1; op2 = o2; }
              tac(String s, address o1, address o2, address o3) {
                      op = s; op1 = o1; op2 = o2; op3 = o3; }
}
```

Чтобы было удобно собирать списки трехадресных инструкций, мы добавим метод с именем `gen()` в дерево *class*, который создает одну трехадресную инструкцию и возвращает новый список размером один, содержащий ее. Unicon-реализация в файле tree.icn показана здесь:

```
method gen(o, o1, o2, o3)
        return [ tac(o, o1, o2, o3) ]
end
```

В версии Unicon не нужно ничего делать, чтобы разрешить пропущенные аргументы и инициализировать `op1...op3` нулевым значением. Соответствующая реализация Java в *tree.java* использует синтаксис метода с переменным аргументом. Это выглядит следующим образом:

```
ArrayList<tac> gen(String o, address a) {
      ArrayList<tac> L = new ArrayList<tac>();
      tac t = null;
      switch(a.length) {
              case 3: t = new tac(o, a[0], a[1], a[2]); break;
              case 2: t = new tac(o, a[0], a[1]); break;
              case 1: t = new tac(o, a[0]); break;
              case 0: t = new tac(o); break;
              default: j0.semerr("gen(): wrong # of arguments");
      }
      L.add(t);
      return L;
}
```

Приведенные выше примеры демонстрируют два способа, которыми Java неловко поддерживает методы с переменным числом аргументов. Во-первых, существует перегрузка методов – класс `tac` имеет четыре различных конструктора, позволяющих использовать разное количество аргументов. С другой стороны, метод `gen()` использует синтаксис переменных аргументов Java, который предоставляет странный массив, не являющийся массивом, для хранения аргументов метода.

Инструкции трехадресных кодов легко разбиваются на короткие последовательности из 1–2 собственных инструкций, и компьютеры со сложными наборами инструкций имеют инструкции, которые имеют три операнда и прямое соответствие трехадресному коду. Теперь рассмотрим, как дополнить узлы дерева, чтобы включить информацию, необходимую для промежуточного кода, включая эти трехадресные инструкции.

Добавление атрибутов промежуточного кода в дерево

В предыдущих двух главах мы добавили в узлы дерева синтаксиса информацию об области видимости таблицы символов и типах. Теперь пришло время добавить представления для нескольких частей информации, необходимых для генерации кода.

Для каждого узла дерева, содержащего промежуточный код, поле с именем *icode* будет обозначать список инструкций кода, которые соответствуют выполнению кода для этого поддерева.

Для выражений второй атрибут с именем addr будет обозначать адрес, по которому можно найти вычисленное значение выражения после выполнения этого выражения.

Для каждого узла дерева, который содержит промежуточный код, поля *first* и *follow* будут обозначать метки, которые будут использоваться в качестве це-

лей, к которым поток управления должен перейти в начале этого кода, или выполнить любую инструкцию, логически следующую сразу после этого кода.

Наконец, для каждого узла дерева, который представляет собой булево выражение, поля *onTrue* и *onFalse* будут содержать метки, которые будут использоваться в качестве целей, когда это булево выражение будет признано истинным или ложным соответственно. Эти имена были выбраны, чтобы избежать зарезервированных в Java слов *true* и *false*.

В Unicon добавление этих атрибутов к дереву классов в файле *tree.icn* дает нам следующее:

```
class tree (id,sym,rule,nkids,tok,kids,isConst,stab,
         typ,icode,addr,first,follow,onTrue,onFalse)
```

Узлы нашего дерева становятся все толще и толще. Хотя нам, возможно, придется использовать тысячи их для компиляции программы, на машине с гигабайтами оперативной памяти расход памяти не будет иметь значения. Соответствующие Java-дополнения к файлу *tree.java* выглядят следующим образом:

```
class tree {
             .   .   .
     typeinfo typ;
     ArrayList<tac> icode;
     address addr, first, follow, onTrue, onFalse;
```

На данном этапе вам, возможно, интересно, как мы планируем вычислять эти новые атрибуты. Ответ в основном заключается в том, что они синтезируются с помощью обратного обхода дерева, который мы рассмотрим в следующих разделах. Но здесь есть несколько нюансов.

Генерация меток и временных переменных

Несколько вспомогательных методов окажутся полезными во время генерации промежуточного кода. Если хотите, можете считать их фабричными методами. Фабричный метод – это метод, который конструирует объект и возвращает его. В любом случае, нам нужен один метод для меток, чтобы облегчить поток управления, и один для временных переменных. Назовем их *genlabel()* и *genlocal()*.

Генератор меток, `genlabel()`, генерирует уникальную метку. Уникальное целое число можно получить из `serial.getid()`, поэтому `genlabel()` может, например, конкатенировать символ «L» с результатом вызова этого метода. Интересным является вопрос о том, должен ли *genlabel()* возвращать метку в виде целого числа или строки, адреса, трехадресной инструкции или списка, содержащего трехадресную инструкцию. Правильным ответом, вероятно, будет адрес. Код Unicon для `genlabel()` в файле *tree.icn* может выглядеть следующим образом:

```
method genlabel()
        return address("lab", serial.getid())
end
```

Соответствующий метод Java в файле *tree.java* выглядит так:

```
address genlabel() {
        return new address("lab", serial.getid());
}
```

Генератор временных переменных, *genlocal()*, должен зарезервировать участок памяти в локальной области. Логически это подразумевает выделение памяти на вершине стека в некотором будущем адресном пространстве, когда сгенерированная программа будет однажды запущена. Это пьянящая задача. На практике выделение стека происходит в виде большого участка памяти при каждом вызове метода. Компилятор вычисляет, насколько большим должен быть этот участок для каждого метода, включая все локальные переменные в программе, а также временные переменные, которые используются для вычисления частичных результатов при выполнении различных операторов, когда выполняются выражения в методе.

Каждая локальная переменная требует определенного количества байтов, но для этой книги выделенные единицы – это полные 64-битные слова с двойным выравниванием. Смещение указывается в байтах, но если вам нужен байт, вы округляете в большую сторону и выделяете слово. Таблица символов – это место, где Jzero выделяет локальные переменные. В коде класса дерева методы могут вызывать *genlocal()* из таблицы символов с помощью выражения *stab.genlocal()*. Для реализации *genlocal()* записи таблицы символов расширяются, чтобы отслеживать адрес, который занимает переменная, а сама таблица символов отслеживает, сколько байтов было выделено в общей сложности. Каждый раз, когда поступает запрос на новую переменную, мы выделяем требуемое для нее количество слов и увеличиваем счетчик на это количество.

Как указано, *genlocal()* выделяет одно слово и выдает для него адрес. Для языка, который выделяет многословные объекты в стеке, *genlocal()* может быть расширен, чтобы принять параметр, определяющий количество слов для выделения, но поскольку Jzero выделяет массивы и экземпляры классов из кучи, для него *genlocal()* может обойтись выделением одного восьмибайтового слова при каждом вызове.

Записи таблицы символов расширяются полем адреса с именем addr. Unicon-дополнение к файлу *symtab_entry.icn* показано здесь:

```
class symtab_entry(sym,parent_st,st,isConst,typ,addr)
```

Java-дополнение к файлу *symtab_entry.java* выглядит следующим образом:

```
public class symtab_entry {
     .   .   .
     address addr;
     .   .   .
     symtab_entry(String s, symtab p, boolean iC,
                    symtab t, typeinfo ti, address a) {
             sym = s; parent_st = p; isConst = iC;
             st = t; typ = ti; addr = a;
```

Класс таблицы символов получает счетчик байтов, показывающий, сколько байтов было выделено в пределах области, соответствующей таблице символов. Вставка в таблицу символов помещает адрес в запись таблицы символов и увеличивает счетчик. Вызов `genlocal()` вставляет новую переменную. Здесь показана Unicon-реализация в файле *symtab.icn*:

```
class symtab(scope, parent, t, count)
      .    .    .
      method insert(s, isConst, sub, typ)
              . . .
              t[s] := symtab_entry(s, self, sub, isConst, typ,
                                   address(scope,count))
              count +:= 8
              .    .    .
      end
      method genlocal()
      local s := "local$" || count
              insert(s, false, , typeinfo("int"))
              return t[s].address
      end
initially
      t := table()
      count := 0
end
```

Представленное выше изменение метода `insert()` передает адрес верхушки области конструктору `symtab_entry` всякий раз, когда выделяется переменная, а затем увеличивает счетчик, чтобы выделить для нее место. Добавление метода `genlocal()` состоит во вставке новой переменной и возвращении ее адреса. Временная переменная содержит символ доллара, `$`, поэтому ее имя не может появиться как имя обычной переменной в исходном коде. Java-реализация этого дополнения к файлу *symtab.java* состоит из следующих изменений:

```
public class symtab {
      .   .   .
      int count;
      .   .   .
      void insert(String s, Boolean iC, symtab sub,
                  typeinfo typ){
              .   .   .
                      t.put(s, new symtab_entry(s, this, iC, sub, typ,
                                   new address(scope,count)));
                count += 8;
                      }
              }
              address genlocal() {
                      String s = "local$" + count;
                      insert(s, false, null, new typeinfo("int"));
                      return t.get(s).addr;
              }
```

Рассмотрим набор инструкций промежуточного кода с помощью вспомогательных методов для генерации меток и временных переменных.

Набор инструкций промежуточного кода

Промежуточный код подобен машинно независимому коду ассемблера для абстрактного процессора. Набор инструкций определяет набор опкодов. Каждый опкод специфицирует свою семантику, включая сколько операндов он использует и какие изменения состояния происходят при его выполнении. Поскольку это промежуточный код, нам не нужно беспокоиться о регистрах или режимах адресации – мы можем просто определить изменения состояния в терминах того, какие модификации должны произойти в основной памяти. Набор инструкций промежуточного кода включает в себя как обычные инструкции, так и псевдоинструкции, как и в других языках ассемблера. Давайте рассмотрим набор опкодов для языка Jzero. Существует две категории опкодов – инструкции и декларации.

Инструкции

За исключением непосредственного режима, операндами инструкций являются адреса, а инструкции, которые неявно разыменовывают значения в памяти, расположены по этим адресам. На типичных современных машинах единицами слов являются 64 бита. Смещение указывается в байтах. Опкоды, эквиваленты на языке C и описания инструкций приведены в табл. 9.1.

Таблица 9.1. Различные опкоды, эквиваленты на языке C и их описания

Опкод	Эквивалент на языке C	Описание
ADD,SUB,MUL,DIV	x=y op z	Сохранить результат бинарной операции над y и z в X
NEG	x = –y	Сохранить результат унарной операции над y в x
ASN	x = y	Сохранить y в x
ADDR	x = &y	Сохранить адрес y в x
LCON	x = *y	Сохранить содержимое, указанное y, в x
SCON	*x = y	Сохранить y в месте, указанном x
GOTO	goto L	Безусловный переход к L
BLT,BLE,BGT,BGE	if(x rop y)goto L	Проверка отношения и условный переход к L
BIF	if (x) goto L	Условный переход к L, если x ≠ 0
BNIF	if (!x) goto L	Условный переход к L, если x = 0
PARM		Сохранить x как параметр (поместить в стек вызовов)

Окончание табл. 9.1

Опкод	Эквивалент на языке C	Описание
CALL	x=p(...)	Вызвать процедуру p с n словами параметров
RET	return x	Возврат из функции с результатом x

Далее мы рассмотрим некоторые декларации.

Декларации

Декларации и другие псевдоинструкции обычно связывают имя с некоторым объемом памяти в одной из областей памяти программы. Некоторые декларации и их описания приведены в табл. 9.2.

Таблица 9.2. Декларации и их описания

Декларация	Описание
glob x,n	Объявляет глобальную переменную с именем x, которая имеет смещение n в глобальной области
proc x,n1,n2	Объявляет процедуру x с n1 словами параметров и n2 словами локальных переменных
loc x,n	Объявляет локальную переменную с именем x, которая имеет смещение n в локальной области
lab Ln	Объявляет метку Ln, которая будет именем инструкции в области кода
end	Объявляет конец текущей процедуры

Эти инструкции и декларации являются общими и могут описывать различные виды вычислений. Ввод/вывод может быть смоделирован путем добавления инструкций или путем выполнения системных вызовов во время выполнения. Далее в данной главе мы будем существенно использовать этот набор инструкций, начиная с раздела «*Генерация кода для выражений*». Но сначала мы должны вычислить еще несколько атрибутов в нашем дереве синтаксиса, которые необходимы для потока управления.

Аннотирование деревьев синтаксиса метками для потока управления

Код в некоторых узлах дерева будет являться источником или целью потока управления. Чтобы сгенерировать код, нам нужен способ генерирования меток для целей и передачи данной информации инструкциям, которые будут направлены к этим целям. Имеет смысл начать с атрибута с именем «первый» (*first*). Первый атрибут содержит метку, на которую могут перейти инструкции ветвления для выполнения данного оператора или выражения. При необходимости его можно синтезировать методом перебора. Если так поступить, то

можно просто присвоить уникальную метку каждому узлу дерева. Результат будет изобиловать избыточными и неиспользуемыми метками, но это будет работать. Для большинства узлов первая метка может быть синтезирована из одной из его дочерних меток, вместо того чтобы выделять новую.

Рассмотрим аддитивное выражение *e1* + *e2*, которое строит нетерминал с именем `AddExpr`. Если бы в *e1* был какой-либо код, он имел бы поле `first`, и это была бы метка, которую следует использовать для первого поля всего `AddExpr`. Если в *e1* нет кода, то в *e2* может быть некоторый код, который предоставит поле `first` для родителя. Если ни одно из подвыражений не имеет кода, то нам нужно сгенерировать новую метку для любого кода, который мы генерируем в узле `AddExpr`, выполняющем сложение. Аналогичная логика применима и к другим операторам. Unicon-реализация метода `genfirst()` в файле *tree.icn* выглядит следующим образом:

```
method genfirst()
      every (!\kids).genfirst()
      case sym of {
      "UnaryExpr": first := \kids[2].first | genlabel()
      "AddExpr"|"MulExpr": first := \kids[1|2].first |
              genlabel()
      .   .   .
      default: first := (!\kids).first
      }
end
```

В случае ветвей в приведенном выше коде опираются на целевую оценку Unicon. Тест «на непустое» (non-null test) применяется к полям `first` для тех дочерних узлов, которые могут иметь код. Если этот тест проваливается, вызывается `genlabel()`, чтобы присвоить `first`, если этот узел будет генерировать инструкцию. По умолчанию, что хорошо для большого количества нетерминалов, расположенных выше в грамматике, – это присвоить `first`, если у дочернего узла есть такое поле, но не вызывать `genlabel()`. Соответствующий Java-код в файле *tree.java* выглядит следующим образом:

```
void genfirst() {
      if (kids != null) for(tree k:kids) k.genfirst();
      switch (sym) {
              case "AddExpr": case "MulExpr": {
                      if (kids[1].first != null) first = kids[1].first;
                      else if (kids[2].first != null)
                               first = kids[2].first;
                      else first = genlabel();
                      }
              .   .   .
              }
}
```

В дополнение к атрибуту `first` нам нужен атрибут с именем `follow`, который обозначает метку для перехода к коду, который идет сразу после данного блока. Это поможет реализовать такие операторы, как *if-then*, а также операторы

break. Атрибут *follow* передает информацию от предков, братьев и сестер, а не от детей. Эта реализация должна использовать наследуемый атрибут, а не синтезированный. Вместо простого восходящего обратного обхода информация копируется вниз в прямом обходе, как это было показано ранее для копирования информации о типе в списки объявления переменных. Атрибут `follow` использует значения атрибута `first` и должен вычисляться после того, как был запущен `genfirst()`.

Рассмотрим самое простое правило грамматики, где вы можете определить атрибут `follow`. В грамматике Jzero основное правило для операторов, выполняющихся последовательно, состоит в следующем:

```
BlockStmts : BlockStmts BlockStmt ;
```

Для наследуемого атрибута родитель (`BlockStmts`, который находится слева от двоеточия) отвечает за предоставление атрибута `follow` для двух дочерних элементов. `follow` левого дочернего элемента будет первой инструкцией в правом дочернем элементе, поэтому атрибут перемещается от одного брата или сестры к другому. `follow` правого дочернего элемента будет тем, что следует за родительским, поэтому он копируется вниз. После установки этих значений родитель должен заставить детей сделать то же самое для своих детей, если таковые имеются. Unicon-реализация в файле *tree.icn* показана здесь:

```
method genfollow()
      case sym of {
      "BlockStmts": {
              kids[1].follow := kids[2].first
              kids[2].follow := follow
              }
      .   .   .
      }
      every (!\kids).genfollow()
end
```

Соответствующий Java-код в файле *tree.java* выглядит следующим образом:

```
void genfollow() {
      switch (sym) {
              case "BlockStmts": {
                      kids[0].follow = kids[1].first;
                      kids[1].follow = follow;
                      break;
                      }
              .   .   .
      }
      if (kids != null) for(tree k:kids) k.genfollow();
}
```

Вычисление этих атрибутов позволяет генерировать инструкции для потока управления, который направляется к этим различным меткам. Вы могли за-

метить, что многие из этих меток могут никогда не использоваться. Мы можем либо сгенерировать их все в любом случае, либо разработать механизм, позволяющий генерировать их только тогда, когда они являются фактической целью инструкции ветвления. Прежде чем мы перейдем к генерации кода для сложных инструкций потока управления, которые используют эти метки, давайте рассмотрим более простую проблему генерации кода для обычных арифметических и аналогичных выражений.

Генерация кода для выражений

Самый простой код для генерации – это прямой код, состоящий из операторов и выражений, которые выполняются последовательно без потока управления. Как было описано ранее в этой главе, для каждого узла необходимо вычислить два атрибута – атрибут, определяющий, где найти значение выражения, называется *addr*, а промежуточный код, необходимый для вычисления его значения, называется *icode*. Значения, которые необходимо вычислить для этих атрибутов в подмножестве выражений грамматики Jzero, приведены в табл. 9.3. Оператор ||| относится к конкатенации списков.

Таблица 9.3. Семантические правила для выражений

Выражение	Семантические правила

Assignment : IDENT '=' AddExpr	`Assignment.addr = IDENT.addr`
	`Assignment.icode = AddExpr.icode |||`
	`gen(ASN, IDENT.addr, AddExpr.addr)`

AddExpr : AddExpr_1 '+' MulExpr	`AddExpr.addr = newtemp()`
	AddExpr_1`.icode ||| MulExpr.icode |||`
	`gen(ADD, AddExpr.addr,` AddExpr_1`.addr), MulExpr.addr`

AddExpr : AddExpr_1 '–' MulExpr	`AddExpr.addr = newtemp()`
	AddExpr_1`.icode ||| MulExpr.icode |||`
	`gen(SUB, AddExpr.addr,` AddExpr_1`.addr), MulExpr.addr`

MulExpr : MulExpr_1 '*' UnaryExpr	`MulExpr.addr = newtemp()`
	MulExpr_1`.icode ||| UnaryExpr.icode |||`
	`gen(MUL, MulExpr.addr,` MulExpr_1`.addr), UnaryExpr.addr`

MulExpr : MulExpr_1 '/' UnaryExpr	`MulExpr.addr = newtemp()`
	MulExpr_1`.icode ||| UnaryExpr.icode |||`
	`gen(DIV, MulExpr.addr,` MulExpr_1`.addr), UnaryExpr.addr`

UnaryExpr : '–' UnaryExpr_1	`UnaryExpr.addr = newtemp()`
	UnaryExpr_1`.icode |||`
	`gen(NEG, UnaryExpr.addr,` UnaryExpr_1`.addr)`

UnaryExpr : '(' AddExpr')'	`UnaryExpr.addr = AddExpr.addr` `UnaryExpr.icode = AddExpr.icode`
UnaryExpr : IDENT	`UnaryExpr.addr = IDENT.addr` `UnaryExpr.icode = emptylist()`

Основным алгоритмом генерации промежуточного кода является восходящий обратный обход дерева синтаксиса. Чтобы представить его в виде неболь-

ших фрагментов, обход разбивается на основной метод, `gencode()`, и вспомогательные методы для каждого нетерминала. В Unicon метод `gencode()` в файле *tree.icn* выглядит следующим образом:

```
method gencode()
      every (!\kids).gencode()
      case sym of {
              "AddExpr": { genAddExpr() }
              "MulExpr": { genMulExpr() }
              .   .   .
              "token": { gentoken() }
              default: {
                      icode := []
                      every icode |||:= (!\kids).icode
                      }
              }
end
```

По умолчанию для узлов дерева, которые не знают, как генерировать код, используется просто конкатенация кодов дочерних узлов. Соответствующий Java-код выглядит следующим образом:

```
void gencode() {
      if (kids != null) for(tree k:kids) k.gencode();
      switch (sym) {
      case "AddExpr": { genAddExpr(); break; }
      case "MulExpr": { genMulExpr(); break; }
      .   .   .
      case "token": { gentoken(); break; }
      default: {
              icode = new ArrayList<tac>();
              if (kids != null) for(tree k:kids)
                  icode.addAll(k.icode);
              }
      }
}
```

Методы, используемые для генерации кода для определенных нетерминалов, должны иногда генерировать различные инструкции в зависимости от производственного правила. Код Unicon для `genAddExpr()` показан здесь:

```
method genAddExpr()
      addr := genlocal()
      icode := kids[1].icode ||| kids[2].icode |||
              gen(if rule=1320 then "ADD" else "SUB",
                  addr, kids[1].addr, kids[2].addr)
end
```

После создания временной переменной для хранения результата код строится путем добавления соответствующей арифметической инструкции в конец кода дочернего элемента. В этом методе правило 1320 относится к сло-

жению, а правило 1321 – к вычитанию. Соответствующий Java-код выглядит следующим образом:

```
void genAddExpr() {
      addr = genlocal();
      icode = new ArrayList<tac>();
      icode.addAll(kids[0].icode);
      icode.addAll(kids[1].icode);
      icode.addAll(gen(((rule==1320)?"ADD":"SUB"), addr,
                        kids[0].addr, kids[1].addr));
}
```

Метод `gentoken()` генерирует код для терминальных символов. Атрибут `icode` обычно пустой. Для переменной атрибут `addr` представляет собой поиск в таблице символов, а для литеральной константы атрибут `addr` – это ссылка на значение в области констант или непосредственное значение. В Unicon метод `gentoken()` выглядит следующим образом:

```
method gentoken()
icode := []
case tok.cat of {
parser.IDENTIFIER: { addr := stab.lookup(tok.text).addr }
parser.INTLIT: { addr := address(«imm», tok.ival) }
.   .   .
}
end
```

Атрибут `icode` представляет собой пустой список, а атрибут `addr` получается с помощью поиска в таблице символов. В Java `gentoken()` выглядит следующим образом:

```
void gentoken() {
      icode = new ArrayList<tac>();
      switch (tok.cat) {
              case parser.IDENTIFIER: {
                      addr = stab.lookup(tok.text).addr; break; }
              case parser.INTLIT: {
                      addr = new address("imm", tok.ival); break; }
              .   .   .
              }
}
```

Из всего этого вы можете заметить, что генерация промежуточного кода для выражений в прямом коде в основном сводится к конкатенации кодов операндов с последующим добавлением одной или нескольких новых инструкций в каждый оператор. Эта работа облегчается за счет предварительного выделения памяти для адресов временных переменных. Код для потока управления представляет собой более сложную задачу.

Генерация кода для потока управления

Генерация кода для управляющих структур, таких как условия и циклы, является более сложной задачей, чем код для арифметических выражений, как было показано в предыдущем разделе. Вместо использования синтезированных атрибутов за один проход снизу вверх код для потока управления применяет информационную метку, которая должна быть помещена туда, где она необходима, с помощью унаследованных атрибутов. Это может включать несколько проходов по дереву синтаксиса. Мы начнем с выражения условий логики, необходимых даже для самых простых потоков управления, таких как операторы *if*, а затем покажем, как применить это к циклам, после чего рассмотрим соображения, необходимые для вызова методов.

Генерация целевых меток для выражений условий

Мы уже настроились на поток управления, присвоив атрибуты `first` и `follow`, как описано в разделе *«Аннотирование деревьев синтаксиса метками для потока управления»*. Рассмотрим, какую роль играют атрибуты `first` и `follow`, начиная с самого простого выражения потока управления, оператора `if`. Разберем, например, следующий фрагмент кода:

```
if (x < 0) x = 1;
y = x;
```

Дерево синтаксиса для этих двух операторов показано на рис. 9.2.

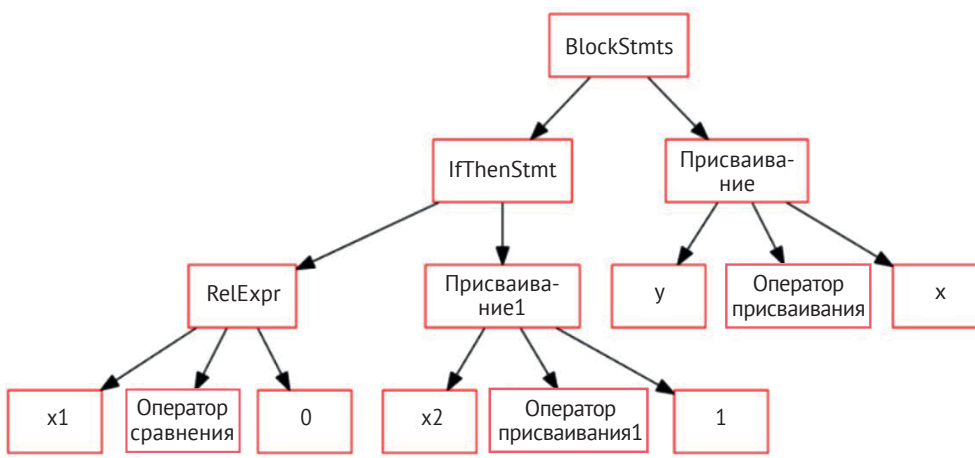


Рис. 9.2. Дерево синтаксиса, иллюстрирующее поток управления

BlockStmts присвоил атрибут *follow* узла *IfThenStmt* атрибуту *first* присваивания *y=x*. Код, сгенерированный для *RelExpr*, должен перейти к метке *first* части `then`, изображенной здесь как *Assignment1*, если *RelExpr* истинно. Он должен перейти к *follow* всего *IfThenStmt*, если *RelExpr* ложно. Чтобы реализовать это, значения меток, вычисленные из *IfThenStmt*, могут быть унаследованы в двух новых атрибутах *RelExpr*. Мы не можем назвать их *true* и *false*, потому что это зарезервированные в Java слова. Назовем атрибут для перехода, когда выраже-

ние истинно, *onTrue*, а атрибут для перехода, когда выражение ложно, – *onFalse*. Семантические правила, которые мы хотим реализовать, показаны в табл. 9.4.

Таблица 9.4. Семантические правила для операторов if-then и if-then-else

Выражение	Семантические правила
IfThenStmt : if'('Expr')'Stmt	Expr.onTrue = Stmt.first Expr.onFalse = IfThenStmt.follow Stmt.follow = IfThenStmt.follow IfThenStmt.icode = (Expr.icode != null) ? Expr.icode : gen(BIF, Expr.onFalse, Expr.addr,con:0) IfThenStmt.icode |||:= gen(LABEL, Expr.onTrue) ||| Stmt.icode
IfThenElseStmt : if'('Expr')'$Stmt_1$ else $Stmt_1$	Expr.onTrue = $Stmt_1$.first Expr.onFalse = $Stmt_2$.first $Stmt_1$.follow = IfThenElseStmt.follow; $Stmt_2$.follow = IfThenElseStmt.follow; IfThenElseStmt.icode = (Expr.icode != null) ? Expr.icode : gen(BIF, Expr.onFalse, Expr.addr,con:0) IfThenElseStmt.icode |||:= gen(LABEL, Expr.onTrue) ||| $Stmt_1$.icode ||| gen(GOTO, IfThenElseStmt.follow) ||| gen(LABEL, Expr.onFalse) ||| $Stmt_2$.icode

Как мы видим, условием в *IfThenStmt* является *Expr*, которое наследует *onTrue* от *Stmt*, который является его частью then, и наследует *onFalse* от родительского атрибута *follow* – любого кода, который следует за всем *IfThenStmt*. Эти атрибуты должны быть унаследованы в булевы подвыражения с помощью таких операторов, как логические AND и OR. Семантические правила для булевых операторов показаны в табл. 9.5.

Таблица 9.5. Семантические правила для булевых выражений

Выражение	Семантические правила
AndExpr : $AndExpr_1$ && EqExpr	EqExpr.first = newlabel(); $AndExpr_1$.onTrue = EqExpr.first; $AndExpr_1$.onFalse = AndExpr.onFalse; EqExpr.onTrue = AndExpr.onTrue; EqExpr.onFalse = AndExpr.onFalse; AndExpr.icode = $AndExpr_1$.icode ||| gen(LABEL, EqExpr.first) ||| EqExpr.icode
OrExpr : $OrExpr_1$ || AndExpr	AndExpr.first = newlabel(); $OrExpr_1$.onTrue = OrExpr.onTrue; $OrExpr_1$.onFalse = AndExpr.first; AndExpr.onTrue = OrdExpr.onTrue; AndExpr.onFalse = OrExpr.onFalse; OrExpr.icode = $OrExpr_1$.icode ||| gen(LABEL, AndExpr.first) ||| AndExpr.icode
UnaryExpr : ! $UnaryExpr_1$	$UnaryExpr_1$.onTrue = UnaryExpr.onFalse $UnaryExpr_1$.onFalse = UnaryExpr.onTrue UnaryExpr.icode = $UnaryExpr_1$.icode

Код для вычисления атрибутов `onTrue` и `onFalse` находится в методе под названием `gentargets()`. Его Unicon-реализация в файле *tree.icn* выглядит следующим образом:

```
method gentargets()
        case sym of {
        «IfThenStmt»: {
                kids[1].onTrue := kids[2].first
                kids[1].onFalse := follow
                }
        «CondAndExpr»: {
                kids[1].onTrue := kids[2].first
                kids[1].onFalse := onFalse
                kids[2].onTrue := onTrue
                kids[2].onFalse := onFalse
                }
        .   .   .
        }
        every (!\kids).gentargets()
end
```

Соответствующий метод Java выглядит так:

```
void gentargets() {
     switch (sym) {
     case "IfThenStmt": {
             kids[0].onTrue = kids[1].first;
             kids[0].onFalse = follow;
             }
     case "CondAndExpr": {
             kids[0].onTrue = kids[1].first;
             kids[0].onFalse = onFalse;
             kids[1].onTrue = onTrue;
             kids[1].onFalse = onFalse;
             }
     .   .   .
     }
     if (kids!=null) for(tree k:kids) k.gentargets();
}
```

Рассмотрев, как назначаются атрибуты `onTrue` и `onFalse`, можно сказать, что последней частью головоломки является код, который генерируется для операторов сравнения, таких как тест $x < y$. Для этих операторов можно было бы сгенерировать код, который вычисляет истинный (1) или ложный (0) результат и сохраняет его во временной переменной, например в арифметическом операторе. Однако смысл вычисления меток `onTrue` и `onFalse` заключался в том, чтобы сгенерировать код, который будет переходить непосредственно к нужной метке в зависимости от того, истинным или ложным был тест. Это необходимо для реализации семантики замыкания булевых операторов, которую Jzero унаследовал от Java, а до этого – от C. Вот Unicon-реализация метода `genRelExpr()`, который вызывается из `gencode()` для генерации промежуточного кода для реляционных выражений:

```
method genRelExpr()
      op := case kids[2].tok.cat of {
              ord("<"): "BLT"; ord(">"): "BGT";
              parser.LESSTHANOREQUAL: "BLE"
              parser.GREATERTHANOREQUAL: "BGT" }
      icode := kids[1].icode ||| kids[3].icode |||
                 gen(op, onTrue, kids[1].addr, kids[3].addr) |||
                 gen("GOTO", onFalse)
end
```

Этот код начинается с установки переменной `op` в трехадресный опкод, который соответствует целочисленной категории оператора, извлеченного из `kids[2].tok.cat`. Затем он создает код путем конкатенации левого и правого операндов, за которым следует условная ветвь, если оператор оценивается как истина, и безусловная ветвь, если оператор ложен. Соответствующая Java-реализация выглядит следующим образом:

```
void genRelExpr() {
      String op = "ERROR";
      switch (kids[1].tok.cat) {
              case '<': op="BLT"; break; case ';': op="BGT"; break;
              case parser.LESSTHANOREQUAL: op="BLE"; break;
              case parser.GREATERTHANOREQUAL: op="BGT";
              }
      icode = new ArrayList<tac>();
      icode.addAll(kids[0].icode); icode.addAll(kids[2].icode);
      icode.addAll(gen(op, onTrue, kids[0].addr,
                       kids[2].addr));
      icode.addAll(gen("GOTO", onFalse));
}
```

По сравнению с кодом, который генерируется для обычной арифметики, код для управляющих структур, таких как операторы *if*, передает много информации о метках. Теперь рассмотрим, что необходимо добавить в код для поддержки управляющих структур цикла.

Генерация кода для циклов

В этом разделе представлены идеи по генерации промежуточного кода для циклов *while* и *for*. Код цикла *while* должен быть практически идентичен оператору *if-then*, с единственным дополнением в виде метки в начале и *goto* в конце для перехода к этой метке. Цикл *for* – это просто цикл *while* с парой дополнительных выражений. В табл. 9.6 показаны семантические правила для этих двух управляющих структур.

Таблица 9.6. Семантические правила для генерации промежуточного кода циклов

Выражение	Семантические правила
WhileStmt : while '('Expr')' Stmt	Expr.onTrue = newlabel(); Expr.first = newlabel(); Expr.false = WhileStmt.follow; Stmt.follow = Expr.first; WhileStmt.icode = gen(LABEL, Expr.first) ||| Expr.icode ||| gen(LABEL, Expr.true) ||| Stmt.icode ||| gen(GOTO, Expr.first) |||

ForStmt : for(ForInit; Expr; ForUpdate
Stmt
 	
он же
 	
ForInit;
while(Expr) {
Stmt
ForUpdate
}
	Expr.true = newlabel();
	Expr.first = newlabel();
	Expr.false = S.follow;
	Stmt.follow = ForUpdate.first;
	S.icode = ForInit.icode |||
	gen(LABEL, Expr.first) |||
	Expr.icode ||| gen(LABEL, Expr.true) |||
	Stmt.icode |||
	ForUpdate.icode |||
	gen(GOTO, Expr.first) |||

Метод `genWhileStmt()` является представителем таких методов генерации кода потока управления, как `genIfStmt()` и `genForStmt()`. Большая часть работы выполняется во время вычисления атрибутов `first`, `follow`, `onTrue` и `onFalse`. Unicon-реализация `genWhileStmt()` выглядит следующим образом:

```
method genWhileStmt()
icode := gen(«LAB», kids[1].first) ||| kids[1].icode |||
       gen("LAB", kids[1].onTrue) |||
       kids[2].icode ||| gen("GOTO", kids[1].first)
end
```

Java-реализация `genWhileStmt()` показана здесь:

```
void genWhileStmt() {
     icode = new ArrayList<tac>();
     icode.addAll(gen("LAB", kids[0].first));
     icode.addAll(kids[0].icode);
     icode.addAll(gen("LAB", kids[0].onTrue));
     icode.addAll(kids[1].icode);
     icode.addAll(gen("GOTO", kids[0].first));
}
```

Осталось представить еще один аспект потока управления. Вызовы методов (или функций) являются основными строительными блоками во всех формах императивного кода и объектно-ориентированного кода.

Генерация промежуточного кода для вызовов методов

Набор инструкций промежуточного кода содержит три опкода, связанных с вызовами методов: *PARM*, *CALL* и *RET*. Чтобы вызвать метод, сгенерированный

код выполняет несколько инструкций *PARM*, по одной для каждого параметра, за которыми следует инструкция *CALL*. Вызванный метод выполняется до тех пор, пока не достигнет инструкции *RET*, после чего он возвращается к вызывающей стороне. Этот промежуточный код представляет собой абстракцию нескольких различных способов, которыми аппаратное обеспечение поддерживает абстракции методов (или функций).

В некоторых процессорах параметры передаются в основном в регистрах, в то время как в других они все передаются в стеке. На уровне промежуточного кода мы должны проверить, выполняются ли инструкции PARM в порядке появления фактических параметров в исходном коде или в обратном порядке. В объектно-ориентированных языках, таких как Jzero, мы также беспокоимся о доступности ссылки на объект внутри вызываемого метода. Языки программирования отвечают на эти вопросы по-разному в разных процессорах, но для наших целей мы будем использовать следующие соглашения о вызове: параметры указываются в обратном порядке, за ними следует экземпляр объекта (self или указатель this) в качестве неявного дополнительного параметра, за которым следует инструкция *CALL*.

Когда `gencode()` достигает *MethodCall*, который является типом первичного выражения в нашей грамматике, он вызовет `genMethodCall()`. Его реализация в Unicon показана здесь:

```
method genMethodCall()
      local nparms := 0
      if k := \ kids[2] then {
              icode := k.icode
              while k.sym === "ArgList" do {
                      icode |||:= gen("PARM", k.kids[2].addr)
                      k := k.kids[1]; nparms +:= 1 }
              icode |||:= gen("PARM", k.addr); nparms +:= 1
              }
      else icode := [ ]
      if kids[1].sym === "QualifiedName" then
              icode |||:= gen("PARM", kids[1].kids[1].addr)
      else icode |||:= gen("PARM", "self")
      icode |||:= gen("CALL", kids[1].addr, nparms)
end
```

Сгенерированный код начинается с кода для вычисления значений параметров. Затем он выдает инструкции `PARM` в обратном порядке, что следует из того, как бесконтекстная грамматика построила дерево синтаксиса для списков аргументов. Самые сложные части этого метода связаны с тем, как промежуточный код узнает адрес, который следует использовать для текущего объекта. Java-реализация `genMethodCall()` показана здесь:

```
void genMethodCall() {
      int nparms = 0;
      icode = new ArrayList<tac>();
      if (kids[1] != null) {
              icode.addAll(kids[1].icode);
```

```
            tree k = kids[1];
            while (k.sym.equals("ArgList")) {
                    icode.addAll(gen("PARM", k.kids[1].addr));
                    k = k.kids[0]; nparms++; }
            icode.addAll(gen("PARM", k.addr)); nparms++;
            }
      if (kids[0].sym.equals("QualifiedName"))
            icode.addAll(gen("PARM", kids[0].kids[0].addr));
      else icode.addAll(gen("PARM", "self"));
      icode.addAll(gen("CALL", kids[0].addr,
                      new address("imm",nparms)));
}
```

То, что было показано в этом разделе, вероятно, убедило вас в том, что генерация кода для вызывающей стороны является более сложной задачей, чем генерация кода для инструкции возврата, которую вы можете изучить в коде этой главы на GitHub. Стоит также отметить, что в код тела каждого метода может быть добавлена инструкция *ret*, чтобы гарантировать, что код никогда не будет выполняться после завершения тела метода в результате перехода к тому, что идет после него.

Проверка сгенерированного промежуточного кода

Вы не можете запустить промежуточный код, но должны тщательно его проверить. Убедитесь, что логика выглядит корректно на тестовых примерах для каждой функции, о которой вы заботитесь. Чтобы проверить сгенерированный код для такого файла, как *hello.java*, выполните следующую команду, используя либо Unicon (левая часть), либо реализацию Java (правая часть). Напоминаю, что для Java в Windows вы должны сначала выполнить что-то вроде *set CLASSPATH=".;C:\byopl"* или его эквивалент в **Панели управления** либо **Настройках**. В Linux это может выглядеть как *export CLASSPATH=.;..*:

```
j0 hello.java                java ch9.j0 hello.java
```

Результат должен выглядеть примерно так, как показано ниже:

```
.string
L0:
        string       "hello, jzero!"
.global
     global  global:8,hello
     global  global:0,System
.code
proc  main,0,0
     ASIZE loc:24,loc:8
     ASN loc:16,loc:24
     ADD loc:32,loc:16,imm:2
     ASN loc:16,loc:32
L138:
     BGT L139,loc:16,imm:3
     GOTO L140
```

```
L139:
      PARM strings:0
      PARM loc:40
      CALL PrintStream__println,imm:1
      SUB loc:48,loc:16,imm:1
      ASN loc:16,loc:48
      GOTO L138
L140:
        RET
end
no errors
```

Просматривая промежуточный код, вы начинаете понимать, что, возможно, сможете закончить этот компилятор и перевести ваш исходный код в машинный код какого-то типа. Если вы не рады, то должны быть рады. На этом этапе можно заметить множество ошибок, например пропущенные функции или операторы ветвления, которые переходят к несуществующим меткам. Так что проверьте все, прежде чем бросаться генерировать окончательный код.

Заключение

В этой главе вы узнали, как генерировать промежуточный код. Генерация промежуточного кода – это первый важный шаг в синтезе инструкций, которые в конечном итоге позволят машине выполнить программу пользователя. Навыки, которые вы приобрели в этой главе, основываются на тех навыках, которые используются в семантическом анализе, например как добавлять семантические атрибуты к узлам дерева синтаксиса и как, при необходимости, обходить узлы дерева синтаксиса сложными способами.

Одной из важных особенностей этой главы был пример набора инструкций промежуточного кода, который мы использовали для языка Jzero. Поскольку код является абстрактным, вы можете добавлять к этому набору инструкций новые инструкции по мере необходимости для вашего языка. Создание списков этих инструкций было простым с использованием типа списка Unicon *data* и довольно простым с помощью типа *ArrayList* в Java.

В этой главе вы узнали, как генерировать код для прямолинейных выражений, таких как арифметические вычисления. Гораздо больше усилий здесь было приложено к инструкциям для потока управления, которые часто включают инструкции *goto*, целевые инструкции которых должны иметь метки. Это повлекло за собой вычисление нескольких атрибутов для меток, включая наследуемые атрибуты, перед построением списков инструкций кода.

Теперь, когда вы сгенерировали промежуточный код, вы готовы перейти к генерации окончательного кода. Однако сначала в главе 10 «*Раскраска синтаксиса в IDE*» вы получите практическое развлечение, состоящее в изучении того, как использовать свои знания для включения раскрашивания синтаксиса в среде IDE.

Глава 10

Раскраска синтаксиса в IDE

Для создания полезного языка программирования требуется не только компилятор или интерпретатор, который позволяет запускать программы. Для этого необходима экосистема инструментов для разработчиков. Эта экосистема часто включает в себя отладчики, интерактивную справку или **интегрированную среду разработки**, обычно называемую **IDE**. IDE в широком смысле можно определить как любую среду программирования, в которой редактирование исходного кода, компиляция, компоновка (если таковые имеются) и выполнение могут быть реализованы в рамках одного **пользовательского интерфейса (UI)**.

В этой главе рассматриваются некоторые проблемы, связанные с включением кода из вашего языка программирования в IDE, чтобы обеспечить раскраску синтаксиса и визуальную обратную связь о синтаксических ошибках. Одна из причин реализации этого заключается в том, что многие программисты не будут воспринимать ваш язык всерьез, если у него нет IDE. Код в данной главе будет примером Unicon, поскольку не существует IDE, которая была бы реализована одинаково в Unicon и Java.

В этой главе рассматриваются следующие основные темы:

- загрузка примеров IDE, используемых здесь;
- интеграция компилятора в редактор программиста;
- избежание повторного разбора всего файла при каждом изменении;
- использование лексической информации для выделения токенов цветом;
- выделение ошибок с помощью результатов разбора;
- добавление поддержки Java.

Навыки, которые необходимо освоить в этой главе, связаны со связью и координацией программных систем. В первую очередь благодаря объединению IDE и компилятора в один исполняемый файл высокопроизводительная связь осуществляется путем передачи ссылок на общие данные, вместо того чтобы прибегать к файловому **вводу/выводу (I/O)** или **межпроцессному взаимодействию (IPC)**.

Примечание

Написание IDE – это большой проект, который мог бы стать темой целой книги. В отличие от других глав этой книги, где мы представляем код компилятора с нуля, в этой главе описывается, как синтаксическая раскраска была добавлена в Unicon IDE. Unicon IDE была написана Клинтоном Джеффри (Clinton Jeffery) и Ноланом Клейтоном (Nolan Clayton) с последующим вкладом многих других людей. Луис Альвидрес (Luis Alvidres) выполнил работу по раскраске синтаксиса в рамках своего магистерского проекта. Отчет о проекте Луиса можно найти на сайте *http://www.unicon.org/reports/alvidres.pdf*.

Глава завершается описанием того, как код Unicon IDE был впоследствии включен в приложение виртуальной среды под названием **Collaborative Virtual Environment (CVE)**. В CVE код IDE был обобщен для поддержки других языков, включая Java и C++. Хани Бани-Саламех (Hani Bani-Salameh) выполнил эту работу в Unicon в рамках своего докторского исследования. Описание добавления поддержки Java в код Unicon IDE сравнимо с тем, что мы могли бы сделать для добавления поддержки нового языка, такого как Jzero, в существующую IDE. В следующем разделе описывается, как получить исходный код программ, обсуждаемых в этой главе.

Загрузка примеров IDE, используемых в этой главе

В этой главе мы рассмотрим две простые IDE, которые иллюстрируют представленные концепции. Первая IDE – это программа под названием **ui**, что означает Unicon IDE. Программа *ui* включена в дистрибутив языка Unicon, где ее можно найти в директории *uni/ide*. Программа состоит примерно из 10 000 строк кода в 26 файлах, не считая кода в библиотечных модулях. Программа *ui* показана на следующем скриншоте (рис. 10.1).

Вторая IDE называется CVE. Помимо прочего, CVE – это часть исследовательского программного обеспечения, которая экспериментально расширяет IDE *ui* для поддержки C++ и Java. Вы можете скачать исходный код CVE с сайта *cve.sf.net*. CVE показана на рис. 10.2. Если сравнить этот скриншот с предыдущим, то можно увидеть, что IDE программы CVE началась с кодовой базы *ui*.

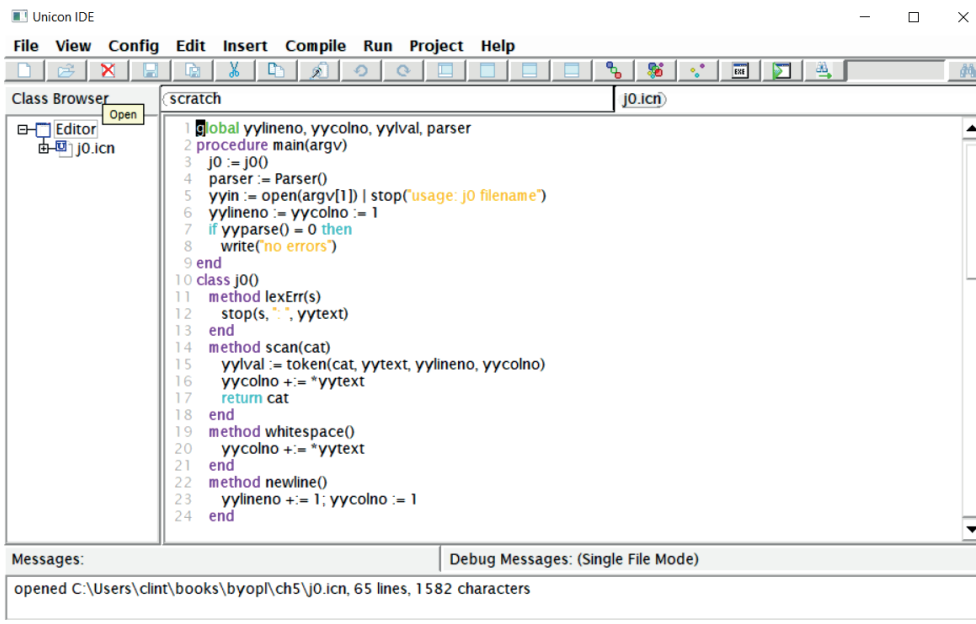


Рис. 10.1. IDE ui

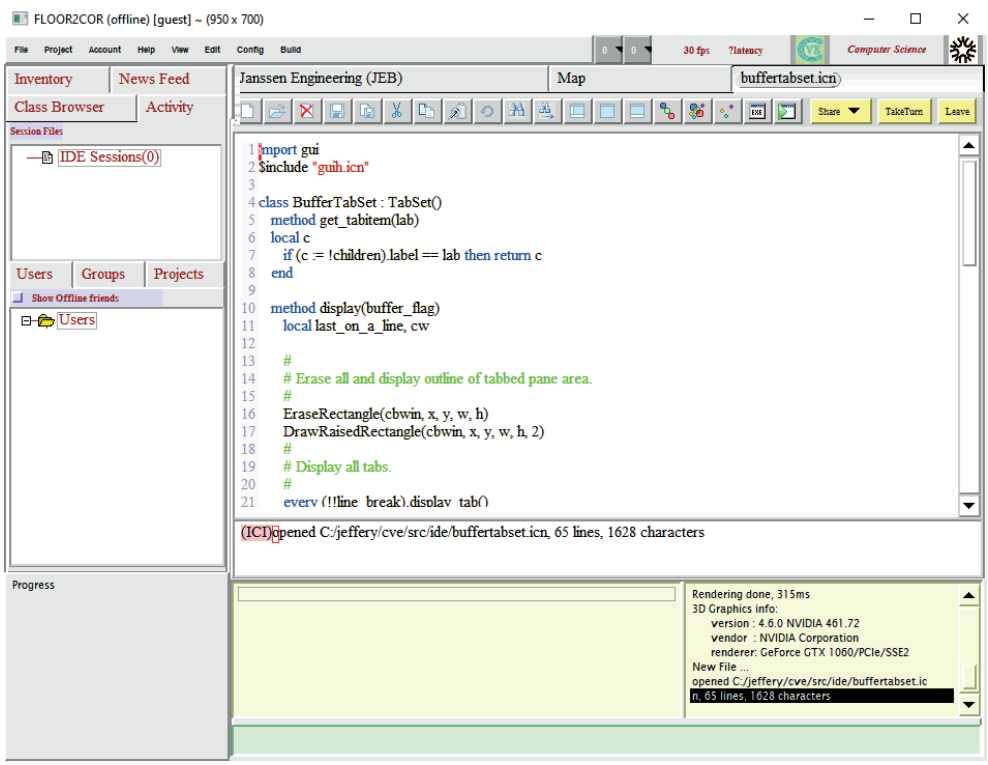


Рис. 10.2. IDE CVE

Исходный код CVE хранится в системе контроля версий **Subversion**, доступной по адресу *subversion.apache.org*. После установки Subversion выполните следующую команду для получения CVE. Команда `svn checkout` создаст подкаталог с именем `cve/` в том каталоге, в котором вы находитесь при выполнении этой команды:

```
svn checkout https://svn.code.sf.net/p/cve/code/trunk/cve
```

Теперь перейдем к краткому описанию IDE Unicon и того, как компилятор Unicon был интегрирован в IDE для раскраски синтаксиса.

Интеграция компилятора в редактор программиста

Первая половина компилятора Unicon, в общих чертах соответствующая главам от второй («*Дизайн языка программирования*») до пятой («*Деревья синтаксиса*») этой книги, была интегрирована в IDE Unicon, известную как *ui*. Фронтенд языка Unicon состоит из трех основных компонентов: **препроцессора**, **лексического анализатора** и **парсера**. В трансляторе Unicon эти компоненты вызываются из процедуры *main()*. Транслятор открывает, считывает и записывает файлы в файловой системе для выполнения операций ввода-вывода, а также обеспечивает обратную связь с пользователем, записывая текст в стандартный вывод или стандартную ошибку в окне консоли либо терминала. В IDE компоненты компилятора вызываются из-за кулис, пока пользователь редактирует свой код в **графическом пользовательском интерфейсе (GUI)**. Исходный код получается непосредственно из памяти в IDE, а выходные данные компилятора получаются из памяти IDE и представляются пользователю. В общей сложности были модифицированы семь файлов из транслятора Unicon, чтобы стать библиотечными модулями, которые можно подключать и использовать из других программ, кроме Unicon. В следующем разделе рассматривается, как исходный код в IDE поступает в модули компилятора. После этого мы рассмотрим, как выходные данные компилятора, включая сообщения об ошибках, поступают в IDE.

Анализ исходного кода из среды IDE

Компилятор обычно получает входные данные путем открытия и чтения из именованного файла. lex-совместимый интерфейс, используемый многими компиляторами, специально указывает, что входные данные поступают от дескриптора открытого файла, хранящегося в глобальной переменной с именем *yyin*. Это слишком медленно для IDE, которая выполняет лексический и синтаксический анализы часто и многократно по мере редактирования пользователем. Вместо чтения из файла фронтенд компилятора Unicon был модифицирован таким образом, чтобы он мог читать исходный код, который уже находился в основной памяти.

Рассмотрим файл с именем *hello.icn*, который содержит трехстрочную программу Hello, World. В IDE исходный код хранится в виде списка из трех строк. Список строк хранится в переменной с именем *contents* в редактируемом виджете текстового списка. Запись этого списка строк на диск и каждый раз вызов

компилятора для его чтения слишком медленны. Модификация компилятора для явной зависимости от списка строк в IDE чуть усложняет компилятор и делает интерфейс между двумя инструментами немного хрупким. С другой стороны, чтение из списка строк – это не совсем ракетостроение. Формат списка строк также позволяет легко выбрать часть файла, а не весь файл целиком, для передачи в парсер.

Лексический анализатор Unicon находится в файле *uni/unicon/unilex.icn* в дистрибутиве Unicon. До интеграции код лексического анализатора Unicon использовал предварительное считывание всего исходного файла в виде большой строки в переменную с именем *buffer*. Поддержка чтения из списка строк означает размещение одной строки за раз в буфере, и всякий раз, когда лексический анализатор достигает конца строки, выполняется следующий код:

```
if type(yyin) == «list» then {
      if buffer := pop(yyin) then {
              yylineno +:= 1
              yycolno := 1
              if tokflags < Newline then tokflags +:= Newline
              return yylex(ender)
              }
      }
```

Этот код использует `pop()` для удаления следующей строки из списка строк вместо вызова `read()` для чтения следующей строки из файла. Поскольку `pop()` модифицирует свой исходный список, лексическому анализу подвергается копия списка строк, предоставленная IDE, а не исходный список строк IDE. Копирование списка (или части списка) строк не требует выделения и копирования всех строковых данных, содержащих код, копируется только структура списка. Теперь рассмотрим, как сообщения компилятора доставляются в GUI IDE.

Отправка выходных данных компилятора в IDE

Вместо того чтобы напрямую писать сообщения об ошибках, семь библиотечных модулей компилятора были модифицированы для построения списка диагностики ошибок. Затем обычный компилятор мог выводить их на консоль, а IDE могла выводить сообщения в подокно или изображать их графически. Рассмотрим возможное сообщение об ошибке, например следующее:

```
hello.icn:5: '}' expected
```

До интеграции компилятор мог написать это с помощью следующей строки кода:

```
write(&errout, filename, «:», lineno, «: «, message)
```

Чтобы интегрировать такие сообщения в IDE, код компилятора был изменен следующим образом:

```
iwrite( filename, «:», lineno, «: «, message)
```

Процедура `iwrite()` фактически сохраняет диагностику в списке с именем `parsingErrors`, который может быть прочитан IDE или записан в *&errout*, в зависимости от того, подключен ли фронтенд компилятора к IDE или компилятору Unicon.

В IDE Unicon эти ошибки парсинга отображаются в текстовом виде в методе ReparseCode(). Вызывается парсер, а затем, если были обнаружены ошибки, выполняются следующие строки:

```
every errorObject := !parsingErrors do {
      errorObject.lineNumber +:= lineNumberOffset
      if errorObject.lineNumber <= *contents then {
              SetErrorLineNumber(errorObject.lineNumber)
              uidlog.MsgBox.set_contents(
                      [errorObject.lineNumber ||": " ||
                              errorObject.errorMessage])
              }
      }
```

Текст сообщения об ошибке помещается в компонент GUI с именем `MsgBox` с помощью вызова его метода `set_contents()`. `MsgBox` отображается под исходным кодом. В дополнение к отображению текста, который вывел бы компилятор в случае ошибки, IDE подсвечивает строку, в которой произошла ошибка. Это обсуждается позже в разделе «*Подсветка ошибок с использованием результатов разбора*».

В этом разделе, посвященном интеграции компилятора в IDE или редакторе программиста, обсуждались основные моменты того, как объединить две большие, сложные, уже существующие части программного обеспечения. Компилятор Unicon и IDE поддерживаются в основном независимо друг от друга. Сохранение связей между ними снижает вероятность того, что изменения в одном из них повлияют на другой. Если вы пишете новую IDE с нуля, чтобы использовать ее вместе с новым компилятором, более широкая интеграция может обеспечить дополнительные возможности или лучшую производительность, но за счет сложности, ремонтопригодности и переносимости. Теперь давайте рассмотрим, как применить проверку синтаксиса без постоянного парсинга файла, пока пользователь редактирует код.

Предотвращение повторного разбора всего файла при каждом изменении

Лексический и синтаксический анализ, необходимый для разбора входных данных, обнаружения и сообщения о синтаксических ошибках, представленные в этой книге с главы 2 («*Дизайн языка программирования*») по главу 8 («*Проверка типов в массивах, вызовах методов и доступах к структурам*»), являются значительными алгоритмами. Хотя инструменты Flex и Yacc, которые мы использовали, являются высокопроизводительными, при большом входном файле сканирование и парсинг становятся настолько медленными, что пользователи не захотят повторять разбор всего файла при каждом его изме-

нении в текстовом редакторе IDE. В ходе тестирования мы обнаружили, что повторный разбор всего файла становится проблемой для файлов размером более 1000 строк.

Сложные алгоритмы инкрементального разбора, которые минимизируют объем, подлежащий повторному разбору после изменений, являются предметом докторских диссертаций и научных статей. Для Unicon IDE используется простой подход. Всякий раз, когда курсор перемещается от измененной строки, выбирается единица парсинга, которая начинается с измененной строки и простирается выше и ниже до границ ближайшей процедуры, метода или другой единицы глобального объявления. Эта единица разбирается заново.

В Unicon это дает очень хорошую производительность. Луис Альвидрес (Luis Alvidres) обнаружил, что когда весь блок объявления повторно обрабатывается после изменения строки, в 98 % случаев компилятор повторно обрабатывает менее 100 строк кода. Большинство остальных 2 % случаев, а именно процедуры или методы размером более 100 строк, по-прежнему не представляют проблемы. Только самые большие процедуры или тела методов приводят к медленному повторному разбору. Это часто машинный код, например выходной код Flex или Yacc, который пользователь редко редактирует вручную. Для таких случаев IDE отключает проверку синтаксиса, чтобы избежать неприемлемого времени отклика пользователя.

Код для выбора фрагмента для повторного разбора, когда курсор перемещается за пределы строки, находится в методе `GetCode()`, который можно найти в классе *BuffEditableTextList*, являющемся подклассом стандартного компонента редактора GUI Unicon под названием *EditableTextList*. *BuffEditableTextList* находится в файле *uni/ide/buffertextlist.icn*. Метод `GetCode()` реализован следующим образом. Сначала идет заголовок метода и набор объявлений локальных переменных:

```
method GetCode()
     local codeSubStringList,
             originalPositionY, currentPositionY, token,
             startPositionY := 0, endPositionY := 0,
             inClass := 0, inMethod := 0
```

В методе `GetCode()` эти переменные играют следующие роли:

- `codeSubStringList` – это список, содержащий номер строки, с которой следует начать отчет об ошибке, за которым следуют строки для анализа кода, который может быть затронут изменениями в текущей строке;
- `originalPositionY` – это строка, в которой был изменен текст;
- `currentPositionY` – переменная, используемая для перехода вверх и вниз от текущей строки;
- `token` – это целочисленная категория, возвращаемая функцией *yylex()*, как показано в главе 2 «*Дизайн языка программирования*»;
- `startPositionY` и `endPositionY` – это строки, которые определяют начало и конец текущего объявления;
- `inClass` и `inMethod` сообщают, находится объявление в классе или методе.

Инициализация в методе GetCode() заключается в сбросе парсера и запуске переменных положения с текущей строки курсора, которая указывает, на какой строке находится курсор. Это показано в следующем фрагменте кода:

```
reinitialize()
originalPositionY := currentPositionY := cursor_y
```

Первичный цикл в этой процедуре идет назад от местоположения курсора, используя функцию лексического анализатора компилятора yylex для просмотра первого токена в каждой строке и поиска ближайшей предыдущей строки, на которой начинается вложенное объявление, как показано в следующем фрагменте кода:

```
while currentPositionY > 0 do {
      yyin := contents[currentPositionY]
      yylex_reinit()
      if (token := yylex()) ~=== EOFX then {
              if token = (PROCEDURE | METHOD | CLASS) then {
                      if token=METHOD then inMethod := 1
                      if token=CLASS then inClass := 1
                      startPositionY := currentPositionY
                      }
              }
      if startPositionY ~= 0 then break
      currentPositionY -:= 1
      }
```

Вы можете видеть, что движение назад достигается путем декрементирования индекса текущей строки, хранящегося в переменной currentPositionY. Предшествующий цикл while завершается, когда найдена строка, начинающаяся с зарезервированного слова PROCEDURE, METHOD или CLASS. Когда цикл while завершается, не найдя вложенного объявления, синтаксический разбор начинается со строки 1. Это достигается с помощью следующего оператора if:

```
if startPositionY = 0 then startPositionY := 1
```

Затем метод выполняет поиск вперед от курсора, чтобы найти конечный токен. Лексические особенности, такие как многострочное продолжение содержимого строки, делают эту задачу более сложной, чем можно было бы ожидать. Следующий цикл while достаточно длинный, поэтому он разделен на несколько сегментов для пояснения. Первый сегмент показывает, что цикл while движется по одной строке за раз в коде для отображения, продвигая currentPositionY в каждой строке и получая содержимое из списка строк переменной – члена класса с именем contents. В Unicon незавершенные строковые константы могут занимать несколько строк, заканчивающихся символом подчеркивания, который обрабатывается внутренним циклом while:

```
currentPositionY := cursor_y
while currentPositionY < *contents + 1 do {
      yyin := contents[ currentPositionY ]
      yylex_reinit()
      while countdoublequotes(yyin)%2=1 & yyin[-1]=="_" do {
              currentPositionY +:= 1
              if not (yyin ||:= contents[currentPositionY]) then {
                      break break
                      }
              }
      yylex_reinit()
```

Основная задача цикла `while`, приведенного в приведенном выше фрагменте кода, представлена во второй половине цикла, показанной далее. Этот внутренний цикл использует лексический анализатор компилятора для идентификации токенов, которые указывают на границу компилируемого блока. Токен `END` указывает на конец компилируемого блока, в то время как `CLASS` и `PROCEDURE` указывают на начало последующего блока:

```
while ( token := yylex() ) ~=== EOFX do {
      case token of {
      END: {
              endPositionY := currentPositionY
              break
              }
      CLASS | PROCEDURE: {
              if currentPositionY ~= startPositionY then {
                      endPositionY := currentPositionY-1
                      break
                      }
              }
      default : break
      }
}
```

Метод завершается построением фрагмента исходного кода для разбора и возвращением его в виде списка строк с предшествующим указанием номера строки, непосредственно предшествующей фрагменту, как показано в следующем фрагменте кода:

```
if endPositionY = 0 then
        return codeSubStringList := [ 0 ] ||| contents
if startPositionY = 0 then startPositionY := 1
if inMethod = 1 then
      codeSubStringList := [ startPositionY,
              "class __Parse()" ] |||
              contents[ startPositionY : endPositionY+1 ]|||
              ["end"]
else if inClass = 1 then
      codeSubStringList := [ startPositionY ] |||
              contents[ startPositionY : endPositionY+1 ]|||
```

```
            ["end"]
else
     codeSubStringList := [ startPositionY ] |||
            contents[ startPositionY : endPositionY+1 ]
return codeSubStringList
```

Внимательный читатель может забеспокоиться о том, может ли функция `GetCode()` в представленном виде иногда пропустить границу объявления и захватить слишком много кода, например если слово `PROCEDURE` или `END` не находится в начале строки. Это верно, но не фатально, поскольку просто означает: если исходный код написан в очень странной манере, средство проверки синтаксиса может повторно обработать больший объем кода, чем необходимо. Теперь давайте рассмотрим, как может быть окрашен исходный код.

Использование лексической информации для раскрашивания токенов

Программистам нужна любая помощь в чтении, понимании и отладке своих программ. На рис. 10.1 исходный код представлен в разных цветах, чтобы улучшить его читабельность. Эта раскраска основана на лексических категориях различных элементов текста. Хотя некоторые люди считают цветной текст просто баловством, а другие вообще не способны видеть цвета, большинство программистов ценят это. Многие опечатки и ошибки редактирования текста обнаруживаются быстрее, когда данный фрагмент исходного кода имеет не тот цвет, которого ожидал программист. По этой причине почти все современные редакторы и IDE включают эту функцию.

Расширение компонента EditableTextList для поддержки цвета

EditableTextList – это компонент GUI Unicon, который отображает видимую часть списка строк с использованием одного шрифта и выбором цвета. *EditableTextList* не позволяет устанавливать шрифт или цвета переднего и заднего плана для отдельных букв либо слов. Для поддержки раскраски синтаксиса в Unicon IDE расширяется подкласс *EditableTextList* с именем *BuffEditableTextList* для представления пользователю исходного кода. *BuffEditableTextList* не является полнотекстовым виджетом. Как и *EditableTextList*, он представляет исходный код в виде списка строк, но *BuffEditableTextList* умеет применять раскраску синтаксиса (и выделять строку ошибки, если таковая имеется) на лету, при создании исходного кода.

Раскрашивание отдельных токенов по мере их создания

Для раскрашивания каждого токена *BuffEditableTextList* вызывает `yylex()`, чтобы получить лексическую категорию для каждого токена в момент его создания. Следующий код, взятый из метода *left_string_unicon()* в классе *BuffEditableTextList*, устанавливает цвет, используя пять настраиваемых пользователем цветов, указанных в настройках. Большинство зарезервированных

слов окрашиваются специальным цветом, обозначенным в настройках, как `syntax_text_color`. Отдельные цвета используются для глобальных объявлений, для границ процедур и методов, а также для строковых литералов и литералов языка C++. Этот простой набор цветовых обозначений можно расширить, назначив разные цвета для нескольких других важных лексических категорий, таких как комментарии или директивы препроцессора:

```
while (token := yylex()) ~=== EOFX do {
      Fg(win, case token of {
              ABSTRACT | BREAK | BY | CASE | CREATE | DEFAULT |
              DO | ELSE | EVERY | FAIL | IF | INITIALLY |
              iconINITIAL | INVOCABLE | NEXT | NOT | OF |RECORD|
              REPEAT | RETURN | SUSPEND | THEN | TO | UNTIL |
                      WHILE : prefs.syntax_text_color
              GLOBAL | LINK | STATIC |
                      IMPORT | PACKAGE | LOCAL :
                              prefs.glob_text_color
              PROCEDURE | CLASS |
                      METHOD | END : prefs.procedure_text_color
              STRINGLIT | CSETLIT : prefs.quote_text_color
              default : prefs.default_text_color
              })
      new_s_Position := yytoken["column"] + *yytoken["s"]-1
      DrawString(win, x, y,
                 s[ last_s_Position : (new_s_Position + 1)])
      off := TextWidth(win,
                       s[ last_s_Position : (new_s_Position + 1)])
      last_s_Position := new_s_Position + 1
      x +:= off
      }
```

Как видно из приведенного выше кода, после установки цвета переднего плана из токена сам токен отображается вызовом `DrawString()`, а смещение пикселя, с которого должен быть нарисован последующий текст, обновляется с помощью вызова `TextWidth()`. Все это вместе взятое позволяет различным лексическим категориям исходного кода быть нарисованными в IDE различными цветами. В отрасли используется термин «раскраска синтаксиса», хотя часть компилятора, которую мы ввели, была только лексическим анализатором, а не функцией парсера, выполняющей синтаксический анализ. Теперь давайте рассмотрим, как привлечь внимание пользователя к строке, если парсер определит, что правки, которые были в ней сделаны, порождают код с синтаксической ошибкой.

Подсветка ошибок с использованием результатов разбора

В компоненте *BuffEditableTextList* метод `fire()` вызывается всякий раз, когда контент изменяется, а также при перемещении курсора. При изменении контента устанавливается флаг `doReparse`, указывающий, что должен быть проверен синтаксис кода. Проверка не происходит до тех пор, пока не будет перемещен курсор. Код метода `fire()` показан здесь:

```
method fire(type, param)
      self$Connectable.fire(type, param)
      if type === CONTENT_CHANGED_EVENT then
              doReparse := 1
      if type === CURSOR_MOVED_EVENT &
                      old_cursor_y ~= cursor_y then
              ReparseCode()
end
```

В представленном выше коде метод `ReparseCode()` периодически вызывается в Unicon IDE в ответ на перемещение курсора, чтобы проверить, не привело ли редактирование к синтаксической ошибке. Только перемещения курсора, изменяющие текущую строку (`old_cursor_y ~= cursor_y`), запускают метод `ReparseCode()`, как показано здесь:

```
method ReparseCode ()
      local s, rv, x, errorObject, timeElapsed,
              lineNumberOffset
      if doReparse === 1 then {
              timeElapsed := &time
              SetErrorLineNumber ( 0 )
              uni_predefs := predefs()
              x := 1
              s := copy(GetCode()) | []
              lineNumberOffset := pop(s)
              preproc_err_count := 0
              yyin := ""
              every yyin ||:= preprocessor(s, uni_predefs) do
                      yyin ||:= "\n"
              if preproc_err_count = 0 then {
                      yylex_reinit()
                      /yydebug := 0
                      parsingErrors := []
                      rv := yyparse()
                      }
              if errors + (\yynerrs|0) + preproc_err_count > 0 then {
                      .   .   .every loop from Sending compiler output to
                      the IDE here
                      }
              else uidlog.MsgBox.set_contents(["(no errors)"])
      doReparse := 0
      }
end
```

Метод `ReparseCode()` ничего не делает, если только код не изменился, на что указывает то, что `doReparse` имеет значение 1. Если код изменился, `ReparseCode()` вызывает `GetCode()`, повторно инициализирует лексический анализатор и парсер, вызывает `yyparse()` и отправляет любую ошибку в окно сообщений IDE. Строка, на которой произошла ошибка, также выделяется при перерисовке кода следующим образом. В методе `draw_line()` класса *BuffEditableTextList*, если

текущая строка является строкой, найденной в переменной `errorLineNumber`, цвет переднего плана устанавливается красным:

```
if \errorLineNumber then {
     if i = errorLineNumber then {
             Fg(self.cbwin, "red")
             }
 }
```

Теперь вы убедились, что установка разных цветов для разных типов токенов, таких как зарезервированные слова, довольно проста и требует участия только лексического анализатора, в то время как проверка синтаксических ошибок в фоновом режиме являлась довольно сложной задачей. Теперь посмотрим, что нужно сделать, чтобы обобщить это для добавления в IDE поддержки нового языка.

Добавление поддержки Java

Unicon IDE поддерживает только Unicon. Совместная виртуальная среда CVE расширяет Unicon IDE, добавляя поддержку Java и C/C++. В этом разделе обсуждаются вопросы, связанные с добавлением нового языка (в нашем случае Java, заменяющего Jzero). В идеальном случае это включало бы замену различных фрагментов жестко запрограммированного кода, специфичного для Unicon, структурой данных, которая обрабатывает части, специфичные для языка. CVE не идеальна, но воплощает в себе часть этого идеала.

CVE больше и сложнее, чем Unicon IDE. Код для IDE находится в подкаталоге CVE *src/ide*, но его GUI интегрирован в более крупное клиентское приложение, код которого находится в *src/client*.

В CVE была добавлена переменная `projecttype`, которая указывает на язык, на котором написана текущая программа пользователя. В некоторых местах многоязыковая поддержка IDE обрабатывает специфические для конкретного языка детали с помощью операторов `if`, как в следующем примере:

```
if projecttype == «Java» then …
else if projecttype == «CPP» then …
else if projecttype == «Unicon» then …
else
```

Код такого рода находится в основном в файле *src/client/menubar.icn*. Он используется для выбора объекта, применяемого для вызова процесса построения или запуска программы. В случае Java объект с именем *javaProject* имеет такие методы, как *RunJava()*. Ручное добавление операторов `if` во многих местах IDE не очень удобно. Насколько это возможно, IDE кодирует языковые различия в структурах данных и использует переменную `projecttype` в качестве указателя для выбора корректных данных из таких структур.

IDE использует **объектно-ориентированный (OO)** подход и инкапсулирует язык, который пользователь применяет в паре объектов. Класс *Language* содержит такие детали, как цвет синтаксиса различных токенов, в то время как

класс *Project* предоставляет специфичный для языка диалог для установки таких параметров, как используемый компилятор и опции, передаваемые при компиляции. В нашем случае файл *src/ide/jproject.icn* содержит большую часть кода, специфичного для Java. В дополнение к диалогу для установки опций Java он содержит методы *CompileJava()*, *RunJava()* и *saveJProject()* со специфическим для Java поведением IDE.

Многоязычная раскраска синтаксиса в CVE обрабатывается путем расширения лексического анализатора Unicon в *src/ide/unilex.icn*, чтобы знать зарезервированные слова для Java (и C/C++). Это обрабатывается в процедуре *reswords()* и содержит простые добавления в таблицу зарезервированных слов. Вместо раскрашивания токенов в подклассе *EditableTextList*, как описано ранее в разделе «*Использование лексической информации для раскрашивания токенов*», в CVE раскраска токенов вынесена в неудачно названный класс *LanguageAbstract* в файле *src/ide/langabstract.icn*. Внутри этого класса метод `token_highlighter()` проверяет расширение текущего файла, чтобы решить, следует ли применять зарезервированные слова и правила раскраски Java, C/C++ или Unicon. Код для этих методов выглядит следующим образом:

```
method token_highlighter(f_name,win,s,
                         last_s_Position,x,y,off)
      if find (".java",f_name) then {
              JTok_highlighting(win,s,last_s_Position,x,y,off)
              language := "Java"
              }
      else if find (".cpp"|".c"|".h",f_name) then {
              CTok_highlighting(win,s,last_s_Position,x,y,off)
              language := "C/C++"
              }
      else if find (".icn",f_name) then {
              UTok_highlighting(win,s,last_s_Position,x,y,off)
              language := "Unicon"
              }
      else language := &null
end
```

Это довольно наивный код для перебора. Самое приятное, что если в данный момент в IDE открыто несколько файлов на разных языках, этот код не запутается, он выбирает метод для вызова каждый раз на лету, основываясь на переданном параметре. Тем не менее он выполняет много лишних проверок, когда метод вызывается многократно для каждого токена, который необходимо отрисовать для представления текущего файла. Метод `JTok_highlighting()`, на который мы здесь ссылались, не показан, так как он является очень похожим Java-эквивалентом кода, представленного ранее в разделе «*Раскрашивание отдельных токенов по мере их создания*».

Поддержка CVE для Java не такая полная, как поддержка Unicon IDE для Unicon. CVE не включает в себя полные фронтенды компилятора для Java (и для C/C++) и поэтому не делает разбор кода «на лету», чтобы сообщать об ошибках синтаксиса, пока пользователь редактирует код. CVE IDE сообщает об ошибках

синтаксиса для Java и C/C++, когда пользователь нажимает кнопки **Compile** или **Run** и вызывается (внешний) компилятор.

В этом разделе описаны подходы, с помощью которых IDE может поддерживать несколько языков, такие как отдельная обработка для Java, C/C++ и Unicon. Программисты могут извлечь пользу из этой особенности, если они могут легко переключаться с одного языка программирования на другой без необходимости изучать новую IDE. Если у вас когда-нибудь появится возможность инвестировать время или деньги в разработку IDE, поддержка нескольких языков может помочь получить максимальную отдачу от таких инвестиций.

Заключение

В этой главе вы узнали, как использовать лексическую и синтаксическую информацию для обеспечения окрашивания текста в IDE. Большая часть окрашивания основана на относительно простом лексическом анализе, и большая часть работы была связана с модификацией фронтенда компилятора для обеспечения интерфейса на основе памяти, вместо того чтобы полагаться на чтение и запись файлов на диск. Здесь вы приобрели несколько навыков. Вы узнали, как выделять цветом зарезервированные слова и другие лексические категории в редакторе программиста, передавать информацию между кодом компилятора и редактором программиста, а также выделять синтаксические ошибки при редактировании.

До сих пор эта книга была посвящена анализу и использованию информации, извлеченной из исходного кода. Остальная часть этой книги посвящена генерации кода и среде выполнения, в которой выполняются программы. Тема, которую мы будем изучать в следующей главе, – интерпретаторы байт-кода.

Часть III

Генерация кода и среды выполнения

После этой части вы, наконец, сможете запускать программы, написанные на новом для вас языке программирования.

Данная часть включает в себя следующие главы:

- глава 11 «Интерпретаторы байт-кода»;
- глава 12 «Генерация байт-кода»;
- глава 13 «Генерация собственного кода»;
- глава 14 «Реализация операторов и встроенных функций»;
- глава 15 «Структуры управления доменом»;
- глава 16 «Сборка мусора»;
- глава 17 «Заключительные размышления».

Глава 11

Интерпретаторы байт-кода

Новый язык программирования может включать новые возможности, которые не поддерживаются непосредственно основными процессорами. Наиболее практичным способом генерации кода для многих языков программирования является генерация байт-кода для абстрактной машины, набор инструкций которой напрямую поддерживает предполагаемую область применения языка. Это важно, потому что делает ваш язык свободным от ограничений, связанных с тем, что умеют делать современные процессоры. Это также позволяет ему генерировать код, более тесно связанный с типами проблем, которые вы хотите решать. Если вы создадите свой собственный набор инструкций байт-кода, то сможете выполнять программы, написав виртуальную машину, которая знает, как интерпретировать этот набор инструкций. В данной главе рассматривается, как разработать набор инструкций и интерпретатор, выполняющий байт-код. Поскольку эта глава тесно связана с *главой 12 «Генерация байт-кода»*, вы можете захотеть прочитать их обе, прежде чем погружаться в код.

В этой главе рассматриваются следующие основные темы:

- понимание того, что такое байт-код;
- сравнение байт-кода с промежуточным кодом;
- создание набора инструкций байт-кода для Jzero;
- реализация интерпретатора байт-кода;
- изучение iconx, интерпретатора байт-кода компании Unicon.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch11*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/327bZWn*.

Интерпретатор байт-кода – это часть программного обеспечения, которая выполняет абстрактный набор машинных инструкций. Мы собираемся узнать об интерпретаторах байт-кода, разобрав простую машину байт-кода для Jzero и сделав беглый взгляд на виртуальную машину Unicon. Но сначала давайте рассмотрим, что мы понимаем под **байт-кодом**.

Понимание, что такое байт-код

Байт-код – это последовательность машинных инструкций, закодированных в двоичном формате и записанных не для выполнения процессором, а для абстрактного (или виртуального) набора машинных инструкций, который воплощает семантику данного языка программирования. Хотя многие наборы инструкций байт-кодов для таких языков, как Java, используют байт в качестве наименьшего размера инструкции, почти все они включают более длинные инструкции. Такие инструкции имеют один или несколько операндов. Поскольку многие виды операндов должны быть выровнены по границе слова (word) с адресом, который является кратным четырем или восьми, более подходящим названием для многих форм байт-кода может быть wordcode. Термин «байт-код» обычно используется для таких абстрактных машин независимо от размера инструкции.

Языки, которые непосредственно ответственны за популяризацию байт-кода, – это Pascal и SmallTalk. Эти языки приняли байт-код по разным причинам, которые остаются важными для языков программирования, определяющихся в терминах своего байт-кода. Java взял эту идею и сделал ее известной во всей компьютерной индустрии.

В Pascal байт-код используется для улучшения переносимости реализации языка в различные аппаратные средства и операционные системы. Гораздо проще перенести интерпретатор байт-кода на новую платформу, чем писать новый генератор кода компилятора для этой платформы. Если большая часть языка написана на этом самом языке, интерпретатор байт-кода может быть единственной частью, которую нужно переносить на новую машину.

SmallTalk популяризировал байт-код по другой причине – чтобы создать уровень абстракции, на котором можно было бы реализовать новые возможности, далекие от аппаратного обеспечения того времени. Интерпретатор байт-кода позволяет разработчику языка создавать новые инструкции по мере необходимости, а также определять семантику среды выполнения, которая присутствует во всех реализациях данного языка.

Чтобы объяснить, что такое байт-код, рассмотрим байт-код, который генерируется из следующего кода Unicon:

```
write(«2 + 2 is «, 2+2)
```

Байт-код разбивает выполнение этого выражения на отдельные машинные инструкции. Человекочитаемое представление байт-кода для этого выражения может выглядеть как следующий байт-код Unicon, называемый **ucode**:

```
      mark L1
      var 0
      str 0
      pnull
      int 1
      int 1
      plus
      invoke 2
      unmark
lab L1
```

Переходя от строки к строке, инструкция `mark` обозначает метку назначения, на которой должно продолжаться выполнение, если какая-либо инструкция не выполнится. В Unicon поток управления в основном определяется сбоем, а не булевыми условиями и явными инструкциями *goto*. Инструкция `var` помещает ссылку на переменную *#0* (`write`) в стек оценки. Аналогично инструкция `str` помещает ссылку на строковую константу *#0* (`2 + 2 is`). Инструкция `pnull` помещается, чтобы обеспечить место в стеке оценки для размещения результата оператора (+). Инструкция `int` помещает ссылку на целочисленную константу, которая имеет значение 2, в местоположение области констант *#1*, это делается дважды для двух операндов сложения. Инструкция `plus` извлекает два верхних элемента стека и складывает их, помещая результат на вершину стека. Инструкция `invoke` выполняет вызов с двумя аргументами. Когда `invoke` завершится, аргументы уже будут выведены, а вершина стека, куда была помещена функция `write()`, будет содержать возвращаемое значение функции.

Из приведенного выше примера видно, что байт-код в некоторой степени напоминает промежуточный код, и это намеренно. Так в чем же разница?

Сравнение байт-кода с промежуточным кодом

В главе 9 «*Генерация промежуточного кода*» мы сгенерировали машинно независимый промежуточный код, используя абстрактные трехадресные инструкции. По своей сложности наборы инструкций байт-кода находятся между трехадресным промежуточным кодом и реальным набором инструкций аппаратного обеспечения. Одна трехадресная инструкция может соответствовать нескольким инструкциям байт-кода. Это относится как к прямой передаче любого экземпляра трехадресной инструкции, так и к тому факту, что может существовать несколько опкодов инструкций байт-кода, которые обрабатывают различные специальные случаи данного трехадресного опкода. Байт-код обычно более сложен, чем промежуточный код, даже если удается избежать встречающихся во многих процессорах сложностей, связанных с режимами адресации операндов. Многие или большинство наборов инструкций байт-кода явно или неявно используют регистры, хотя машины байт-кода обычно намного проще, чем аппаратное обеспечение процессоров, с точки зрения количества регистров и их распределения, которые компилятор должен выполнить для генерации кода.

Байт-код – это, как правило, двоичный формат файла. Двоичные форматы очень трудны для чтения. Говоря о байт-коде в этой главе, мы будем приводить примеры в ассемблероподобном формате, но сам байт-код состоит из одних единиц и нулей.

Сравнение программы *hello world* в промежуточном коде и байт-коде может дать вам некоторое представление об их сходстве и различиях. Мы будем использовать следующую программу *hello.java* в качестве примера. Она просто печатает сообщение, если вы передадите ей аргументы командной строки, но содержит как арифметические инструкции, так и инструкции управления:

```
public class hello {
      public static void main(String argv[]) {
              int x = argv.length;
              x = x + 2;
              if (x > 3) {
                      System.out.println("hello, jzero!");
              }
      }
}
```

Трехадресный код Jzero для этой программы выглядит следующим образом. Его операнды включают несколько видов ссылок на память, начиная от локальных переменных и заканчивая метками областей кода. Процедура `main()` состоит из 11 инструкций и 20 операндов, в среднем почти по два операнда на инструкцию:

```
.string
L0:   string  "\"hello, jzero!\""
.global
      global  global:8,hello
      global  global:0,System
.code
proc  main,0,0
      ASIZE   loc:24,loc:8
      ASN     loc:16,loc:24
      ADD     loc:32,loc:16,imm:2
      ASN     loc:16,loc:32
L75:  BGT     L76,loc:16,imm:3
        GOTO  L77
L76:  PARM    strings:0
      FIELD   loc:40,global:0,class:0
      PARM    loc:40
      CALL    PrintStream__println,1
L77:  RET
end
```

Байт-код Java JVM для этой программы, полученный командой *javap -c*, показан ниже (комментарии удалены). Функция `main()` состоит из 14 инструкций с четырьмя операндами, что соответствует менее чем трети операнда на каждую инструкцию:

```
public class hello {
      public hello();
              Code:
              0: aload_0
              1: invokespecial #1
              4: return
      public static void main(java.lang.String[]);
              Code:
                      0: aload_0
                      1: arraylength
```

```
                    2: istore_1
                    3: iload_1
                    4: iconst_2
                    5: iadd
                    6: istore_1
                    7: iload_1
                    8: iconst_3
                    9: if_icmple 20
                    12: getstatic #2
                    15: ldc #3
                    17: invokevirtual #4
                    20: return
}
```

Инструкции в этом методе `main()` иллюстрируют некоторые характеристики лежащей в их основе виртуальной машины итерпретатора байт-кода Java. Это стековая машина. Семейства инструкций `load` и `store` вводят и перемещают переменную между пронумерованным слотом в области основной памяти и вершиной стека, где вычисляются выражения. Этот набор инструкций типизирован, с мнемоническими префиксами для каждого из встроенных скалярных атомарных типов языка Java (`i` – для целых чисел, `f` – для чисел с плавающей запятой и так далее). В нем есть встроенные инструкции для специальных целей, например для возвращения длины массива. Семь целых чисел от –1 до 5 имеют опкоды, которые вводят эти константы. Такая инструкция, как `iadd`, вводит два значения, складывает их, а затем выводит результат.

В этой главе мы представим более простой набор инструкций байт-кода, но приятно знать, над чем работают самые блестящие умы в отрасли. Итак, давайте рассмотрим более простой набор инструкций байт-кода, который подходит для Jzero.

Построение набора инструкций байт-кода для Jzero

В этом разделе описывается простой формат файла и набор инструкций для кода Jzero, сгенерированного из трехадресного промежуточного кода. Для языка, который вы создаете, можно использовать подмножество набора инструкций байт-кода Java. Байт-код Java – это сложный формат, если бы это было не так, мы бы не стали тратить время на представление чего-то более простого. Представленный здесь набор инструкций немного шире, чем используемый Jzero, чтобы позволить общие расширения.

Определение формата файла байт-кода Jzero

Формат Jzero состоит из заголовка, за которым следует раздел данных, а затем последовательность инструкций. Файлы Jzero интерпретируются как последовательность 8-байтовых слов в формате с прямым порядком байтов (little-endian format). Заголовок состоит из необязательного скрипта самоисполнения, магического слова, номера версии и смещения слова первой инструкции относительно магического слова. Скрипт самоисполнения – это набор команд, написанных на каком-либо языке, зависящем от платформы, который вызывает интерпре-

татор, передавая ему файл Jzero в качестве аргумента командной строки. Если присутствует, скрипт самоисполнения должен быть, при необходимости, заполнен так, чтобы его размер был кратен 8 байтам. Магическое слово – это 8 байт, содержащих строку «*Jzero!!\0*». Номер версии – это еще 8 байт, содержащих версию, например 1.0, заполненных нулями, допустим «*1 . 0\0\0\0\0\0*». Смещение слова первой инструкции в наименьшем случае будет равно 3, это число относится к магическому слову. Смещение слова 3 указывает на пустой раздел констант из 0 слов. После магического слова, слова версии и слова смещения начинается выполнение с команды, смещение которой указано в третьем слове.

После заголовка идет раздел статических данных, в котором, в языке Jzero, есть место для статических переменных, а также констант, включая строки. В более серьезном языке может быть несколько видов разделов статических данных. Например, может быть один подраздел данных только для чтения, один – для данных, которые начинаются неинициализированными и не должны физически занимать место в файле на диске, и третий – для статически инициализированных (ненулевых) данных. Для Jzero мы просто выделим на диске один раздел для всего этого.

После раздела данных остальная часть файла состоит из инструкций. Каждая инструкция в формате Jzero представляет собой одно 64-битное слово, содержащее опкод (8 бит), область операндов (8 бит) и операнд (48 бит). Область операндов и операнд используются не во всех опкодах.

Формат Jzero определяет следующие опкоды, представленные в табл. 11.1.

Таблица 11.1. Набор инструкций языка Jzero

Опкод	Мнемонический код	Описание
1	HALT	Остановить
2	NOOP	Ничего не делать
3	ADD	Сложить два верхних целых числа в стеке, вывести сумму
4	SUB	Вычесть два верхних целых числа в стеке, вывести разность
5	MUL	Умножить два верхних целых числа в стеке, вывести произведение
6	DIV	Разделить два верхних целых числа в стеке, вывести частное
7	MOD	Разделить два верхних целых числа в стеке, вывести остаток
8	NEG	Изменение знака целого числа на вершине стека
9	PUSH	Переместить значение из памяти на вершину стека
10	POP	Удалить значение с вершины стека и поместить его в память
11	CALL	Вызвать функцию с n параметрами в стеке
12	RETURN	Вернуться к вызывающей функции с возвращением значения x
13	GOTO	Установить указатель инструкции в местоположение L

Окончание табл. 11.1

Опкод	Мнемонический код	Описание
14	BIF	Удалить стек; если он ненулевой, установить указатель инструкции в L
15	LT	Удалить два значения, сравнить, вставить 1, если первое меньше, иначе – 0
16	LE	Удалить два значения, сравнить, вставить 1, если первое меньше или равно второму, иначе – 0
17	GT	Удалить два значения, сравнить, вставить 1, если первое больше, иначе – 0
18	GE	Удалить два значения, сравнить, вставить 1, если первое больше или равно второму, иначе – 0
19	EQ	Удалить два значения, сравнить, вставить 1, если они равны, иначе – 0
20	NEQ	Удалить два значения, сравнить, вставить 1, если они не равны, иначе – 0
21	LOCAL	Выделить n слов в стеке
22	LOAD	Косвенная вставка; чтение через указатель
23	STORE	Косвенное удаление; запись через указатель

Сравните это с набором инструкций, определенных для промежуточного кода. Набор инструкций промежуточного кода имеет более высокий уровень и допускает три операнда. Этот набор инструкций является более низкоуровневым, и инструкции имеют ноль или один операнд.

Байт области операндов рассматривается как знаковое 8-битное значение. Для неотрицательных значений формат Jzero определяет следующие области операндов:

- область 0 == нет операнда (R_NONE);
- область 1 == абсолютный (R_ABS). Операнд представляет собой смещение слова относительно магического слова;
- область 2 == немедленный (R_IMM). Операндом является значение;
- область 3 == стек (R_STACK). Операндом является смещение слова относительно текущего указателя стека;
- область 4 == куча (R_HEAP). Операндом является смещение слова относительно текущего указателя кучи.

Исходный код интерпретатора байт-кода должен иметь возможность ссылаться на эти опкоды и области операндов по имени. В Unicon можно было бы использовать набор символов *$define*, но вместо этого применяется набор констант в одноэлементном классе с именем *Op*, чтобы сохранить схожесть кода в Unicon и Java. Файл *Op.icn*, содержащий реализацию Unicon, показан здесь:

```
class Op(HALT, NOOP, ADD, SUB, MUL, DIV, MOD, NEG, PUSH,
      POP,
      CALL, RETURN, GOTO, BIF, LT, LE, GT, GE, EQ, NEQ, LOCAL,
      LOAD, STORE, R_NONE, R_ABS, R_IMM, R_STACK, R_HEAP)
initially
      HALT := 1; NOOP := 2; ADD := 3; SUB := 4; MUL := 5
      DIV := 6; MOD := 7; NEG := 8; PUSH := 9; POP := 10
      CALL := 11; RETURN := 12; GOTO := 13; BIF := 14; LT := 15
      LE := 16; GT := 17; GE := 18; EQ := 19; NEQ := 20
      LOCAL := 21; LOAD := 22; STORE := 23
      R_NONE := 0; R_ABS := 1; R_IMM := 2
      R_STACK := 3; R_HEAP := 4
      Op := self
end
```

Соответствующий класс Java выглядит следующим образом:

```
public class Op {
      public final static short HALT=1, NOOP=2, ADD=3, SUB=4,
              MUL=5, DIV=6, MOD=7, NEG=8, PUSH=9, POP=10, CALL=11,
              RETURN=12, GOTO=13, BIF=14, LT=15, LE=16, GT=17, GE=18,
              EQ=19, NEQ=20, LOCAL=21, LOAD=22, STORE=23;
      public final static short R_NONE=0, R_ABS=1, R_IMM=2,
              R_STACK=3, R_HEAP=4;
}
```

Наличие набора опкодов – это хорошо, но более интересные различия между трехадресным кодом и байт-кодом заключаются в семантике инструкций. Мы обсудим это позже в разделе «*Выполнение инструкций*». Прежде чем мы перейдем к этому, вам необходимо узнать больше о том, как работает стековая машина, а также несколько других деталей реализации.

Понимание основ работы стековой машины

Подобно Unicon и Java, машина байт-кода Jzero использует архитектуру стековой машины. Большинство инструкций неявно считывают или записывают значения в стек или из стека. Например, рассмотрим инструкцию ADD. Чтобы сложить два числа, вы помещаете их в стек и выполняете инструкцию ADD. Сама инструкция ADD не принимает никаких операндов, она удаляет из стека два числа, складывает их и помещает результат в стек. Теперь рассмотрим вызов функции с *n* параметрами, синтаксис которого выглядит следующим образом:

```
arg0 (arg1, …, argN)
```

В стековой машине это может быть реализовано следующей последовательностью инструкций:

```
push reference to function arg0
evaluate (compute and push) arg1
.   .   .
```

```
evaluate (compute and push) argN
call n
```

Вызов функции будет использовать свой операнд (`n`) для нахождения `arg0`, адреса вызываемой функции. Когда вызов функции завершится, все аргументы будут удалены, а возвращаемое значение функции будет находиться на вершине стека, где ранее содержался `arg0`. Теперь рассмотрим некоторые другие аспекты реализации интерпретатора байт-кода.

Реализация интерпретатора байт-кода

Интерпретатор байт-кода выполняет следующий алгоритм, который реализует цикл «*выборка–декодирование–исполнение*» (*fetch-decodeexecute*) в программном обеспечении. Большинство интерпретаторов байт-кода практически постоянно используют по крайней мере два регистра – **указатель инструкции** и **указатель стека**. Машина Jzero также включает **регистр указателя базы** для отслеживания кадров вызовов функций и **регистр указателя кучи**, который хранит ссылку на текущий объект.

Хотя указатель инструкции явно упоминается в следующем псевдокоде цикла fetch-decodeexecute, указатель стека используется почти так же часто, но чаще всего он применяется неявно, как побочный продукт семантики инструкций большинства опкодов:

```
load the bytecode into memory
initialize interpreter state
repeat {
      fetch the next instruction,
      advance the instruction pointer
      decode the instruction
      execute the instruction
}
```

Интерпретаторы байт-кода обычно реализуются на языке программирования низкоуровневых систем, таком как C, а не на высокоуровневом прикладном языке, таком как Java или Unicon. По этой причине примеры реализации, возможно, покажутся несколько иконоборческими для закаленных системных программистов. Все в Java объектно-ориентировано, поэтому интерпретатор байт-кода реализован в классе с именем *bytecode*. Наиболее естественным представлением необработанной последовательности байтов в Unicon является строка, в то время как в Java наиболее естественным представлением является массив байтов.

Чтобы реализовать алгоритм интерпретатора байт-кода, в этом разделе каждая из его частей представлена в отдельном подразделе. Сначала рассмотрим, как загрузить байт-код в память.

Загрузка байт-кода в память

Чтобы загрузить байт-код в память, интерпретатор байт-кода должен получить байт-код посредством ввода/вывода некоторого типа. Обычно это де-

лается путем открытия и чтения из именованного локального файла. Когда используются исполняемые заголовки, запущенная программа открывается и считывает себя, как файл данных. Байт-код Jzero определяется как последовательность 64-битных двоичных целых чисел, но это представление является более естественным в одних языках, чем в других.

В Unicon загрузка файла может выглядеть следующим образом:

```
class j0machine(code, ip, stack, sp, bp, hp, op, opr, opnd)
      method loadbytecode(filename)
              sz := stat(filename).st_size
              f := open(filename) | stop("cannot open program.j0")
              s := reads(f, sz)
              close(f)
              s ? {
                      if tab(find("Jzero!!\01.0\0\0\0\0\0")) then {
                              return code := tab(0)
                              }
                      else stop("file ", filename, " is not a Jzero file")
                      }
      end
end
```

Вызов `reads()` в этом примере считывает весь файл байт-кода в одну непрерывную последовательность байтов. В Unicon это представляется как строка. В Java для этого используется массив байтов с оберткой `ByteBuffer` для обеспечения легкого доступа к словам внутри кода. Метод `loadbytecode()` в файле *j0machine.java* выглядит следующим образом:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.charset.StandardCharsets;
import java.nio.ByteBuffer;
public class j0machine {
      public static byte[] code, stack;
      public static ByteBuffer codebuf, stackbuf;
      .   .   .
      public static boolean loadbytecode(String filename)
              throws IOException {
                      code = Files.readAllBytes(Paths.get(filename));
                      byte[] magstr = "Jzero!!\01.0\0\0\0\0\0".getBytes(
                                      StandardCharsets.US_ASCII);
                      int i = find(magstr, code);
                      if (i>=0) {
                      code = Arrays.copyOfRange(code, i, code.length);
                      codebuf = ByteBuffer.wrap(code);
                      return true;
                      }
                      else return false;
              }
}
```

Вызов `copyOfRange()` копирует байт-код в новый массив, в котором отсутствует необязательный исполняемый заголовок. Это делается для упрощения последующих ссылок на код и статическую область, которые являются смещениями относительно магического слова. Для поиска магической строки внутри массива байтов Java требуется следующий вспомогательный метод:

```
public static int find(byte[]needle, byte[]haystack) {
      for( ; i < haystack.length - needle.length+1; ++i) {
                      boolean found = true;
                      for(int j = 0; j < needle.length; ++j) {
                              if (haystack[i+j] != needle[j]) {
                                      found = false;
                                      break;
                              }
                      }
                      if (found) return i;
              }
      return-1;
}
```

Кроме загрузки байт-кода в память перед началом выполнения, интерпретатор байт-кода должен инициализировать свои регистры.

Инициализация состояния интерпретатора

Состояние интерпретатора байт-кода включает в себя области памяти, указатели инструкции и стека, а также небольшое количество постоянных или статических данных, используемых интерпретатором. Метод `init()` выделяет и инициализирует область кода, вызывая метод `loadbytecode()`, и выделяет область стека. Метод `init()` устанавливает регистр инструкции в 0, указывая, что выполнение начнется с первой инструкции в области кода. Стек инициализируется как пустой.

В Unicon инициализация состоит из следующего кода. Для статических переменных Unicon должен выделить отдельную область статических данных, поскольку строковый тип, который используется для загрузки байт-кода, является неизменяемым. И он, и стек интерпретации байт-кода реализованы как списки целых чисел. Это использует тот факт, что Unicon версии 13 и выше реализует списки целых чисел в непрерывном блоке памяти:

```
class j0machine(code, ip, stack, sdr)
      .   .   .
      method init(filename)
              ip := 0
              if not loadbytecode(filename) then
                      stop("cannot open program.j0")
              ip := 16
              ip := finstr := 8*getOpnd()
              data := Data(code[25:ip+1])
              stack := list()
      end
end
```

Соответствующий код Java выглядит следующим образом. Распределение стека в 100 000 слов является несколько произвольным:

```
public class j0machine {
      public static byte[] code, stack;
      public static ByteBuffer codebuf, stackbuf;
      public static int ip, sp;
      public static boolean[] hasOpnd = new boolean[22];
      .   .   .
      public static void init(String filename)
              throws IOexception {
                      ip = sp = 0;
                      if (! loadbytecode(filename)) {
                              System.err.println("cannot open program.j0");
                              System.exit(1);
                              }
                      stack = new byte[800000];
                      stackbuf = ByteBuffer.wrap(stack);
              }
}
```

Выполнение программы в Jzero начинается с выполнения функции с именем `main()`. Это функция в байт-коде Jzero, а не в Java-реализации интерпретатора байт-кода.

Когда запускается функция Jzero `main()`, она ожидает, что в стеке появится обычная запись активации, где можно получить доступ к параметрам. Самый простой способ обеспечить это – инициализировать указатель инструкции на короткую последовательность инструкций байт-кода, которые вызывают `main()` и завершаются после ее возвращения. Таким образом вы можете инициализировать стек, чтобы он содержал параметры функции `main`, если таковые имеются, и инициализировать указатель инструкции, чтобы он указывал на инструкцию `CALL`, которая вызывает `main`, после которой следует инструкция `HALT`.

В случае с Jzero `main()` не имеет параметров, и последовательность запуска всегда будет следующей:

```
PUSH main
CALL 0
HALT
```

Поскольку последовательность запуска одинакова для каждой программы, можно было бы встроить эту последовательность байт-кода в сам код интерпретатора виртуальной машины, и некоторые машины байт-кода делают это. Сложность заключается в том, что смещение кода (адрес) `main()` будет меняться от программы к программе, если только это не сделано жестко, и компоновщик вынужден всегда размещать `main()` в одно и то же место. В случае с Jzero достаточно и приемлемо, чтобы последовательность запуска всегда начинала область кода со смещения слова, указанного в заголовке. Теперь рассмотрим, как интерпретатор извлекает следующую инструкцию.

Выборка инструкций и продвижение указателя инструкции

Регистр `ip`, называемый **указателем инструкции**, хранит местоположение текущей инструкции. Интерпретаторы байт-кода могут представлять его как переменную, обозначающую указатель на код или целочисленный индекс, рассматривая код как массив. В Jzero это байтовое смещение от магического слова. Выборка инструкции в байт-коде – это операция, которая считывает следующую инструкцию в коде. Сюда входит опкод, который должен быть прочитан, а также любые дополнительные байты или слова, которые являются операндами для некоторых инструкций. В Unicon этот метод `fetch()` расположен в классе `j0machine`. Он выглядит следующим образом:

```
class j0machine(code, ip, stack, op, opnd)
      .   .   .
      method fetch()
              op := ord(code[1+ip])
              opr := ord(code[2+ip])
              if opr ~= 0 then opnd := getOpnd()
              ip +:= 8
      end
end
```

Соответствующая Java-версия метода `fetch()` выглядит так:

```
public class j0machine {
      public static byte[] code, stack;
      public static int ip, sp, op
      public static long opnd;
      .   .   .
      public static void fetch() {
              op = code[ip];
              opr = code[ip+1];
              if (opr != 0) { opnd = getOpnd(); }
              ip += 8;
      }
}
```

Метод `fetch()` зависит от метода `getOpnd()`, который считывает следующее слово из кода. В Unicon метод `getOpnd()` может быть реализован следующим образом:

```
method getOpnd()
        return signed(reverse(code[ip+3+:6]))
end
```

Теперь, когда мы рассмотрели выборку инструкций, давайте посмотрим, как выполняется декодирование инструкций.

Декодирование инструкций

Шаг декодирования – это большая проблема в аппаратном обеспечении процессоров, в интерпретаторе байт-кода он не имеет большого значения, но должен быть быстрым. Вам не нужна длинная цепочка операторов *if-else-if* в главном цикле, которая будет выполняться очень часто. Вы хотите, чтобы декодирование занимало небольшое постоянное время, независимо от количества опкодов в вашем наборе инструкций. Поэтому, как правило, его следует реализовать либо с помощью поиска в таблице, либо с помощью управляющей структуры `switch` или `case`. Unicon-реализацию декодирования инструкций можно увидеть в следующем методе `interp()`, который реализует цикл fetch-decode-execute:

```
class j0machine(code, ip, stack)
      .   .   .
      method interp()
              repeat {
                      fetch()
                      case (op) of {
                              Op.HALT: { stop("Execution complete.") }
                              Op.NOOP: {   .   .   .   }
                              .   .   .
                              default: { stop("Illegal opcode " + op) }
                              }
                      }
      end
end
```

Соответствующий код Java выглядит следующим образом:

```
public class j0machine {
      public static byte[] code, stack;
      public static int ip, sp, op, opnd;
      .   .   .
      public static void interp() {
              for(;;) {
                      fetch();
                      switch (op) {
                              case Op.HALT: { stop("Execution complete.");
                              break; }
                              case Op.NOOP: { break; }
                              .   .   .
                              default: { stop("Illegal opcode " + op); }
                              }
                      }
      }
}
```

Ключевые части цикла интерпретатора, которые еще предстоит показать, – это реализация различных инструкций. Здесь приведена пара примеров, кото-

рые зависят от метода `stop()`, для реализации выполнения инструкции `HALT`. В Unicon `stop()` является встроенным методом, а в Java он может быть реализован вот так:

```
public static void stop(String s) {
      System.err.println(s);
      System.exit(1);
      }
```

В следующем разделе описывается остальная часть выполнения цикла fetch-decode-execute.

Выполнение инструкций

Для каждой из инструкций Jzero их выполнение заключается в заполнении тела соответствующего случая (case). В Unicon инструкция сложения может выглядеть как следующая ветвь случая:

```
Op.ADD: {
      val1 := pop(stack); val2 := pop(stack)
      push(stack, val1 + val2)
}
```

Соответствующая реализация в Java выглядит так:

```
case Op.ADD: {
      long val1 = stackbuf.getLong(sp--);
      long val2 = stackbuf.getLong(sp--);
      stackbuf.putLong(sp++, val1 + val2);
      break;
}
```

Аналогичный код применяется для `SUB`, `MUL`, `DIV`, `MOD`, `LT` и `LE`.

Инструкция `PUSH` принимает операнд из памяти и помещает его в стек. Сложной частью этого (в Unicon и Java, где указатели подделываются) является интерпретация операнда для получения значения из памяти. Это выполняется отдельным методом разыменования. Внутренние вспомогательные функции, такие как `deref()`, являются частью среды выполнения и будут рассмотрены в разделе «*Написание среды выполнения для Jzero*». Unicon-реализация инструкции `PUSH` выглядит следующим образом:

```
Op.PUSH: {
      val := deref(opr, opnd)
      push(stack, val)
}
```

Эквивалентный код Java выглядит так:

```
case Op.PUSH: {
      long val = deref(opr, opnd);
      push(val);
      break;
}
```

Инструкция POP удаляет значение из стека и сохраняет его в ячейке памяти, указанной операндом памяти. Реализация инструкции POP в Unicon выглядит следующим образом:

```
Op.POP: {
      val := pop(stack)
      assign(opnd, val)
}
```

Эквивалентный код Java выглядит так:

```
case Op.POP: {
      long val = pop();
      assign(opnd, val);
      break;
}
```

Инструкция GOTO устанавливает регистр указателя команды в новое место. Как и ожидалось, в Unicon это просто:

```
Op.GOTO: {
        ip := opnd
}
```

Эквивалентный код Java выглядит следующим образом:

```
case Op.GOTO: {
      ip = (int)opnd;
      break;
}
```

Инструкция условного ветвления, BIF (branch-if), удаляет вершину стека. Если вершина стека ненулевая, то она устанавливает регистр указателя инструкции на новое место, например на инструкцию GOTO. В Unicon это реализовано следующим образом:

```
Op.BIF: {
      if pop(stack)~=0 then
              ip := opnd
}
```

Эквивалентный код Java выглядит так:

```
case Op.BIF: {
      if (pop() != 0)
              ip = (int)opnd;
      break;
}
```

Инструкция вызова также похожа на GOTO. Она сохраняет адрес, указывающий, где выполнение должно возобновиться после команды возврата. Функция для вызова указывается в адресе перед *n* параметрами на вершине стека. Неотрицательный адрес в функциональном слоте является местом, куда должен быть установлен указатель инструкции. Если функция отрицательна, это вызов функции среды выполнения number -n. Это показано в следующей Unicon-реализации инструкции CALL:

```
Op.CALL: {
      f := stack[1+opnd]
      if f >= 0 then {
              push(stack, ip)
                      push( stack, bp) # save old ip
                      bp := *stack # set new bp
                      ip := f
              }
      else if f = -1 then do_println()
}
```

Эквивалентный код Java выглядит следующим образом:

```
case Op.CALL: {
      long f;
      f = stackbuf.getLong(
                           sp-8-(int)(8*opnd));
      if (f >= 0) {
              push( ip);
              push( bp);
              bp = sp;
              ip = (int)f;
              }
      else if (f == -1) do_println();
      else { stop("no CALL defined for " + f); }
              break;
}
```

Инструкция возврата также является GOTO, за исключением того, что она переходит в место, которое ранее было сохранено в стеке:

```
Op.RETURN: {
      while *stack > bp do pop(stack)
      bp := pop(stack)
      ip := pop( stack )
}
```

Эквивалентный код Java выглядит так:

```
case Op.RETURN: {
              sp = bp;
              bp = (int)pop();
              ip = (int)pop();
      break;
}
```

Операция выполнения интерпретатора Jzero довольно короткая и милая. Некоторые интерпретаторы байт-кода имеют дополнительные инструкции для ввода/вывода, но мы передаем эти задачи небольшому набору функций, которые могут быть вызваны из сгенерированного кода. Мы кратко рассмотрим эти функции среды выполнения, но сначала разберем метод `main()`, который запускает интерпретатор Jzero из командной строки.

Запуск интерпретатора Jzero

Функция `main()`, запускающая интерпретатор Jzero, находится в модуле с именем *j0x*. Этот лаунчер короткий и приятный. Код Unicon выглядит следующим образом, и его можно найти в файле *j0x.icn*:

```
procedure main(argv)
      if not (filename := argv[1]) then
              stop("usage: j0x file[.j0]")
      if not (filename[-3:0] == ".j0") then argv[1] ||:= ".j0"
      j0machine := j0machine()
      j0machine.init(filename)
      j0machine.interp()
end
```

Соответствующий Java-код в файле *j0x.java* выглядит следующим образом:

```
public class j0x {
      public static void main(String[] argv) {
              if (argv.length < 1) {
                      System.err.println("usage: j0x file[.j0]");
                      System.exit(1);
                      }
              String filename = argv[0];
              if (! filename.endsWith(".j0"))
                      filename = filename + ".j0";
              j0machine.init(filename);
              j0machine.interp();
      }
}
```

Вскоре мы увидим, насколько хорошо работает этот интерпретатор. Но сначала давайте рассмотрим, как встроенные функции включены в среду выполнения Jzero.

Написание среды выполнения для Jzero

В реализации языка программирования среда выполнения – это код, который включается для обеспечения базовых функций, необходимых для выполнения сгенерированного кода. Как правило, чем выше уровень языка и чем больше он удален от аппаратного обеспечения, тем больше среда выполнения. Среда выполнения Jzero настолько мала, насколько это возможно, она только поддерживает несколько внутренних вспомогательных функций, таких как `deref()`, и некоторые основные функции для ввода и вывода. Эти функции написаны на языке реализации (в нашем случае Unicon или Java), а не на языке Jzero. Вот метод `deref()` в Unicon:

```
method deref(reg, od)
      case reg of {
              Op.R_ABS: {
                      if od < finstr then return data.word(od)
                      else return code[od]
                      }
              Op.R_IMM: { return od }
              Op.R_STACK: { return stack[bp+od] }
              default: { stop("deref region ", reg) }
      }
end
```

Каждая область имеет свой код разыменования, который соответствует тому, как эта область хранится. Соответствующая Java-реализация `deref()` выглядит следующим образом:

```
public static long deref(int reg, long od) {
switch(reg) {
case Op.R_ABS: { return codebuf.getLong((int)od); }
case Op.R_IMM: { return od; }
case Op.R_STACK: { return stackbuf.getLong(bp+(int)od); }
default: { stop("deref region " + reg); }
}
return 0;
}
```

В случае встроенных функций мы должны иметь возможность вызывать их из сгенерированного кода Jzero. Реализация встроенных функций, таких как `System.out.println()`, и то, как они вызываются из интерпретатора байт-кода, будет рассмотрено в главе 14 «*Реализация операторов и встроенных функций*». Теперь, наконец, пришло время разобрать, как запустить интерпретатор байт-кода Jzero.

Запуск программы Jzero

На данном этапе нам нужно иметь возможность протестировать наш интерпретатор байт-кода, но мы еще не представили генератор кода, который

создает этот байт-код! По этой причине большая часть тестирования интерпретатора байт-кода отложена до следующей главы, где мы представим генератор кода. Пока же приведем программу *hello world*. Ее исходный код следующий:

```
public class hello {
     public static main(String argv[]) {
             System.out.println("hello");
     }
}
```

Соответствующий байт-код Jzero может выглядеть примерно так. В каждой строке показано одно слово. Строки в шестнадцатеричном формате показывают каждый байт в виде двух шестнадцатеричных цифр. Опкод находится в крайнем левом байте, затем – байт области операнда, а потом – операнд в оставшихся шести байтах:

```
«Jzero!!\0»
«1.0\0\0\0\0\0»
0x0000040000000000
«hello\0\0\0»
0x0902380000000000          push main
0x0B02000000000000          call 0
0x0100000000000000          halt
0x0902FFFFFFFFFFFF          push -1 (println)
0x0902180000000000          push "hello"
0x0B02010000000000          call 1
0x0C02000000000000          return 0
```

Если это записано в двоичном виде в файл с именем *hello.j0*, то выполнение команды *j0x hello* выведет `hello`, как и ожидалось. Этот маленький, но конкретный пример должен возбудить ваш аппетит перед гораздо более интересными примерами, которые мы создадим в следующей главе. Пока же сравните простоту Jzero с некоторыми из более интересных возможностей, которые можно найти, изучив интерпретатор байт-кода Unicon.

Изучение iconx, интерпретатора байт-кода Unicon

Язык Unicon и его предшественник, Icon, имеют общую архитектуру и реализацию в виде интерпретатора байт-кода и программы среды выполнения под названием *iconx*. По сравнению с интерпретатором байт-кода Jzero в предыдущем разделе, *iconx* больше и сложнее, а также имеет преимущество реального использования в течение длительного времени. По сравнению с виртуальной машиной Java, *iconx* мал и прост, и он относительно доступен для изучения. Подробное описание *iconx* можно найти в *сборнике «Реализация Icon и Unicon»* (*The Implementation of Icon and Unicon: a Compendium*). Данный раздел можно рассматривать как краткое введение в эту работу.

Понимание целенаправленного байт-кода

Unicon имеет необычный байт-код. Краткий пример был приведен ранее в этой главе, в разделе «*Понимание, что такое байт-код*». Этот язык ориентирован на достижение цели. Все выражения успешны или терпят неудачу. Многие выражения, называемые **генераторами**, могут выдавать дополнительные результаты по требованию, когда окружающее выражение терпит неудачу. **Перебор с возвратом** встроен в интерпретатор байт-кода, чтобы сохранить состояние таких выражений-генераторов и возобновить их позже, при необходимости.

Оценка выражений, направленных на достижение цели, может быть реализована многими способами. Набор инструкций байт-кода Unicon, который он в значительной степени унаследовал от Icon, имеет очень необычную семантику, которая отражает способ достижения цели, найденный в исходном языке. Части инструкций помечены информацией, указывающей им, куда двигаться в случае неудачи. В пределах таких частей инструкций состояние генераторов сохраняется в стеке спагетти, и если выражение терпит неудачу, возобновляется работа последнего приостановленного генератора.

Сохранение информации о типе во время выполнения

В Unicon переменные могут содержать значения любого типа, и значения «знают», к какому типу они относятся. Это способствует гибкости языка и соответствует полиморфному коду[1], но снижает скорость выполнения и требует больше памяти для работы. В реализации языка C все переменные, включая переменные, хранящиеся в структурах, таких как списки или записи, представлены в *дескрипторе*, объявленном как тип `struct descrip`. Этот тип содержит два слова – *dword*, или слово дескриптора, для информации о типе, и *vword*, или слово значения, для значения. Реализация данной структуры на языке C показана здесь:

```
struct descrip {
      word dword;
      union {
              word integr;
              double realval;
              char *sptr;
              union block *bptr;
              dptr descptr;
              } vword;
};
```

Строки имеют специальный формат `dword`, для строки слово информации о типе содержит длину строки, знаковый бит этого слова является флагом, который указывает, является ли значение не строкой в том случае, если при-

[1] В вычислительной технике полиморфный код – это код, который использует полиморфный движок для мутирования, сохраняя при этом исходный алгоритм нетронутым, то есть код меняется сам при каждом запуске, но функция кода (его семантика) вообще не меняется. – *Прим. перев.*

сутствует код информации о типе. Для чисел слово `vword` дескриптора также имеет специальный формат. Для целых и вещественных чисел слово значения содержит значение, для всех других типов это слово является указателем на значение. Используются три различных вида указателей, и указатель на блок объединения может указывать на любой из нескольких десятков или около того различных типов данных Unicon. Какое поле блока `vword` использовать, решается во всех случаях путем просмотра `dword`.

Выборка, декодирование и выполнение инструкций

Для байт-кода Unicon цикл fetch-decode-execute содержится в функции языка C под названием *interp()*. В соответствии с этой главой она состоит из бесконечного цикла, внутри которого находится оператор переключателя. Одно из различий между инструкциями Unicon и Jzero, как описано в этой главе, заключается в том, что опкоды Unicon обычно имеют размер в полслова, и если они содержат операнд, то он, как правило, занимает целое слово, следующее за этим полусловным опкодом. Поскольку многие инструкции не имеют операнда, это может сделать код более компактным, а так как операнды являются полными словами, они могут содержать собственный указатель языка C, а не смещение относительно базового указателя для данной области памяти. Байт-код Unicon вычисляется компилятором и хранится в исполняемом файле на диске с использованием смещений, и при первом выполнении исполняемого файла смещения преобразуются в указатели, а опкод преобразуется, чтобы указать, что теперь он содержит указатели. Этот умный самомодифицирующийся код создает дополнительные проблемы для безопасности потоков, что означает, что байт-код не может быть выполнен из постоянной памяти или памяти только для чтения.

Создание остальной части среды выполнения

Еще одно различие между Iconx и интерпретатором Jzero, представленным в этой главе, заключается в том, что интерпретатор байт-кода Unicon имеет огромную среду выполнения, состоящую из многочисленных сложных возможностей, таких как высокоуровневая графика и сетевое взаимодействие. Там, где интерпретатор байт-кода Jzero может составлять 80 % кода, а 20 % остаются среде выполнения, функция *interp()* в ядре Unicon может составлять только 5 % кода, а остальные 95 % – это реализация множества встроенных функций и операторов. Эта среда выполнения написана на языке под названием RTL, который является своего рода супермножеством языка C со специальными возможностями для поддержки системы типов Unicon, привязки типов и правил автоматического преобразования типов.

В этом разделе было представлено краткое введение в реализацию интерпретатора байт-кода Unicon. Вы увидели, что интерпретаторы байт-кода языков программирования часто являются более интересными и сложными, чем интерпретатор Jzero. Они могут включать в себя новые структуры управления, высокоуровневые и/или специфические для данной области типы данных и многое другое.

Заключение

В этой главе были представлены основные элементы интерпретаторов байт-кода. Знание того, как реализовать интерпретатор байт-кода, позволит вам создавать гибкий код, не заботясь о наборах аппаратных инструкций, регистрах и режимах адресации.

Во-первых, вы узнали, что определение набора инструкций включает опкоды и правила для обработки любых операндов в этих инструкциях. Вы узнали, как реализовать общую семантику стековой машины, а также инструкции байт-кода, соответствующие специфическим для данной области особенностям языка. Затем познакомились с тем, как читать и выполнять файлы байт-кода, включая взаимозаменяемую работу с последовательностями байтов и слов как в Unicon, так и в Java.

Учитывая существование интерпретатора байт-кода, в следующей главе мы обсудим генерацию байт-кода из промежуточного кода, чтобы мы могли запускать программы, скомпилированные с помощью нашего компилятора. Учитывая существование интерпретатора байт-кода, в следующей главе мы обсудим генерацию байт-кода из промежуточного кода, чтобы мы могли запускать программы, скомпилированные с помощью нашего компилятора.

Вопросы

1. Интерпретатор байт-кода может использовать набор инструкций, содержащий до трех адресов (операндов) на инструкцию, например трехадресный код. В отличие от этого, интерпретатор Jzero использует ноль или один операнд на инструкцию. Каковы плюсы и минусы использования трехадресного кода в интерпретаторе байткода, например в промежуточном коде?
2. В реальных процессорах и во многих интерпретаторах байт-кода, основанных на языке C, адреса байт-кода представлены буквенными машинными адресами. Однако интерпретаторы байт-кода, которые были показаны в этой главе, реализуют адреса байт-кода как позиции или смещения в пределах выделенных блоков памяти. Находится ли язык программирования, не имеющий типа указателя данных, в фатально невыгодном положении при реализации интерпретатора байт-кода, по сравнению с языком, который поддерживает типы указателей данных?
3. Если код представлен в памяти как неизменяемое строковое значение, какие ограничения накладывает это на реализацию интерпретатора байт-кода?

Глава 12

Генерация байт-кода

В этой главе мы продолжим генерацию кода, взяв промежуточный код из главы 9 «*Генерация промежуточного кода*», и сгенерируем из него байт-код. Когда вы переводите промежуточный код в формат, который будет выполняться, вы генерируете **окончательный код**. Обычно это происходит во время компиляции, но может происходить и позже – во время линковки, загрузки или выполнения. Мы будем генерировать байт-код обычным способом – во время компиляции. Эта глава и следующая глава о генерации собственного кода представляют вам две формы конечного кода, между которыми вы можете выбирать.

Перевод промежуточного кода в байт-код выполняется путем прохождения по списку промежуточных инструкций, переводя каждую инструкцию промежуточного кода в одну или более инструкций байт-кода. Для обхода списка используется простой цикл, причем для каждой инструкции промежуточного кода используется свой фрагмент кода. Несмотря на то что цикл, используемый в этой главе, прост, генерация окончательного кода остается очень важным основным навыком, который вы должны приобрести, чтобы воплотить в жизнь ваш новый язык программирования.

В этой главе рассматриваются следующие основные темы:

- преобразование промежуточного кода в байт-код Jzero;
- сравнение ассемблера байт-кода с двоичными форматами;
- линковка, загрузка и включение среды выполнения;
- пример Unicon – генерация байт-кода в icont.

С помощью функциональности, которую мы создадим в этой главе, мы сможем генерировать код, работающий с интерпретатором байт-кода, представленным в предыдущей главе.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch12*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/3oR6zGt*.

Преобразование промежуточного кода в байт-код Jzero

Генератор промежуточного кода Jzero из главы 9 «*Генерация промежуточного кода*» обошел дерево и создал список промежуточного кода как синтезированный атрибут для каждого узла дерева, названный `icode`. Промежуточным кодом для всей программы является атрибут `icode` корневого узла дерева синтаксиса. В данном разделе мы будем использовать этот список для создания нашего выходного байт-кода. Чтобы сгенерировать байт-код, метод `gencode()` в классе *j0* вызывает новый метод этого класса, названный `bytecode()`, и передает ему промежуточный код в файле `root.icode` в качестве входных данных. Метод Unicon `gencode()`, который вызывает эту функциональность в файле *j0.icn*, выглядит следующим образом. Две выделенные строки в конце фрагмента кода добавлены для генерации байт-кода, проверяемого простым выводом текста:

```
method gencode(root)
      root.genfirst()
      root.genfollow()
      root.gentargets()
      root.gencode()
      bcode := bytecode(root.icode)
      every (! (\bcode)).print()
end
```

Метод `bytecode()` принимает список `icode`, а его возвращаемым значением является список объектов класса *byc*. В этом примере полученный байт-код выводится в текстовом виде, в готовом компиляторе по умолчанию обычно выводится двоичный формат, а компилятор Jzero будет поддерживать оба формата. Соответствующий код Java для метода `gencode()` показан в следующем фрагменте кода. Генерация вывода, выполняемая в операторе `if`, в этом случае немного более сложна:

```
public static void gencode(root) {
      root.genfirst();
      root.genfollow();
      root.gentargets();
      root.gencode();
      ArrayList<byc> bcode = bytecode(root.icode);
      if (bcode != null) {
              for (int i = 0; i < bcode.size(); i++)
                      bcode.get(i).print();
              }
}
```

Каждый элемент списка `bcode` представляет собой инструкцию байт-кода, для которой нам нужен класс. Назовем его `byc`, сокращенно от bytecode. Теперь давайте рассмотрим код этого класса.

Добавление класса для инструкций байт-кода

Мы можем представить наш байт-код буквально, используя 64-битное слово в формате, представленном в главе 11 «*Интерпретаторы байт-кода*». Представление инструкций байт-кода в виде объектов облегчает вывод как в виде человекочитаемого текста, так и в двоичной форме. Представление в виде списка объектов также делает более удобным анализ для оптимизации окончательного кода.

Класс `byc` похож на класс `tac`, но вместо **кода операции (опкода)** и полей для трех операндов он представляет только опкод, область операнда и операнд, если он присутствует, как описано в главе 11. Класс также содержит несколько методов, включая методы для печати в текстовой и двоичной формах. Методы `print()` и `printb()` будут представлены в разделе «*Сравнение ассемблера байт-кода с двоичными форматами*». Ниже приводится описание класса Unicon `byc` из файла *byc.icn*:

```
class byc(op, opreg, opnd)
      method print() … end
      method printb() … end
      method addr(a) … end
initially(o, a)
        op := o; addr(a)
end
```

Соответствующий класс Java в файле *byc.java* выглядит следующим образом:

```
public class byc {
      int op, opreg;
      long opnd;
      public byc(int o, address a) {
              op=o; addr(a);
      }
      public void print() { … }
      public void printb() { … }
      public void addr(address a) { … }
}
```

Как часть класса `byc` нам нужен метод `addr()`, который обеспечивает преобразование из адресов трехадресного кода в адреса байт-кода. Давайте рассмотрим это далее.

Соответствие адресов промежуточного кода адресам байт-кода

Хотя наборы инструкций совершенно разные, адреса в промежуточном и окончательном кодах обозначают примерно одно и то же. Поскольку мы разрабатываем и промежуточный код, и байт-код, мы можем определить адреса в байт-коде так, чтобы они были намного ближе к адресам промежуточного кода, чем это будет в следующей главе при переходе от промежуточного кода к собственному коду. В любом случае, область и смещение от адреса из главы 9 «*Генерация промежуточного кода*» должны быть отображены в `opreg` и `opnd` класса `byc`. Это

обрабатывается методом addr() в классе byc, который принимает экземпляр класса address в качестве параметра и устанавливает opreg и opnd. Код Unicon в файле *byc.icn* выглядит следующим образом:

```
method addr(a)
      if /a then opreg := Op.R_NONE
      else case a.region of {
      "loc": { opreg := Op.R_STACK; opnd := a.offset }
      "glob": { opreg := Op.R_ABS; opnd := a.offset }
      "const": { opreg := Op.R_ABS; opnd := a.offset }
      "lab": { opreg := Op.R_ABS; opnd := a.offset }
      "obj": { opreg := Op.R_HEAP; opnd := a.offset }
      "imm": { opreg := Op.R_IMM; opnd := a.offset }
      }
end
```

Соответствующий метод Java в файле *byc.java* показан ниже:

```
public void addr(address a) {
      if (a == null) opreg = Op.R_NONE;
      else switch (a.region) {
      case "loc": { opreg = Op.R_STACK; opnd = a.offset; break; }
      case "glob": { opreg = Op.R_ABS; opnd = a.offset; break; }
      case "const": { opreg = Op.R_ABS; opnd = a.offset; break; }
      case "lab": { opreg = Op.R_ABS; opnd = a.offset; break; }
      case "obj": { opreg = Op.R_HEAP; opnd = a.offset; break; }
      case "imm": { opreg = Op.R_IMM; opnd = a.offset; break; }
      }
}
```

При наличии класса byc необходима еще одна вспомогательная функция, для того чтобы сформулировать метод генератора кода bytecode(). Нам нужен удобный производственный метод для генерации инструкций байт-кода и присоединения их к списку bcode. Мы назовем этот метод bgen().

Метод bgen() в классе *j0* аналогичен gen() из класса *tree*, он производит одноэлементный список, содержащий экземпляр byc. Код Unicon выглядит следующим образом:

```
method bgen(o, a)
        return [byc(o, a)]
end
```

Соответствующая реализация в Java выглядит так:

```
public ArrayList<byc> bgen(int o, address a) {
      ArrayList<byc> L = new ArrayList<byc>();
      byc b = new byc(o, a);
      L.add(b);
      return L;
}
```

Теперь, наконец, пришло время представить генератор байт-кода.

Реализация метода генератора байт-кода

Далее показана Unicon-реализация метода `bytecode()` в классе *j0*. Реализация должна заполнить одну ветвь случая для каждого опкода в трехадресном наборе инструкций, приведенном в главе 9 «*Генерация промежуточного кода*». Случаев будет очень много, поэтому мы представим каждый из них отдельно, начиная с этого:

```
method bytecode(icode)
      rv := []
      every i := 1 to *\icode do {
              instr := icode[i]
              case instr.op of {
                      "ADD": { ... append translation of ADD to return val }
                      "SUB": { ... append translation of SUB to return val }
                      ...
                      }
              }
      return rv
end
```

Здесь показана Java-реализация `bytecode()`:

```
public static ArrayList<byc> bytecode(
ArrayList<tac> icode)
      {
      ArrayList<byc> rv = new ArrayList<byc>();
         for(int i=0; i<icode.size(); i++) {
              tac instr = icode.get(i);
              switch(instr.op) {
              case "ADD": { ... append translation of ADD to rv }
              case "SUB": { ... append translation of SUB to rv }
                      ...
                      }
              }
      return rv;
}
```

В рамках этого метода `bytecode()` мы теперь можем предоставить переводы для каждой из трехадресных инструкций. Начнем с простых выражений.

Генерация байт-кода для простых выражений

Различные случаи для каждого трехадресного опкода имеют много общих элементов, таких как, например, перемещение значений из памяти в стек оценки. Случай сложения, возможно, демонстрирует наиболее распространенную схему перевода. В Unicon сложение обрабатывается следующим образом:

```
«ADD»: {
      bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
              j0.bgen(Op.PUSH, instr.op3) ||| j0.bgen(Op.ADD) |||
              j0.bgen(Op.POP, instr.op1)
}
```

Этот код считывает операнд 2 и операнд 3 из памяти и помещает их в стек. Фактически инструкция ADD работает полностью из стека. Результат затем удаляется из стека и помещается в операнд 3. В Java реализация сложения состоит из следующего кода:

```
case "ADD": {
      bcode.addAll(j0.bgen(Op.PUSH, instr.op2));
      bcode.addAll(j0.bgen(Op.PUSH, instr.op3));
      bcode.addAll(j0.bgen(Op.ADD, null));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      break;
}
```

Набор инструкций промежуточного кода, представленный в главе 9 «*Генерация промежуточного кода*», определяет 19 трехадресных инструкций, которые должны быть переведены в окончательный код. Схема генерации окончательного кода, проиллюстрированная предыдущей инструкцией ADD, используется для других арифметических инструкций. Для унарного оператора, такого как NEG, схема немного упрощается, как мы видим здесь:

```
«NEG»: {
      bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
              j0.bgen(Op.NEG) ||| j0.bgen(Op.POP, instr.op1)
}
```

В Java реализация отрицания состоит из следующего кода:

```
case «NEG»: {
      bcode.addAll(j0.bgen(Op.PUSH, instr.op2));
      bcode.addAll(j0.bgen(Op.NEG, null));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      break;
}
```

Еще более простая инструкция, такая как ASN, может быть достойна специального выделения при проектировании набора инструкций вашей машины байт-кода, но для стековой машины вы можете придерживаться того же сценария и еще больше упростить предыдущий шаблон, как показано в следующем фрагменте кода:

```
«ASN»: {
      bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
              j0.bgen(Op.POP, instr.op1)
}
```

В Java реализация присваивания может выглядеть следующим образом:

```
case «ASN»: {
      bcode.addAll(j0.bgen(Op.PUSH, instr.op2));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      break;
}
```

Код, состоящий из арифметических выражений и присваиваний, является ядром большинства языков программирования. Теперь пришло время рассмотреть генерацию кода для некоторых других инструкций промежуточного кода, начиная с тех, которые используются для обработки указателей.

Генерация кода для обработки указателей

Три из трехадресных инструкций промежуточного кода, определенных в главе 9 «*Генерация промежуточного кода*», относятся к использованию указателей: `ADDR`, `LCON` и `SCON`. Инструкция `ADDR` превращает адрес в памяти в часть данных, которыми можно манипулировать для выполнения таких операций, как арифметика указателей. Она перемещает свой операнд, ссылку на адрес в одной из областей памяти, как если бы это было значение в режиме непосредственного ввода, как показано в следующем фрагменте кода:

```
"ADDR": {
      bcode |||:= j0.bgen(Op.ADDR, instr.op2)
      bcode |||:= j0.bgen(Op.POP, instr.op1)
}
```

В Java реализация инструкции `ADDR` состоит из этого кода:

```
case «ADDR»: {
      bcode.addAll(j0.bgen(Op.ADDR, instr.op2));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      break;
}
```

Инструкция `LCON` читает из памяти, на которую указывает другая память, как показано здесь:

```
«LCON»: {
      bcode |||:= j0.bgen(Op.LOAD, instr.op2)
      bcode |||:= j0.bgen(Op.POP, instr.op1)
}
```

В Java реализация инструкции `LCON` состоит из следующего кода:

```
case «LCON»: {
      bcode.addAll(j0.bgen(Op.LOAD, instr.op2));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      break;
}
```

Инструкция SCON записывает в память, на которую указывает другая память, как показано ниже:

```
«SCON»: {
      bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
              j0.bgen(Op.STORE, instr.op1)
}
```

В Java реализация инструкции SCON состоит из следующего кода:

```
case «SCON»: {
      bcode.addAll(j0.bgen(Op.PUSH, instr.op2));
      bcode.addAll(j0.bgen(Op.STORE, instr.op1));
      break;
}
```

Эти инструкции важны для поддержки структурированных типов данных, таких как массивы. Теперь рассмотрим генерацию байт-кода для потока управления, начиная с семейства инструкций GOTO.

Генерация байт-кода для безусловных и условных переходов

Семь инструкций промежуточного кода относятся к инструкциям условного и безусловного перехода. Самой простой из них является инструкция безусловного перехода GOTO. Инструкция GOTO присваивает новое значение регистру указателя инструкции. Неудивительно, что байт-код GOTO является реализацией трехадресной инструкции GOTO. Код Unicon для преобразования промежуточного кода GOTO в байт-код GOTO показан здесь:

```
«GOTO»: {
        bcode |||:= j0.bgen(Op.GOTO, instr.op1)
}
```

В Java реализация инструкции GOTO состоит из следующего кода:

```
case «GOTO»: {
      bcode.addAll(j0.bgen(Op.GOTO, instr.op1));
      break;
}
```

Инструкции условного перехода в трехадресном коде преобразуются в более простые инструкции окончательного кода. Для набора инструкций байт-кода, представленных в предыдущей главе, это означает помещение операндов в стек перед инструкцией условного перехода байт-кода. Реализация инструкции BLT в Unicon выглядит следующим образом:

```
«BLT»: {
     bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
             j0.bgen(Op.PUSH, instr.op3) ||| j0.bgen(Op.LT) |||
             j0.bgen(Op.BIF, instr.op1)
}
```

В Java реализация генерации байт-кода для инструкции BLT состоит из следующего кода:

```
case «BLT»: {
     bcode.addAll(j0.bgen(Op.PUSH, instr.op2));
     bcode.addAll(j0.bgen(Op.PUSH, instr.op3));
     bcode.addAll(j0.bgen(Op.LT, null));
     bcode.addAll(j0.bgen(Op.BIF, instr.op1));
     break;
}
```

Эта схема используется для нескольких трехадресных инструкций с более простым кодом, используемым для BIF и BNIF. Теперь рассмотрим более сложные формы передачи управления, которые связаны с вызовами и возвратами методов.

Генерация кода для вызовов методов и возвратов

Три из трехадресных инструкций обрабатывают очень важную тему вызовов и возвратов функций и методов. Последовательность из нуля или более инструкций PARM помещает значения в стек, инструкция CALL выполняет вызов метода, а инструкция RET возвращает значения из метода вызывающей стороне. Этот вызов трехадресного кода должен быть преобразован в набор инструкций байт-кода стековой машины, который требует, чтобы адрес вызываемой процедуры был вставлен (в слот стека, где будет находиться возвращаемое значение) до того, как будут вставлены другие параметры. Мы могли бы вернуться и изменить наш трехадресный код, чтобы он лучше соответствовал стековой машине, но тогда он не будет так хорошо подходить для собственного кода x86_64.

Инструкция PARM – это просто помещение в стек, за исключением случаев, когда это первый параметр и требуется адрес процедуры, как показано в следующем фрагменте кода:

```
«PARM»: {
      if /methodAddrPushed then {
              every j := i+1 to *icode do
                      if icode[j].op == "CALL" then {
                              bcode |||:= j0.bgen(Op.PUSH, icode[j].op2)
                              break
                      }
              methodAddrPushed := 1
              }
      bcode |||:= j0.bgen(Op.PUSH, instr.op1)
}
```

Каждый цикл ищет ближайшую инструкцию CALL и вставляет адрес ее метода. В Java реализация инструкции PARM аналогична, как видно здесь:

```
case «PARM»: {
      if (methodAddrPushed == false) {
              for(int j = i+1; j<icode.length; j++) {
                      tac callinstr = icode.get(j);
                      if (callinstr.op.equals("CALL")) {
                              bcode.addAll(j0.bgen(Op.PUSH, callinstr.op2));
                              break;
                      }
              }
      }
      bcode.addAll(j0.bgen(Op.PUSH, instr.op1));
      break;
}
```

При предварительном вводе адреса метода инструкция CALL действует прямолинейно. После вызова значение *op1* в трехадресном коде извлекается из стека, как и в других выражениях. Поле источника *op2* – это адрес метода, который использовался до первой инструкции PARM. Поле источника *op3* задает количество параметров, которое используется в качестве операнда в CALL. Байт-код показан в следующем фрагменте кода:

```
«CALL»: {
      bcode |||:= j0.bgen(Op.CALL, instr.op3)
      bcode |||:= j0.bgen(Op.POP, instr.op1)
      methodAddrPushed := &null
}
```

В Java реализация инструкции CALL состоит из следующего кода:

```
case «CALL»: {
      bcode.addAll(j0.bgen(Op.CALL, instr.op3));
      bcode.addAll(j0.bgen(Op.POP, instr.op1));
      methodAddrPushed = false;
      break;
}
```

Unicon-реализация инструкции RETURN выглядит так:

```
«RETURN»: {
        bcode |||:= j0.bgen(Op.RETURN, instr.op1)
}
```

В Java реализация инструкции RETURN состоит из следующего кода:

```
case «RETURN»: {
     bcode.addAll(j0.bgen(Op.RETURN, instr.op1));
     break;
}
```

Генерация кода для вызовов методов и возвратов не представляет особой сложности. Теперь давайте рассмотрим, как обрабатывать псевдоинструкции трехадресного кода.

Обработка меток и других псевдоинструкций промежуточного кода

Псевдоинструкции не преобразуются в код, но они присутствуют в связанном списке трехадресных инструкций и требуют внимания в окончательном коде. Наиболее распространенной и очевидной псевдоинструкцией является **метка**. Если окончательный код генерируется в формате человекочитаемого ассемблера, метки могут быть сгенерированы, как есть. Хотя LAB и Op.LABEL не являются инструкциями, они являются элементами в списках промежуточного кода и сгенерированного байт-кода соответственно. В Unicon это выражается следующим образом:

```
«LAB»: {
        bcode |||:= j0.bgen(Op.LABEL, instr.op1)
}
```

Вот соответствующий код в Java:

```
case «LAB»: {
     bcode.addAll(j0.bgen(Op.LABEL, instr.op1));
     break;
}
```

Для окончательного кода, сгенерированного в двоичном формате, метки требуют дополнительной обработки, поскольку они должны быть заменены соответствующими смещениями байтов или адресами.

Так как метка – это имя или псевдоним для адреса конкретной инструкции, в двоичном формате байт-кода она обычно заменяется смещением байта в той или иной форме. По мере генерации окончательного кода создается таблица, содержащая соответствие между метками и смещениями.

В нескольких предыдущих разделах была получена структура данных, содержащая представление байт-кода, а затем показано, как преобразуются раз-

личные трехадресные инструкции. Теперь перейдем к созданию кода в текстовом и двоичном форматах.

Сравнение ассемблера байт-кода с двоичными форматами

Машины байт-кода имеют тенденцию использовать более простые форматы, чем собственный код, где двоичные объектные файлы являются нормой. Некоторые машины байт-кода, такие как Python, скрывают свой формат байт-кода полностью или делают его необязательным. Другие, такие как Unicon, используют человекочитаемый ассемблероподобный текстовый формат для скомпилированных модулей. В случае с Java они, кажется, сделали все возможное, чтобы избежать предоставления ассемблера, с целью усложнения другим языкам работы со своей **виртуальной машиной (ВМ)**.

В случае с Jzero и его машиной у нас есть сильные стимулы к тому, чтобы все было как можно проще. Класс `byc` определяет два метода вывода – `print()` для текстового формата и `printb()` для двоичного формата. Вы можете сами решить, какой из них вам больше нравится.

Вывод байт-кода в формате ассемблера

Метод `print()` в классе *byc* аналогичен методу, используемому в классе *tac*. Для каждого элемента списка создается одна строка вывода. Unicon-реализация метода `print()` в классе *byc* показана здесь. Параметр `f`, который по умолчанию используется для стандартного вывода, задает имя:

```
method print(f:&output)
      if op === LABEL then write(f, addrof(), ":")
      else write(f, nameof(), " ", addrof())
end
```

Соответствующая реализация Java показана ниже. Перегрузка метода используется для того, чтобы сделать параметр необязательным:

```
public void print(PrintStream f) {
      if (op == LABEL) f.println(addrof() + ":");
      else f.println("\t" + nameof() + " " + addrof());
}
public void print() { print(System.out); }
```

Текстовые методы `print()` просто перекладывают большую часть работы на вспомогательные методы, которые выдают человекочитаемые представления опкода и операнда. Код Unicon для метода `nameof()`, который преобразует опкод в строку, показан в следующем примере:

```
method nameof()
      static opnames
      initial opnames := table(Op.HALT, "halt", Op.NOOP,
```

```
            "noop",
            Op.ADD, "add", Op.SUB, "sub", Op.MUL, "mul",
            Op.DIV, "div", Op.MOD, "mod", Op.NEG, "neg",
            Op.PUSH, "push", Op.POP, "pop", Op.CALL, "call",
            Op.RETURN, "return", Op.GOTO, "goto", Op.BIF, "bif",
            Op.LT, "lt", Op.LE, "le", Op.GT, "gt", Op.GE, "ge",
            Op.EQ, "eq", Op.NEQ, "neq", Op.LOCAL, "local",
            Op.LOAD, "load", Op.STORE, "store")
      return opnames[op]
end
```

Соответствующий код Java, показанный здесь, использует `HashMap`:

```
      static HashMap<Short,String> ops;
      static { ops = new HashMap<>();
            ops.put(Op.HALT,"halt"); ops.put(Op.NOOP,"noop");
            ops.put(Op.ADD,"add"); ops.put(Op.SUB,"sub");
            ops.put(Op.MUL,"mul"); ops.put(Op.DIV, "div");
            ops.put(Op.MOD,"mod"); ops.put(Op.NEG, "neg");
            ops.put(Op.PUSH,"push"); ops.put(Op.POP, "pop");
            ops.put(Op.CALL, "call"); ops.put(Op.RETURN, "return");
            ops.put(Op.GOTO, "goto"); ops.put(Op.BIF, "bif");
            ops.put(Op.LT, "lt"); ops.put(Op.LE, "le");
            ops.put(Op.GT, "gt"); ops.put(Op.GE, "ge");
            ops.put(Op.EQ, "eq"); ops.put(Op.NEQ, "neq");
            ops.put(Op.LOCAL, "local"); ops.put(Op.LOAD, "load");
            ops.put(Op.STORE, "store");
}
public String nameof() {
        return opnames.get(op);
}
```

Другой вспомогательной функцией, вызываемой из метода `print()`, является метод `addrof()`, который выводит человекочитаемое представление адреса на основе области операнда и полей операнда. Его реализация в Unicon показана ниже:

```
method addrof()
      case opreg of {
            Op.R_NONE: return ""
            Op.R_ABS: return "@"+
            java.lang.Long.toHexString(opnd);
            Op.R_IMM: return string(opnd)
            Op.R_STACK: return "stack:" + String.valueof(opnd)
            Op.R_HEAP: return "heap:" + String.valueof(opnd)
            default: return string(opreg) ":" || opnd
            }
end
```

Соответствующий Java-код для `addrof()` показан здесь:

```
public String addrof() {
      switch (opreg) {
      case Op.R_NONE: return "";
      case Op.R_ABS: return "@"+
              java.lang.Long.toHexString(opnd);
      case Op.R_IMM: return String.valueOf(opnd);
      case Op.R_STACK: return "stack:" + String.valueOf(opnd);
      case Op.R_HEAP: return "heap:" + String.valueOf(opnd);
      }
      return String.valueOf(opreg)+":"+String.valueOf(opnd);
}
```

Теперь давайте посмотрим на соответствующий двоичный вывод.

Вывод байт-кода в двоичном формате

Методы `printb()` организованы аналогично, но там, где `print()` нужны имена объектов, `printb()` нужно поставить все биты в ряд и вывести двоичное слово. Его реализация в Unicon показана ниже:

```
method printb(f:&output)
      writes(f, "\t", char(op), char(opregn))
      x := opnd
      every !6 do {
              writes(f, char(iand(x, 255)))
              x := ishift(x, -8)
              }
end
```

Соответствующая Java-реализация `printb()` такая:

```
public void printb(PrintStream f) {
      long x = opnd;
      f.print((byte)op);
      f.print((byte)opreg);
         for(int i = 0; i < 6; i++) {
              f.print((byte)(x & 0xff));
              x = x>>8;
              }
}
public void printb() { printb(System.out); }
```

В этом разделе мы рассмотрели, как вывести наш код на внешний носитель. Контраст между текстовыми и двоичными форматами был разительным, причем двоичные форматы были немного более трудоемкими, по крайней мере с человеческой точки зрения. Теперь давайте рассмотрим другие вопросы, необходимые для выполнения программы, помимо сгенерированного кода. Сюда входит связь сгенерированного кода с другим кодом, особенно со средой выполнения.

Линковка, загрузка и включение среды выполнения

В отдельно скомпилированном языке собственного кода выходной двоичный формат, полученный на этапе компиляции, не является исполняемым. Машинный код выводится в объектном файле, который должен быть связан (линкован) с другими модулями, и адреса между ними должны быть разрешены, чтобы сформировать исполняемый файл. На этом этапе включается среда выполнения путем связывания в объектных файлах, которые поставляются с компилятором, а не только в других модулях, написанных пользователем. В старые времена загрузка полученного исполняемого файла была тривиальной операцией. В современных системах это сложнее из-за таких вещей, как разделяемые объектные библиотеки.

Реализация байт-кода часто имеет существенные отличия от только что описанной традиционной модели. Java не выполняет шаг линковки, или, возможно, вы можете сказать, что он компонует код во время загрузки. Среду выполнения Java можно считать четко разделенной между большим числом функциональностей, встроенных в интерпретатор **Java VM (JVM)**, и еще большим их числом как в байт-коде, так и в собственном коде, которые должны быть загружены для запуска различных частей стандартного языка Java. С точки зрения стороннего наблюдателя, одной из удивительных вещей в Java является огромное количество операторов импорта, которые разработчик должен размещать в начале каждого файла, использующего что-либо из стандартных библиотек Java.

В случае с Jzero жесткие ограничения позволяют сделать все это максимально простым. Не существует отдельной компиляции или линковки. Загрузка предельно проста и была рассмотрена в предыдущей главе. Среда выполнения встроена в интерпретатор байт-кода, что является еще одной уловкой, чтобы позволить языку избежать линковки. Теперь давайте рассмотрим генерацию байт-кода в Unicon, еще одну реализацию байт-кода, которая работает совсем иначе, чем Java или Jzero.

Пример Unicon – генерация байт-кода в icont

Формат вывода скомпилированного байт-кода Unicon – это человекочитаемый текст в файлах *ucode*. Такие файлы сначала генерируются, а затем линкуются и преобразуются в двоичный формат *icode* программой на языке C под названием *icont*, которая вызывается транслятором Unicon. Программа *icont* играет роль генератора кода, ассемблера и линковщика для формирования законченной программы байт-кода в двоичном формате. Вот некоторые подробности.

Функция языка C `gencode()` программы *icont* считывает строки текста в файле *ucode* и преобразует их в двоичный формат, следуя схеме, показанной здесь. Существует интересное сходство между этим псевдокодом и циклом fetch-decode-execute, используемым в интерпретаторе байт-кода. Здесь мы получаем текстовый байт-код со входа, декодируем опкод и записываем двоичный байт-код с небольшими различиями в формате в зависимости от байт-кода:

```
void gencode() {
      while ((op = getopc(&name)) != EOF) {
              switch(op) {
              .   .   .
              case Op_Plus:
                      newline();
                      lemit(op, name);
                      break;
              .   .   .
              }
      }
}
```

Функция `lemit()` и около семи связанных с ней функций с префиксом `lemit*()` используются для добавления байт-кода в непрерывный массив байтов в двоичном формате. Метки, связанные с инструкциями, преобразуются в смещения байтов. Прямые ссылки на метки, которые еще не встречались, помещаются в связанные списки и возвращаются позже, когда будет встречена целевая метка. Код на языке C для `lemitl()` выдает инструкцию с меткой, как показано здесь:

```
static void lemitl(int op, int lab, char *name)
      {
      misalign();
      if (lab >= maxlabels)
              labels = (word *) trealloc(labels, NULL, &maxlabels,
              sizeof(word), lab - maxlabels + 1, "labels");
      outop(op);
      if (labels[lab] <= 0) { /* forward reference */
      outword(labels[lab]);
      labels[lab] = WordSize - pc;
      }
      else outword(labels[lab] - (pc + WordSize));
      }
```

Как можно догадаться по названию, функция `misalign()` генерирует инструкции без действия (no-op instructions), чтобы гарантировать, что инструкции начинаются на границах слов. Первый оператор `if` увеличивает таблицу массивов, если это необходимо. Второй оператор `if` обрабатывает метку, которая является прямой ссылкой на инструкцию, которая еще не существует, вставляя ее в начало связанного списка инструкций, которые должны быть возвращены, когда все инструкции будут существовать.

Компоновка двоичного кода выполняется макросами *outop()* и *outword()*, которые выводят опкод и операнд, имеющие длину целого числа и слова соответственно. Эти макросы могут быть определены по-разному на разных платформах, но на большинстве машин они просто вызывают функции с именами `intout()` и `wordout()`. Обратите внимание, что в следующем фрагменте кода двоичный код представлен в машинном формате и отличается на **центральных процессорах (CPU)** с различными размерами слов или порядком следо-

вания байтов. Это обеспечивает хорошую производительность за счет переносимости байт-кода:

```
static void intout(int oint)
     {
     int i;
     union {
             int i;
             char c[IntBits/ByteBits];
             } u;
     CodeCheck(IntBits/ByteBits);
     u.i = oint;
     for (i = 0; i < IntBits/ByteBits; i++)
             codep[i] = u.c[i];
     codep += IntBits/ByteBits;
     pc += IntBits/ByteBits;
     }
```

После этого славного примера кода на C вы, вероятно, будете рады вернуться к Unicon и Java. Но C действительно делает низкоуровневые двоичные манипуляции несколько более простыми, чем Unicon или Java. Мораль этой истории такова: *изучайте правильные инструменты для каждого вида работы*.

Заключение

В этой главе вы узнали, как генерировать байт-код для программных интерпретаторов байт-кода. Навыки, которые вы приобрели, включают в себя то, как обходить связанный список промежуточного кода и, для каждого опкода промежуточного кода и псевдоинструкции, как преобразовать их в инструкции набора инструкций байт-кода. Существовали большие различия между семантикой трехадресной машины и машины байт-кода. Многие инструкции промежуточного кода были преобразованы в три или более инструкций машины байт-кода. Обработка инструкций *CALL* была немного причудливой, но для вас важно выполнять вызовы функций способом, требуемым базовой машиной. Изучая все это, вы также узнали, как выводить байт-код в текстовом и двоичном форматах.

В следующей главе представлена альтернатива, более привлекательная для некоторых языков, – генерация собственного кода для основных процессоров.

Вопросы

1. Опишите, как инструкции промежуточного кода, содержащие до трех адресов, преобразуются в последовательность инструкций стековой машины, содержащих не более одного адреса.
2. Если определенная инструкция (допустим, это инструкция 15, со смещением байта 120) помечена пятью различными метками (например, L2, L3, L5, L8 и L13), как эти метки обрабатываются при генерации двоичного байт-кода?

3. В промежуточном коде вызов метода состоит из последовательности инструкций PARM, за которой следует инструкция CALL. Хорошо ли согласуется с промежуточным кодом описанный байт-код для выполнения вызова метода в байт-коде? Что похоже и что отличается?
4. Инструкции *CALL* в **объектно-ориентированных (ОО)** языках, таких как Jzero, всегда предваряются ссылкой на объект (*self* или *this*), для которого вызываются методы… или нет? Объясните ситуацию, в которой инструкция метода *CALL* может не содержать ссылки на объект, и как генератор кода, описанный в этой главе, должен справиться с этой ситуацией.
5. Наш код для помещения в стек методов в первой инструкции *PARM* предполагал, что никакие вложенные последовательности *PARM…CALL* не возникают внутри окружающей последовательности *PARM…CALL*. Можем ли мы гарантировать, что это так для таких примеров, как *f(0, g(1), 2)*?

Глава 13

Генерация собственного кода

В этой главе показано, как получить промежуточный код из главы 9 «*Генерация промежуточного кода*» и генерировать **собственный код**. Термин «собственный» относится к любому набору инструкций, представленному аппаратно на данной машине. В данной главе рассматривается простой генератор собственного кода для x64, доминирующей архитектуры для ноутбуков и настольных компьютеров.

В этой главе разбираются следующие основные темы:

- принятие решения о генерации собственного кода;
- знакомство с набором инструкций x64;
- использование регистров;
- преобразование промежуточного кода в код x64;
- генерация выходных данных x64.

Навыки, развиваемые здесь, включают базовое распределение регистров, выбор инструкций, написание ассемблерных файлов, вызов ассемблера и линкера для создания собственного исполняемого файла. Функциональность, помещенная в эту главу, генерирует код, который изначально выполняется на обычных компьютерах.

Технические требования

Код для этой главы доступен на GitHub: *https://github.com/PacktPublishing/Build-Your-Own-Programming-Language/tree/master/ch13*.

Видеоролик «Код в действии» для этой главы можно найти здесь: *https://bit.ly/2Zdky0I*.

Принятие решения о генерации собственного кода

Генерация собственного кода требует больше усилий, чем генерация байт-кода, но обеспечивает более быстрое выполнение. Собственный код может также использовать меньше памяти или электроэнергии. Он окупается, если конечные пользователи экономят время или деньги, но ориентация на конкретный **центральный процессор (CPU)** приводит к потере переносимости. Возможно, вы захотите сначала реализовать байт-код, а собственный код генерировать, только если язык станет достаточно популярным, чтобы оправдать усилия. Однако есть и другие причины для генерации соб-

ственного кода. Возможно, вы сможете написать свою среду выполнения, используя возможности, предоставляемые другим компилятором. Например, наша среда выполнения Jzero x64 построена с использованием библиотеки языка C **GNU's Not Unix (GNU)**. Теперь давайте рассмотрим некоторые особенности архитектуры x64.

Знакомство с набором инструкций x64

В этом разделе представлен краткий обзор набора инструкций x64, но вам рекомендуется обратиться к руководствам программиста по архитектуре **Advanced Micro Devices (AMD)** или Intel. Книга Дугласа Тейна (Douglas Thain) *«Введение в компиляторы и проектирование языков» (Introduction to Compilers and Language Design)*, доступная на сайте *http://compilerbook.org*, также содержит полезный материал по x64.

x64 – это сложный набор инструкций со множеством функций обратной совместимости. В этой главе рассматривается подмножество x64, которое используется для создания базового генератора кода Jzero. Мы используем **синтаксис ассемблера AT&T**, чтобы наши сгенерированные выходные данные можно было преобразовать в формат двоичного объектного файла ассемблером GNU. Это делается ради мультиплатформенной переносимости.

x64 имеет сотни инструкций с такими именами, как *ADD* для сложения или *MOV* для копирования значения в новое место. Когда инструкция имеет два операнда, максимум один из них может быть ссылкой на основную память. Инструкция x64 может иметь суффикс, указывающий, сколько байтов читаются или записываются, хотя имя **регистра** в инструкции часто делает суффикс излишним. Jzero использует два суффикса инструкций x64 – B для однобайтовых операций над строками и Q для 64-битных операций **с четырьмя словами**. 64-битное слово – это **четверка** слов с точки зрения 16-битного набора инструкций Intel конца 1970-х годов. В этой главе мы используем инструкции, приведенные в табл. 13.1.

Таблица 13.1. Инструкции для примеров в этой главе

Инструкция	Описание
addq	Сложить 64-битное значение с другим 64-битным значением
call	Сохранить адрес возврата в (%rsp), декрементировать %rsp, перейти к функции
cmpq	Сравнить два значения и установить биты кода условия
goto	Перейти к новому месту в коде
jle	Перейти, если меньше или равно
leaq	Вычислить адрес
movq	Переместить 64-битное значение из источника в место назначения
negq	Инвертировать 64-битное значение

Окончание табл. 13.1

Инструкция	Описание
popq	Получить значение из (%rsp) и инкрементировать %rsp
pushq	Сохранить значение в (%rsp) и декрементировать %rsp
ret	Получить значение из (%rsp), инкрементировать %rsp и перейти по адресу
.global	Этот символ должен быть виден из других модулей
.text	Поместить следующие байты в область кода
.type	Этот символ имеет следующий тип

Теперь пришло время определить класс для представления этих инструкций в памяти.

Добавление класса для инструкций x64

Класс *x64* представляет **код операции (опкод)** и операнды, разрешенные в x64. Операнд может быть регистром или ссылкой на значение в памяти. Вы можете увидеть иллюстрацию этого класса в следующем фрагменте кода:

```
class x64(op, opnd1, opnd2)
        method print() .   .   . end
initially(o, o1, o2)
        op := o; opnd1 := o1; opnd2 := o2
end
```

Соответствующий класс Java в файле *x64.java* выглядит следующим образом:

```
public class x64 {
     String op;
     x64loc opnd1, opnd2;
     public x64(String o, Object src, Object dst) {
             op=o; opnd1 = loc(src); opnd2 = loc(dest); }
     public x64(String o, Object opnd) {
             op=o; opnd1 = loc(opnd); }
     public x64(String o) { op=o; }
     public void print() { .   .   . }
}
```

В рамках этого класса *x64* мы переводим адреса из трехадресного кода в адреса x64.

Соответствие областей памяти регистровым режимам адресации x64

Для реализации областей памяти кода, глобальной/статической памяти, стека и кучи в x64 мы должны решить, как обращаться к памяти в каждой области памяти. Инструкции x64 позволяют операндам быть либо регистром, либо адре-

сом памяти. Jzero добавляет смещение к значению в регистре для вычисления адреса, как показано в табл. 13.2.

Таблица 13.2. Режимы доступа к памяти, используемые в этой главе

Режим доступа	Описание
$ k	Непосредственный режим, значение находится в инструкции
k (r)	Косвенный режим, извлечь из памяти k байт в регистр r

В непосредственном режиме значение находится в инструкции. В косвенном режиме основная память определяется относительно регистра x64. Доступ к различным областям памяти осуществляется как смещение относительно различных регистров. Доступ к глобальной и статической памяти осуществляется относительно указателя инструкции, локальная память доступна относительно базового указателя, а доступ к памяти кучи осуществляется относительно регистра указателя кучи. Давайте рассмотрим более широко, как используются регистры.

Использование регистров

Доступ к оперативной памяти медленный. На производительность сильно влияет то, как используются регистры. Оптимальное распределение регистров является **недетерминированной полиномиально-полной (nondeterministic polynomial-complete, NP-complete)** задачей. Эта задача очень сложна. Оптимизирующие компиляторы тратят большие усилия на распределение регистров. Это выходит за рамки данной книги.

В x64 имеется 16 регистров общего назначения, как показано в табл. 13.3, но многие регистры играют особую роль. Арифметика выполняется в регистре аккумулятора, `rax`. Регистры имеют версии от 8- до 64-битных. Jzero использует только 64-битные версии регистров плюс те 8-битные регистры, которые необходимы для строк. В синтаксисе AT&T именам регистров предшествует знак процента, как в `%rax`.

Таблица 13.3. Регистры x64

Регистр	Описание/роль
rip	Указатель инструкции
rax	Аккумулятор. Также возвращаемое значение функции
rbx	Вторичный аккумулятор
rbp	Указатель кадра. Локальные переменные относятся к этому указателю
rsp	Указатель стека. Память между `rbp` и `rsp` является локальной областью
rdi	Указатель назначения. Хранит параметр #1

Окончание табл. 13.3

Регистр	Описание/роль
rsi	Индекс источника. Хранит параметр #2
rdx	Вторичный аккумулятор. Хранит параметр #3
rex	Хранит параметр #4
r8	Хранит параметр #5
r9	Хранит параметр #6
rlO-rl5	Открытые регистры, используемые для любых целей

Многие регистры сохраняются как часть инструкции *call*. Чем больше регистров, тем медленнее выполнение вызовов функций. Эти вопросы определяются **соглашениями о вызове** компилятора. Jzero сохраняет только измененные регистры перед данным вызовом. Прежде чем перейти к собственно генератору кода, давайте немного подробнее рассмотрим, как собственный код использует регистры.

Начинаем с нулевой стратегии

Минимальная регистровая стратегия – это нулевая стратегия, которая преобразует адреса промежуточного кода в адреса x64. Значения загружаются в регистр аккумулятора `rax` для выполнения операций над ними. Результаты немедленно возвращаются в оперативную память.

Базовый указатель `rbp` и указатель стека `rsp` управляют записями активации, которые также называются кадрами. Текущая запись активации связана с регистром базового указателя `rbp`. Текущая локальная область стека находится между базовым указателем и указателем стека. Схема стека x64 показана на рис. 13.1.

В x64 классическая компоновка стека несколько изменена. Шесть регистров *rdi* через *r9* используются для передачи первых шести параметров. Нулевая стратегия сохраняет параметры в памяти, когда начинается вызов функции. Инструкция *div* использует регистр *rdx*, поэтому, помимо того что он используется для передачи параметра #3, *rdx* также необходим для инструкции *div*. Этот недостаток дизайна не влияет на нулевую стратегию.

Назначая регистры для ускорения локальной области, Jzero сопоставляет регистры *rdi-r14* с первыми 88 байтами локальной области. По мере выполнения трехадресных инструкций генератор кода отслеживает для каждого регистра, загружено ли в него значение и было ли оно изменено из соответствующей ячейки оперативной памяти. Генератор кода использует значение в регистре до тех пор, пока этот регистр не будет использован для чего-то другого.

Класс с именем `RegUse` отслеживает в оперативной памяти содержимое соответствующего регистра, если таковое имеется, и было ли его значение изменено с момента последней загрузки в оперативную память. Здесь показана Unicon-реализация `RegUse` в файле *RegUse.icn*:

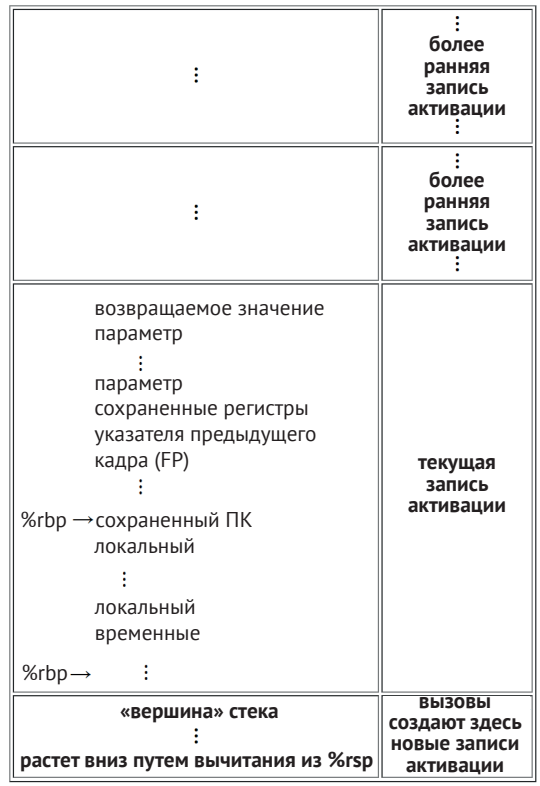


Рис. 13.1. Стек x64, управляемый как последовательность записей активации, растущая вниз

```
class RegUse (reg, offset, loaded, dirty)
      method load()
              if \loaded then fail
              loaded := 1
              return j0.xgen("movq", offset||"(%rbp)", reg)
      end
      method save()
              if /dirty then fail
              dirty := &null
              return j0.xgen("movq", reg, offset||"(%rbp)")
      end
end
```

Поле `reg` обозначает имя строкового регистра, `offset` – смещение байта относительно базового указателя. Булевы флаги `loaded` и `dirty` отслеживают, содержит ли регистр значение и был ли он изменен соответственно. Методы `load()` и `save()` не загружают и не сохраняют, они генерируют инструкции для загрузки и сохранения регистра и устанавливают соответственно флаги `loaded` и `dirty`. Соответствующий код Java выглядит следующим образом:

```
public class RegUse {
```

```
        public String reg;
        int offset;
        public boolean loaded, dirty;
        public RegUse(String s, int i) {
                reg = s; offset=i; loaded=dirty=false; }
        public ArrayList<x64> load() {
                if (loaded) return null;
                loaded = true;
                return j0.xgen("movq", offset+"(%rbp)", reg);
        }
        public ArrayList<x64> save() {
                if (!dirty) return null;
                dirty = false;
                return j0.xgen("movq", reg, offset+"(%rbp)");
        }
}
```

Список экземпляров класса `RegUse` хранится в переменной с именем `regs` класса `j0`, чтобы для каждого первого слова в локальной области использовался соответствующий регистр. Список строится в Unicon следующим образом:

```
off := 0
regs := [: RegUse(«%rdi»|»%rsi»|»%rdx»|»%rcx»|»%r8»|
          "%r9"|"%r10"|"%r11"|"%r12"|"%r13"|"%r14", off-:=8) :]
```

Этот код Unicon просто демонстрирует себя. Оператор | производит все имена регистров для раздельных вызовов `RegUse()`, запускаемых и перехватываемых оператором понимания списка `[: :]`. Хитрость x64 заключается в том, что все смещения являются отрицательными целыми числами, потому что стек растет вниз. В Java эта инициализация выполняется так, как показано здесь:

```
RegUse [] regs = new RegUse[]{ new RegUse(«%rdi», -8),
      new RegUse("%rsi", -16), new RegUse("%rdx", -24),
      new RegUse("%rcx", -32), new RegUse("%r8", -40),
      new RegUse("%r9",-48), new RegUse("%r10", -56),
      new RegUse("%r11", -64), new RegUse("%r12", -72),
      new RegUse("%r13", -80), new RegUse("%r14", -88) };
```

Структура данных работает на границах **базового блока**, сохраняя измененные регистры в памяти и очищая загруженные флаги при появлении метки или инструкции перехода. В начале вызываемой функции флаги параметров `loaded` и `dirty` устанавливаются в *true*, указывая на значения, которые должны быть сохранены в локальной области, прежде чем этот регистр может быть использован повторно. Теперь пришло время рассмотреть, как каждый элемент промежуточного кода преобразуется в код x64.

Преобразование промежуточного кода в код x64

Генератор промежуточного кода из главы 9 «*Генерация промежуточного кода*» поместил промежуточный код для всей программы в атрибут *icode* в корне дерева синтаксиса. Булева переменная `isNative` определяет, генерировать ли байт-код, как показано в предыдущей главе, или собственный код x64. Для генерации x64-кода метод `gencode()` класса *j0* вызывает новый метод в этом классе с именем `x64code()` и передает ему промежуточный код в файле `root.icode` в качестве входных данных. Выходной x64-код помещается в список переменных с именем `xcode` в классе *j0*. Метод Unicon `gencode()`, вызывающий эту функциональность в файле *j0.icn*, выглядит следующим образом:

```
method gencode(root)
      root.genfirst()
      root.genfollow()
      root.gentargets()
      root.gencode()
      xcode := []
      if \isNative then {
              x64code(root.icode)
              x64print()
              }
      else {
              bcode := bytecode(root.icode)
              every (! (\bcode)).print()
      }
end
```

Новый выделенный код представляет собой собственную альтернативу предыдущему поколению байт-кода, который по-прежнему доступен из командной строки. Метод `x64code()` получает список *icode*, а его возвращаемое значение – список объектов класса `x64`. В данном примере полученный x64-код выводится в текстовом виде, мы позволяем ассемблеру сделать работу за нас, чтобы получить двоичный формат. Соответствующий код Java для метода `gencode()` показан здесь:

```
public static ArrayList<x64> xcode;
public static void gencode(root) {
      root.genfirst();
      root.genfollow();
      root.gentargets();
      root.gencode();
      xcode = new ArrayList<x64>();
      if (isNative && xcode != null) {
              x64code(root.icode);
              x64print();
      } else {
              ArrayList<byc> bcode = bytecode(root.icode);
              if (bcode != null)
```

```
                        for (int i = 0; i < bcode.size(); i++)
                                bcode.get(i).print();
        }
}
```

Теперь рассмотрим, как адреса промежуточного кода становятся ссылками на память в x64.

Соответствие адресов промежуточного кода местоположению в x64

Адреса в промежуточном коде представляют собой абстрактные пары (регион, смещение), представленные в классе `address` из главы 9 «*Генерация промежуточного кода*». Соответствующий класс `x64loc` представляет местоположения в x64, которые включают информацию о режиме адресации или регистр для использования. Реализация Unicon в файле *x64loc.icn* выглядит следующим образом:

```
class x64loc(reg, offset, mode)
initially(x,y,z)
      if \z then { reg := x; offset := y; mode := z }
      else if \y then {
      if x === "imm" then { offset := y; mode := 5 }
      else if x === "lab" then { offset := y; mode := 6 }
      else {
      reg := x; offset := y
      if integer(y) then mode := 3 else mode := 4
      }
      }
      else {
      if integer(x) then { offset := x; mode := 2 }
      else if string(x) then { reg := x; mode := 1 }
      else stop("bad x64loc ", image(x))
      }
end
```

Поле `reg` – это имя регистра строки. Поле `offset` – это либо целочисленное смещение, либо имя строки, от которой вычисляется смещение. `mode` – 1 для регистра, 2 – для абсолютного адреса и 3 – для регистра и целочисленного смещения. Режим 4 – для регистра и имени смещения строки, 5 – для непосредственного значения и 6 – для метки. Java-реализация в файле *x64loc.java* выглядит следующим образом:

```
public class x64loc {
      public String reg; Object offset;
      public int mode;
      public x64loc(String r) { reg = r; mode = 1; }
      public x64loc(int i) { offset=(Object)Integer(i); mode=2; }
      public x64loc(String r, int off) {
              if (r.equals("imm")) {
```

```
                    offset=(Object)Integer(off); mode = 5; }
            else if (r.equals("lab")) {
                    offset=(Object)Integer(off); mode = 6; }
            else { reg = r; offset = (Object)Integer(off);
                    mode = 3; }
    }
    public x64loc(String r, String s) {
            reg = r; offset = (Object)s; mode=4; }
}
```

В коде Java есть конструкторы для различных типов памяти. Область и смещение класса `address` должны быть преобразованы в экземпляр класса `x64loc`, который является операндом в классе `x64`. Это делается с помощью метода `loc()` в классе *j0*, который получает адрес в качестве параметра и возвращает экземпляр `x64loc`. Код Unicon для метода `loc()` в файле *j0.icn* выглядит следующим образом:

```
method loc(a)
     if /a then return
     case a.region of {
     "loc": { if a.offset <= 88 then return loadreg(a)
               else return x64loc("rbp", -a.offset) }
     "glob": { return x64loc("rip", a.offset) }
     "const": { return x64loc("imm", a.offset) }
     "lab": { return x64loc("lab", a.offset) }
     "obj": { return x64loc("r15", a.offset) }
     "imm": { return x64loc("imm", a.offset) }
     }
end
```

Когда код преобразует адрес в экземпляр `x64loc`, смещения локальной области преобразуются в отрицательные значения, поскольку стек растет вниз. Методы Java в файле *j0.java* показаны здесь:

```
public static x64loc loc(String s) { return new x64loc(s);}
public static x64loc loc(Object o) {
     if (o instanceof String) return loc((String)o);
     if (o instanceof address) return loc((address)o);
     return null;
}
public static x64loc loc(address a) {
     switch (a.region) {
     case "loc": { if (a.offset <= 88) return loadreg(a);
                       else return x64loc("rbp", -a.offset); }
     case "glob": { return x64loc("rip", a.offset); }
     case "const": { return x64loc("imm", a.offset); }
     case "lab": { return x64loc("lab", a.offset); }
     case "obj": { return x64loc("r15", a.offset); }
     case "imm": { return x64loc("imm", a.offset); }
     default: { semErr("x64loc unknown region"); return null; }
     }
}
```

Вспомогательный метод `loadreg()` используется для локальных смещений в первых 88 байтах. Если значение еще не присутствует в назначенном регистре, для его размещения используется инструкция `movq`, как показано в следующем фрагменте кода:

```
method loadreg(a)
        r := a.offset/8 + 1
        if / (regs[r].loaded) then {
                every put(xcode,
                          !xgen(«movq»,(-
                          a.offset)||»(%rbp)»,regs[r].reg))
                regs[r].loaded := «true»
                }
        return x64loc(regs[a.offset/8+1].reg)
end
```

Здесь показана реализация `loadreg()` в Java:

```
public static x64loc loadreg(address a) {
     long r = a.offset/8;
       if (!regs[r].loaded) {
            xcode.addAll(xgen("movq",
                    String.valueOf(-a.offset)+"(%rbp)", regs[r].reg));
            regs[r].loaded = true;
            }
            return x64loc(regs[a.offset/8+1].reg);
}
```

При наличии класса `x64` необходима еще одна вспомогательная функция, для того чтобы сформулировать метод генератора кода `x64code()`. Нам нужен удобный производственный метод генерации инструкций x64 и присоединения их к списку `xcode`. Этот метод `xgen()` преобразует операнды источника и получателя в экземпляры `x64loc`, которые могут добавлять инструкции `movq` для загрузки значений в регистры. Код Unicon выглядит следующим образом:

```
method xgen(o, src, dst)
        return [x64(o, loc(src), loc(dst))]
end
```

Существует много версий соответствующей реализации Java, показанной здесь, для обработки случаев, когда источником или получателем являются адреса или строковые имена регистров:

```
public static ArrayList<x64> l64(x64 x) {
        return new ArrayList<x64>(Arrays.asList(x)); }
public static ArrayList<x64> xgen(String o){
        return l64(new x64(o)); }
public static ArrayList<x64> xgen(String o,
     address src, address dst) {
     return l64(new x64(o, loc(src), loc(dst))); }
```

```
public static ArrayList<x64> xgen(String o, address opnd) {
        return l64(new x64(o, loc(opnd))); }
public static ArrayList<x64> xgen(String o, address src,
                                  String dst) {
        return l64(new x64(o, loc(src), loc(dst))); }
public static ArrayList<x64> xgen(String o, String src,
                                  address dst) {
        return l64(new x64(o,loc(src),loc(dst))); }
public static ArrayList<x64> xgen(String o, String src,
                                  String dst) {
        return l64(new x64(o,loc(src),loc(dst))); }
public static ArrayList<x64> xgen(String o, String opnd) {
        return l64(new x64(o, loc(opnd))); }
```

В приведенном выше фрагменте кода метод `l64()` просто создает один элемент `ArrayList`, содержащий объект `x64`. Остальные – это просто множество реализаций `xgen()`, которые получают различные типы параметров. Теперь, наконец, пришло время представить метод генератора кода x64.

Реализация метода генератора кода x64

Unicon-реализация метода `x64code()` в классе *j0* выглядит следующим образом. Реализация должна заполнить один случай для каждого опкода в трехадресном наборе инструкций. Случаев будет много, поэтому мы представим каждый из них отдельно, а первый показан здесь:

```
method x64code(icode)
     every i := 1 to *\icode do {
             instr := icode[i]
             case instr.op of {
                             "ADD": { ... append translation of ADD to xcode }
                             "SUB": { ... append translation of SUB to xcode }
                             .   .   .
                     }
             }
end
```

Java-реализация `x64code()` показана ниже:

```
public static void x64code(ArrayList<tac> icode) {
     int parmCount = -1;
             for(int i=0; i<icode.size(); i++) {
                     tac instr = icode.get(i);
                     switch(instr.op) {
                     case "ADD": { ... append translation of ADD to xcode}
                     case "SUB": { ... append translation of SUB to xcode}
                             ...
                             }
                     }
     }
```

В рамках этого метода x64code() мы предоставим переводы для каждой из трехадресных инструкций. Начнем с простых выражений.

Генерация кода x64 для простых выражений

Случаи для трехадресного опкода, в частности для кода сложения, имеют много общих элементов. В Unicon код x64 для сложения создается следующим образом:

```
"ADD": { xcode |||:= xgen("movq", instr.op2, "%rax") |||
                     xgen("addq", instr.op3, "%rax") |||
                     xgen("movq", "%rax", instr.op1) }
```

В этом коде операнд 2 и операнд 3 считываются из памяти и помещаются в стек. Сама инструкция ADD работает полностью из стека. Результат затем извлекается из стека и помещается в операнд 3. В Java реализация сложения содержит следующий код:

```
case "ADD": { xcode.addAll(xgen("movq", instr.op2, "%rax"));
              xcode.addAll(xgen("addq", instr.op3, "%rax"));
                 xcode.addAll(xgen("movq", "%rax", instr.op1));
               break;
```

Существует 19 или около того трехадресных инструкций. Схема генерации окончательного кода, проиллюстрированная выше для инструкции ADD, используется и для других арифметических инструкций. Для унарного оператора, такого как NEG, схема немного упрощается, как мы видим здесь:

```
"NEG": { xcode |||:= xgen(«movq», instr.op2, "%rax") |||
                xgen("negq", "%rax") |||
                xgen("movq", "%rax", instr.op1) }
```

В Java реализация отрицания состоит из следующего кода:

```
case "NEG": { xcode.addAll(xgen(«movq», instr.op2, "%rax"));
              xcode.addAll(xgen("negq", "%rax"));
              xcode.addAll(xgen("movq", "%rax", instr.op1));
              break; }
```

Еще более простая инструкция, такая как ASN, возможно, заслуживает специального случая, поскольку код x64 имеет инструкции прямого перемещения из памяти в память, но один из вариантов – придерживаться того же сценария и упростить предыдущий шаблон еще больше, например так:

```
«ASN»: { xcode |||:= xgen(«movq», instr.op2, «%rax») |||
                     xgen("movq", "%rax", instr.op1) }
```

В Java реализация присваивания может выглядеть следующим образом:

```
case "ASN": { xcode.addAll(xgen("movq", instr.op2, "%rax"));
              xcode.addAll(xgen("movq", "%rax", instr.op1));
              break; }
```

Выражения являются наиболее распространенными элементами в коде. Следующая категория – указатели.

Генерация кода для обработки указателей

Три из трехадресных инструкций относятся к использованию указателей: ADDR, LCON и SCON. Инструкция ADDR превращает адрес в памяти в часть данных, которыми можно манипулировать для выполнения таких операций, как арифметика указателей. Она перемещает свой операнд, ссылку на адрес, в одну из областей памяти, как если бы это было значение непосредственного режима. Код проиллюстрирован в следующем фрагменте:

```
"ADDR": { xcode |||:= xgen("leaq", instr.op2, "%rax")
          xcode |||:= xgen("%rax", instr.op1) }
```

В Java реализация инструкции ADDR состоит из следующего кода:

```
case "ADDR": { xcode.addAll(xgen("leaq", instr.op2, "%rax"));
               xcode.addAll(xgen("%rax", instr.op1));
               break; }
```

Инструкция LCON читает из памяти, на которую указывает другая память, таким образом:

```
"LCON": { xcode |||:= xgen("movq", instr.op2, "%rax") |||
                      xgen("movq", "(%rax)", "%rax") |||
                      xgen("movq", "%rax", instr.op1) }
```

В Java реализация инструкции LCON состоит из следующего кода:

```
case "LCON": { xcode.addAll(xgen("movq", instr.op2, "%rax"));
               xcode.addAll(xgen("movq", "(%rax)", "%rax"));
               xcode.addAll(xgen("movq", "%rax", instr.op1));
               break; }
```

Инструкция SCON записывает в память, на которую указывает другая память, вот так:

```
"SCON": { xcode |||:= xgen("movq", instr.op2, "%rbx") |||
                      xgen("movq", instr.op1, "%rax")
                      xgen("movq", "%rbx", "(%rax)") }
```

В Java реализация инструкции SCON состоит из следующего кода:

```
case "SCON": { xcode.addAll(xgen("movq", instr.op2, "%rbx"));
               xcode.addAll(xgen("movq", instr.op1, "%rax"));
               xcode.addAll(xgen("movq", "%rbx", "(%rax)"));
               break; }
```

Эти инструкции важны для поддержки структурированных типов данных, таких как массивы. Теперь рассмотрим генерацию байт-кода для потока управления, начиная с инструкции GOTO.

Генерация собственного кода для безусловных и условных переходов

Семь инструкций промежуточного кода относятся к инструкциям перехода. Самая простая из них – это безусловный переход, или инструкция GOTO. Инструкция GOTO присваивает новое значение регистру указателя инструкции. Неудивительно, что байт-код GOTO является реализацией трехадресной инструкции GOTO, как показано в следующем фрагменте кода:

```
"GOTO": { xcode |||:= xgen("goto", instr.op1) }
```

В Java реализация инструкции GOTO состоит из следующего кода:

```
case "GOTO": { xcode.addAll(xgen("goto", instr.op1));
               break; }
```

Инструкции условного перехода трехадресного кода преобразуются в более простые инструкции окончательного кода. Для набора инструкций x64 это означает выполнение инструкции сравнения, которая устанавливает коды условий перед одной из инструкций условного перехода x64. Unicon-реализация инструкции BLT выглядит так:

```
"BLT": { xcode |||:= xgen("movq", instr.op2, "%rax") |||
                     xgen("cmpq", instr.op3, "%rax") |||
                     xgen("jle", instr.op1) }
```

В Java реализация генерации байт-кода для инструкции BLT состоит из следующего кода:

```
case "BLT": { xcode.addAll(xgen("movq", instr.op2, "%rax"));
              xcode.addAll(xgen("cmpq", instr.op3, "%rax"));
              xcode.addAll(xgen("jle", instr.op1));
              break; }
```

Эта схема используется для нескольких трехадресных инструкций. Теперь рассмотрим более сложные формы передачи потока управления, связанные с вызовами и возвратами методов.

Генерация кода для вызовов методов и возвратов

Три инструкции промежуточного кода обрабатывают очень важную тематику вызовов и возвратов функций и методов. Последовательность из нуля или более инструкций PARM помещает значения в стек, после чего инструкция CALL выполняет вызов метода. Инструкция RET внутри вызываемого метода возвращает из метода в место вызова.

Это соглашение о вызове трехадресного кода должно быть реализовано с помощью набора инструкций x64, предпочтительно со стандартными соглашениями вызова для этой архитектуры, что требует передачи первых шести параметров в определенные регистры.

Чтобы передать параметры в корректные регистры, инструкция PARM должна отслеживать номер параметра. Код Unicon для инструкции PARM следующий:

```
"PARM": { if /parmCount then {
            parmCount := 1
            every j := i+1 to *icode do
              if icode[j].op == "CALL" then break
              parmCount +:= 1
            }
      else parmCount -:= 1
      genParm(parmCount, instr.op1) }
```

Для первого параметра цикл every подсчитывает количество параметров перед инструкцией CALL. Метод genParm() вызывается с номером текущего параметра и операндом. В Java реализация инструкции PARM аналогична, как можно увидеть здесь:

```
case "PARM": { if (parmCount == -1) {
                for(int j = i+1; j<icode.size(); j++) {
                     tac callinstr = icode.get(j);
                     if (callinstr.op.equals("CALL"))
                             break;
                     parmCount++;
                     }
                }
              else parmCount--;
              genParm(parmCount, instr.op1);
              break; }
```

Приведенные выше случаи для параметров зависят от метода genParm(), который генерирует код в зависимости от номера параметра. Перед загрузкой регистров для нового вызова функции значения регистров, которые были изменены, должны быть сохранены в своих ячейках оперативной памяти следующим образом:

```
method genParm(n, addr)
```

```
      every (!regs).save()
      if n > 6 then xcode |||:= xgen("pushq", addr)
      else xcode |||:= xgen("movq", addr, case n of {
              1: "%rdi"; 2: "%rsi"; 3: "%rdx";
              4: "%rcx"; 5: "%r8"; 6: "%r9"
      })
end
```

Соответствующая Java-реализация `genParm()` выглядит так:

```
public static void genParm(int n, address addr) {
      for (RegUse x : regs) x.save();
      if (n > 6) xcode.addAll(xgen("pushq", addr));
      else {
              String s = "error:" + String.valueOf(n);
              switch (n) {
              case 1: s = "%rdi"; break; case 2: s = "%rsi"; break;
              case 3: s ="%rdx"; break; case 4: s = "%rcx"; break;
              case 5: s = "%r8"; break; case 6: s = "%r9"; break;
              }
              xcode.addAll(xgen("movq", addr, s));
      }
}
```

Далее следует инструкция `CALL`. После вызова значение `op1` в трехадресном коде сохраняется из регистра `rax`. Поле источника `op2` – это адрес метода, который использовался до первой инструкции `PARM`. Поле источника `op3` задает количество параметров, которые не используются в x64. Код показан в следующем фрагменте:

```
"CALL": { xcode |||:= xgen("call", instr.op3)
        xcode |||:= xgen("movq", "%rax", instr.op1)
        parmCount := -1 }
```

В Java реализация инструкции `CALL` выглядит следующим образом:

```
case "CALL": { xcode.addAll(xgen("call", instr.op3));
              xcode.addAll(xgen("movq", "%rax", instr.op1));
              parmCount = -1;
              break; }
```

Реализация инструкции `RETURN` в Unicon выглядит так:

```
"RETURN": { xcode |||:= xgen("movq", instr.op1, «%rax») |||
                xgen("leave") ||| xgen("ret", instr.op1) }
```

В Java реализация инструкции `RETURN` выглядит следующим образом:

```
case "RETURN":{ xcode.addAll(xgen("movq", instr.op1, "%rax"));
                xcode.addAll(xgen("leave"));
                xcode.addAll(xgen("ret", instr.op1));
break; }
```

Генерация кода для вызовов методов и возвратов не представляет особой сложности. Теперь рассмотрим, как обрабатывать псевдоинструкции трехадресного кода.

Обработка меток и псевдоинструкций

Псевдоинструкции, такие как метки, не преобразуются в код, но они присутствуют в связанном списке трехадресных инструкций и требуют учета в окончательном коде. Наиболее распространенной и очевидной псевдоинструкцией является **метка**. Если окончательный код генерируется в человекочитаемом ассемблерном формате, метки могут быть сгенерированы как есть, в зависимости от любых различий в формате, необходимых для того, чтобы сделать их легальными в ассемблерном файле. Если мы генерируем окончательный код в двоичном формате, на данном этапе метки требуют точного расчета и полностью заменяются фактическими смещениями байтов в сгенерированном машинном коде. Код проиллюстрирован здесь:

```
"LAB": { every (!regs).save()
         xcode |||:= xgen("lab", instr.op1) }
```

Эквивалентная реализация в Java показана ниже:

```
case "LAB": { for (RegUse ru : regs) ru.save();
              xcode.addAll(xgen("lab", instr.op1)); break; }
```

В качестве примера других типов псевдоинструкций рассмотрим, какой код x64 выводить для начала и конца методов. В начале метода в промежуточном коде все, что у вас есть, – это псевдоинструкция `proc x,n1,n2`. Код Unicon для этой псевдоинструкции показан здесь:

```
"proc": {
     n := (instr.op1.offset + instr.op2.offset) * 8
     xcode |||:= xgen(".text") |||
                 xgen(".globl", instr.op1) |||
                 xgen(".type", instr.op1, "@function") |||
                 xgen(instr.op1||":") |||
                 xgen("pushq", "%rbp") |||
                 xgen("movq", "%rsp", "%rbp") |||
                 xgen("subq", "$"||n, "%rsp")
     every i := !(instr.op2.offset) do
             regs[i].loaded := regs[i].dirty := "true"
     every j := i+1 to 11 do
             regs[i].loaded := regs[i].dirty := "false"
}
```

В приведенном выше коде директива `.text` указывает ассемблеру записывать строку за строкой в раздел кода. Директива `.globl` указывает, что на имя метода должна быть установлена ссылка из других модулей. Директива `.type` указывает, что символ является функцией. Директива под директивой `.type` объявляет (искаженное) имя функции ассемблерной меткой, то есть это имя может быть использовано как ссылка на данную функцию в области кода. Инструкция `pushq` сохраняет предыдущий базовый указатель стека. Инструкция `movq` устанавливает базовый указатель для новой функции на текущей вершине стека.

Присвоение `n` вычисляет общее количество байтов локальной области, включая пространство для параметров, переданных в регистры, но скопированных в стековую память, если метод вызывает другой метод. Инструкция `subq` выделяет память, перемещая указатель стека вниз на соответствующую величину. Два цикла помечают используемые параметры, отмечая при этом, что остальные регистры чисты. В Java соответствующий код для заголовка метода выглядит так:

```
case "proc": {
      xcode.addAll(xgen(".text"));
      xcode.addAll(xgen(".globl", instr.op1));
      xcode.addAll(xgen(".type", instr.op1, "@function"));
      xcode.addAll(xgen(instr.op1 + ":"));
      xcode.addAll(xgen("pushq", "%rbp"));
      xcode.addAll(xgen("movq", "%rsp", "%rbp"));
      int n = (instr.op1.offset + instr.op2.offset) * 8;
      xcode.addAll(xgen("subq", "$"+n, "%rsp"));
      int j;
      for (j = 0; j < instr.op2.offset; j++)
              regs[j].loaded = regs[j].dirty = true;
      for (; j < 11; j++)
              regs[j].loaded = regs[j].dirty = false;
      break;
}
```

Псевдоинструкция `end` несколько проще, как мы видим ниже. Мы не хотим свалиться в конец метода, поэтому формируем инструкции для восстановления старого указателя кадра и возврата наряду с директивами ассемблера для конца функции:

```
"end": {
        xcode |||:= xgen("leave") ||| xgen("ret")
}
```

Соответствующая Java-реализация псевдоинструкции `end` показана ниже:

```
case "end": {
      xcode.addAll(xgen("leave"));
      xcode.addAll(xgen("ret"));
      break;
}
```

В предыдущих нескольких разделах была получена структура данных, содержащая представление байт-кода, а затем было показано, как преобразуются различные трехадресные инструкции. Теперь перейдем к созданию собственного выходного кода x64 из списка объектов x64.

Генерация выходных данных x64

Как и во многих традиционных компиляторах, собственный код для Jzero будет получен путем выполнения следующих шагов. Во-первых, мы запишем связанный список объектов x64 на человекочитаемом языке ассемблера с расширением *.s*. Затем мы вызовем GNU-ассемблер для преобразования этого списка в двоичный объектный файл с расширением *.o*. Исполняемый файл создается путем вызова линкера, который объединяет набор *.o* файлов, указанных пользователем, с набором *.o* файлов, содержащим код библиотеки среды выполнения, и данные, на которые ссылается сгенерированный код. В этом разделе представлен каждый из указанных этапов, начиная с создания ассемблерного кода.

Запись кода x64 в формате ассемблера

В этом разделе приводится краткое описание формата ассемблера x64, поддерживаемого ассемблером GNU, который использует синтаксис AT&T. Инструкции и псевдоинструкции располагаются в строку с отступом слева на табуляцию (или восемь пробелов). Метки являются исключением из этого правила, поскольку они не содержат отступов и состоят из идентификатора, за которым следует двоеточие. Псевдоинструкции начинаются с точки. После мнемоники инструкции или псевдоинструкции может стоять табуляция или пробелы, за которыми следуют ноль, один или два операнда, разделенных запятыми, в зависимости от требований инструкции.

В качестве примера всего этого рассмотрим простой файл ассемблера x64, содержащий функцию, которая ничего не делает и возвращает значение 42. В ассемблере это может выглядеть следующим образом:

```
      .text
      .globl   two
      .type    two,   @function
two:
.LFB0:
      pushq    %rbp
      movq     %rsp,  %rbp
      movl     $42,  -4(%rbp)
      movl     -4(%rbp),   %eax
      popq     %rbp
      ret
.LFE0:
        .size    two,  .-two
```

В классе *j0* есть метод `x64print()`, который выводит список объектов *x64* в текстовый файл в этом формате. Как видно из его кода, показанного ниже, он вызывает метод `print()` для каждого из объектов *x64* в списке `xcode`:

```
method x64print()
        every (!xcode).print()
end
```

Здесь показана Java-реализация `x64print()` в файле *j0.java*:

```
public static void x64print() {
        for(x64 x : xcode) x.print();
}
```

Показав, как пишется ассемблерный код, пришло время рассмотреть, как вызвать ассемблер GNU для создания объектного файла.

Переход от ассемблера к объектному файлу

Объектные файлы – это двоичные файлы, содержащие реальный машинный код. Ассемблерный файл, записанный в предыдущем разделе, собирается с помощью команды `as`, как показано здесь:

```
as --gstabs+ -o two.o two.s
```

В этой командной строке `--gstabs+` является рекомендуемой опцией, которая включает отладочную информацию. `-o two.o` – опция, задающая имя выходного файла.

Полученный двоичный файл `two.o` не может быть понят человеком непосредственно, но его можно просмотреть с помощью различных инструментов. Просто для развлечения первые 102 байта единиц и нулей из файла `two.o` показаны на рис. 13.2. Каждая строка содержит шесть байт, справа показана интерпретация в **американском стандартном коде для обмена информацией (American Standard Code for Information Interchange – ASCII)**. На скриншоте показаны единицы и нули в текстовом виде благодаря инструменту под названием *xxd*, который выводит биты в текстовом виде. Конечно, компьютер обычно обрабатывает от 8 до 64 бит за раз без предварительной транслитерации в текстовую форму.

```
01111111 01000101 01001100 01000110 00000010 00000001  .ELF..
00000001 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 00000001 00000000  ......
00111110 00000000 00000001 00000000 00000000 00000000  >.....
00000000 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 00010000 00000010  ......
00000000 00000000 00000000 00000000 00000000 00000000  ......
00000000 00000000 00000000 00000000 01000000 00000000  ....@.
00000000 00000000 00000000 00000000 01000000 00000000  ....@.
00001011 00000000 00001010 00000000 01010101 01001000  ....UH
10001001 11100101 11000111 01000101 11111100 00000100  ...E..
00000000 00000000 00000000 10001011 01000101 11111100  ....E.
01011101 11000011 00000000 01000111 01000011 01000011  ]..GCC
00111010 00100000 00101000 01010101 01100010 01110101  : (Ubu
01101110 01110100 01110101 00100000 00110111 00101110  ntu 7.
00110101 00101110 00110000 00101101 00110011 01110101  5.0-3u
```

Рис. 13.2. Двоичные представления неудобны для человека, но компьютеры предпочитают их

Не случайно байты 2–4 файла содержат надпись ELF. **Executable and Linkable Format (ELF)** – один из наиболее популярных многоплатформенных форматов объектных файлов, и первые четыре байта определяют формат файла. Достаточно сказать, что такие форматы двоичных файлов важны для машин, но сложны для человека. Теперь рассмотрим, как объектные файлы объединяются для формирования исполняемых программ.

Линковка, загрузка и включение среды выполнения

Задача объединения набора двоичных файлов для создания исполняемого файла называется **линковкой**. Это еще один раздел, о котором можно написать целую книгу. Для Jzero очень хорошо то, что мы можем позволить программе линкера GNU `ld` сделать эту работу. Она принимает опцию `-o` файла, чтобы определить имя выходного файла, а затем – любое количество объектных файлов с расширением `.o`. Объектные файлы для рабочего исполняемого файла включают в себя файл запуска, который инициализирует и вызывает *main()*, часто называемый `crt1.o`, за которым следуют файлы приложения, а затем ноль или более файлов библиотек среды выполнения. Если мы создаем библиотеку среды выполнения Jzero с именем *libjzero.o*, командная строка `ld` может выглядеть следующим образом:

```
ld -o hello /usr/lib64/crt1.o hello.o -ljzero
```

Если ваша библиотека среды выполнения вызывает функции из настоящей библиотеки C, вам придется их также включить. Полная ссылка на основе `ld` для среды выполнения, построенной поверх **GNU Compiler Collection (GCC)**, *glibc* выглядит следующим образом:

```
ld -dynamic-linker /lib64/ld-linux-x86-64.so.2 \
     /usr/lib/x86_64-linux-gnu/crt1.o \
     /usr/lib/x86_64-linux-gnu/crti.o \
     /usr/lib/gcc/x86_64-linux-gnu/7/crtbegin.o \
     hello.o -ljzero \
     -lc /usr/lib/gcc/x86_64-linux-gnu/7/crtend.o \
     /usr/lib/x86_64-linux-gnu/crtn.o
```

Вашим пользователям не придется часто набирать эту командную строку, поскольку она будет заложена в код вызова линкера вашего компилятора. Но у нее есть фатальный недостаток – она не переносима и зависит от версии. Чтобы использовать существующую библиотеку C GCC из вашей среды выполнения, вы можете позволить существующей установке GCC выполнить линковку за вас, например так:

```
gcc -o hello hello.o
```

Линкер должен собрать один большой двоичный код из нескольких входных двоичных объектных кодов. В дополнение к объединению всех инструкций в объектных файлах основная работа линкера заключается в определении адресов всех функций и глобальных переменных в исполняемом файле. Линкер также должен обеспечить для каждого объектного файла механизм, позволяющий находить адреса функций и переменных из других объектных файлов.

Для функций и переменных, которые не определены в пользовательском коде, а являются частью среды выполнения языка, линкер должен иметь механизм для поиска среды выполнения и включения из нее того, что необходимо. Среда выполнения включает в себя стартовый код, который инициализирует среду выполнения и подготавливает все для вызова *main()*. Она может включать один или несколько объектных файлов, которые всегда подключаются к любому исполняемому файлу для данного языка. Самое главное – линкер предоставляет способ поиска и линковки только тех частей среды выполнения, которые явно вызываются из пользовательского кода.

В современных системах со временем все усложняется. Стандартно принято откладывать различные аспекты линковки и загрузки до времени выполнения, в частности чтобы позволить процессам использовать библиотечный код, который уже загружен для использования другими процессами.

Заключение

В этой главе вы узнали, как генерировать собственный код для процессоров x64. Среди навыков, которые вы освоили, основным был обход связанного списка промежуточного кода и его преобразование в инструкции набора инструкций x64. Кроме того, вы научились записывать код x64 в формате ассемблера GNU. Наконец, вы узнали, как вызвать ассемблер и линкер для преобразования собственного кода в формат объектных и исполняемых файлов ELF. В следующей главе мы рассмотрим более подробно задачу реализации новых высокоуровневых операторов и встроенных функций в среде выполнения вашего языка.

Вопросы

1. Какие основные новые концепции становятся необходимыми для генерации собственного кода x64 по сравнению с байт-кодом?
2. Каковы преимущества и недостатки предоставления адресов глобальных переменных в виде смещения относительно регистра указателя инструкции *%rip*?
3. Одной из важных проблем, влияющих на производительность современных компьютеров, является скорость выполнения вызовов функций и возвратов. Почему скорость вызова функций важна? При каких обстоятельствах архитектура x64 способна выполнять быстрые вызовы функций и возвраты? Есть ли в архитектуре x64 аспекты, которые могут замедлить вызов функций?

Глава 14

Реализация операторов и встроенных функций

Новые языки программирования изобретаются потому, что время от времени для решения проблем в новых областях применения необходимы новые идеи и новые вычислительные возможности. Библиотеки функций или классов являются основным средством расширения языков программирования за счет использования дополнительных вычислительных возможностей, но если бы добавления библиотеки было всегда достаточно, вам бы не пришлось создавать свой собственный язык, не так ли?

В этой и следующей главах обсуждаются расширения языка, выходящие за рамки библиотек. В этой главе будет разобрано, как поддерживать очень высокоуровневые и специфические для конкретной области возможности языка путем добавления встроенных в язык операторов и функций. В следующей главе будет обсуждаться добавление управляющих структур.

Добавление операторов и встроенных функций может сократить и уменьшить то, что должны написать программисты, чтобы решить определенные проблемы в вашем языке, улучшить его производительность или сделать возможной семантику языка, которая в противном случае была бы сложной. Эта глава иллюстрирует идеи в контексте Jzero, уделяя особое внимание типам *String* и *array*. Для сравнения в последующих разделах описывается, как операторы и функции реализованы в Unicon.

В этой главе будут рассмотрены следующие основные темы:

- реализация операторов;
- написание встроенных функций;
- интеграция встроенных функций со структурами управления;
- разработка операторов и функций для Unicon.

Благодаря этой главе вы узнаете, как писать части среды выполнения, которые слишком сложны для того, чтобы быть инструкциями в наборе инструкций. Вы также узнаете, как добавить в ваш язык специфичные для конкретной области возможности. Начнем с того, как реализовать операторы высокого уровня!

Реализация операторов

Операторы – это выражения, которые вычисляют значение. Простые операторы, которые вычисляют свои результаты с помощью нескольких инструк-

ций базовой машины, были рассмотрены в предыдущих главах. В этом разделе описано, как реализовать оператор, выполняющий много шагов. Вы можете называть эти операторы **составными операторами**. В этом случае основной сгенерированный код может выполнять вызовы функций в базовой машине.

Функции, вызываемые из сгенерированного кода, пишутся на языке реализации, а не на исходном языке. Они могут быть более низкого уровня и делать вещи, которые невозможны на исходном языке. Например, правила передачи параметров могут отличаться в языке реализации от правил в языке программирования, который вы создаете.

Если вы задаетесь вопросом, когда следует превратить новое вычисление в оператор, то можете обратиться к главе 2 «Дизайн языка программирования». Вместо того чтобы повторять этот материал, мы просто отметим, что операторы, как правило, ограничены работой максимум с тремя операндами и что большинство операторов используют два или один операнд.

Если вы можете использовать аналогии с арифметикой, чтобы повторно использовать соответствующие знакомые операторы для ваших новых вычислений, отлично. В противном случае вы ожидаете, что программисты должны выучить и запомнить новые шаблоны, что требует от них очень многого. Вы можете добавить сотни операторов в свой язык, но человеческий мозг не сможет запомнить такое количество. Если вы попытаетесь ввести больше операторов, чем, например, клавиш на клавиатуре, ваш язык может быть отвергнут из-за чрезмерной когнитивной нагрузки. Теперь давайте рассмотрим, в какой степени добавление новых операторов в язык приводит или является следствием добавления новых аппаратных возможностей.

Подразумевают ли операторы аппаратную поддержку, и наоборот

Точно так же, как вы можете обнаружить, что ваше обычное вычисление может заслуживать того, чтобы быть оператором в вашем языке, разработчики аппаратного обеспечения могут понять, что компьютеры должны поддерживать общие вычисления с помощью собственных инструкций. Когда разработчики языка понимают, что вычисление должно быть оператором в их языке, это делает вычисления кандидатом на аппаратную реализацию. Аналогично, когда разработчики аппаратных средств реализуют общие вычисления в своей аппаратуре, разработчики языка должны задаться вопросом, должны ли эти вычисления поддерживаться непосредственно операторами или другим синтаксисом. Вот пример.

До появления в 1994 году платформы 80486 большинство ПК не поставлялись с аппаратным обеспечением для работы с плавающей запятой. На некоторых платформах сопроцессор с плавающей запятой был дорогим дополнением, необходимым только для научных вычислений. Если вы реализовывали компилятор, вы реализовывали тип данных с плавающей запятой в программном обеспечении в виде набора функций. Эти системные функции среды выполнения вызывались из сгенерированного кода, но были прозрачны для программиста. Программа, в которой объявлены две переменные с плавающей запятой, *f1* и *f2*, и выполняется выражение *f1* + *f2*, вычислит сумму с плавающей

запятой, не замечая, что сгенерированный код включает вызовы функций, которые могут быть в 10 или 100 раз медленнее, чем сложение двух целых чисел.

Вот еще один пример, который может оказаться для вас болезненным. После того как программа под названием Doom создала огромный спрос на 3D-графику в 1990-х годах, были разработаны графические процессоры (GPU). Они поддерживают вычисления, выходящие далеко за рамки их первоначальной сферы применения – игр и других 3D-программ. Однако они не поддерживаются непосредственно в большинстве основных языков, а крутая кривая обучения и сложное программирование для GPU уменьшили их огромное влияние. Подведем итог: существует богатая сочная серая зона между операторами, которые должны быть встроены в язык программирования для упрощения программирования, а также операторами, которые должны быть встроены в аппаратное обеспечение. Теперь давайте узнаем, как добавлять составные операторы, на примере добавления в язык Jzero одного из них – конкатенации.

Добавление конкатенации строк в генерацию промежуточного кода

Для Jzero тип String (строка) является важным, но он не был реализован в предыдущих главах о генерации кода или интерпретации байт-кода, которые были сосредоточены на вычислениях целых чисел. Класс String имеет оператор конкатенации, который мы должны реализовать. Некоторые компьютеры поддерживают конкатенацию аппаратно для некоторых представлений строк. Для Jzero String является классом, а конкатенация сравнима с методом – либо фабричным методом, либо конструктором, поскольку она возвращает новую строку, а не изменяет свои аргументы.

В любом случае, пришло время реализовать *s1+s2*, где *s1* и *s2* – строки. Для промежуточного кода мы можем добавить новую инструкцию *SADD*. Если вы не хотите этого делать, то можете сгенерировать код, вызывающий метод для конкатенации строк, но мы будем работать с инструкцией промежуточного кода. Правило генерации кода для оператора plus будет генерировать различный код в зависимости от типа. Прежде чем реализовать это, мы должны модифицировать метод `check_types()` в классе *tree* так, чтобы инструкция «строка *s1* плюс строка *s2*» была корректной и вычисляла строку. В реализации Unicon измените строки в файле *tree.icn*, где сложение проверяется на тип, чтобы разрешить тип `String`, следующим образом:

```
if op1.str() === op2.str() === («int»|»double»|"String")
        then return op1
```

В реализации Java добавьте следующее *OR* в файле *tree.java*:

```
if (op1.str().equals(op2.str()) &&
    (op1.str().equals(«int») ||
     op1.str().equals(«double») ||
     op1.str().equals("String")))
  return op1;
```

Модифицировав средство проверки типов, чтобы разрешить конкатенацию строк, аналогично расширим метод генерации промежуточного кода `genAddExpr()`. Модификации Unicon в файле *tree.icn* выделены в теле метода, показанном здесь:

```
method genAddExpr()
      addr := genlocal()
      icode := kids[1].icode ||| kids[2].icode
      if typ.str() == "String" then {
              if rule ~= 1320 then
                      j0.semErr("subtraction on strings is not defined")
              icode |||:= gen("SADD", addr,
                      kids[1].addr, kids[2].addr)
              }
      else icode |||:= gen(if rule=1320 then "ADD" else "SUB",
              addr, kids[1].addr, kids[2].addr)
end
```

Проверка по правилу производства 1320 происходит потому, что тип `String` не поддерживает вычитание. Соответствующие Java-модификации в файле *tree.java* выглядят следующим образом:

```
void genAddExpr() {
      addr = genlocal();
      icode = new ArrayList<tac>();
      icode.addAll(kids[0].icode); icode.addAll(kids[1].icode);
      if (typ.str().equals("String")) {
              if (rule != 1320)
                      j0.semErr("subtraction on strings is not defined");
              icode.addAll(gen("SADD", addr,
                                  kids[0].addr,kids[1].addr);
              }
      else icode.addAll(gen(((rule==1320)?"ADD":"SUB"),
                              addr, kids[0].addr, kids[1].addr));
end
```

На данном этапе мы добавили инструкцию промежуточного кода для конкатенации строк. Теперь пришло время реализовать ее в среде выполнения. Сначала рассмотрим интерпретатор байт-кода.

Добавление конкатенации строк в интерпретатор байт-кода

Поскольку интерпретатор байт-кода является программным, мы можем просто добавить еще одну инструкцию байт-кода для конкатенации строк, как мы это сделали для промежуточного кода. Опкод #22 для инструкции `SADD` должен быть добавлен в файлы *Op.icn* и *Op.java*. Мы должны модифицировать генератор байт-кода, чтобы он генерировал инструкцию байт-кода `SADD` для инструкции промежуточного кода `SADD`. Реализация Unicon в методе `bytecode()` файла *j0.icn* выглядит следующим образом:

```
«SADD»: {
      bcode |||:= j0.bgen(Op.PUSH, instr.op2) |||
                  j0.bgen(Op.PUSH, instr.op3) |||
                  j0.bgen(Op.SADD) |||
                  j0.bgen(Op.POP, instr.op1)
      }
```

Если это похоже на код для инструкции ADD, то в этом и есть смысл. Как и в случае с инструкцией ADD, окончательный код состоит в основном из преобразования трехадресной инструкции в последовательность одноадресных инструкций. Java-реализация в файле *j0.java* показана здесь:

```
case «SADD»: {
      rv.addAll(j0.bgen(Op.PUSH, instr.op2));
      rv.addAll(j0.bgen(Op.PUSH, instr.op3));
      rv.addAll(j0.bgen(Op.SADD, null));
      rv.addAll(j0.bgen(Op.POP, instr.op1));
      break;
      }
```

Мы также должны реализовать эту инструкцию байт-кода, что означает, что мы должны добавить ее в интерпретатор байт-кода. Поскольку языки реализации Unicon и Java оба имеют высокоуровневые строковые типы с семантикой, подобной Jzero, мы можем надеяться, что реализация будет простой. Если Jzero-представление String в интерпретаторе байт-кода *j0x* является строкой базового языка реализации, то реализация инструкции SADD будет просто выполнять конкатенацию строк. Однако в большинстве языков семантика исходного языка отличается от семантики языка реализации, поэтому обычно необходимо реализовать представление типа исходного языка, которое моделирует семантику исходного языка в базовом языке реализации.

Выдав это предупреждение, давайте посмотрим, сможем ли мы реализовать строки Jzero как обычные строки Unicon и Java. В этом случае инструкция SADD в методе interp() файла j0machine.icn почти не отличается от целочисленной инструкции ADD:

```
Op.SADD: {
      val1 := pop(stack); val2 := pop(stack)
      push(stack, val1 || val2)
}
```

Эта реализация Unicon опирается на тот факт, что для стека значений Unicon безразлично, если вы иногда помещаете в него целые числа, а иногда – строки. В Unicon есть базовая область строк, где находится память для содержимого строк, и интерпретатор байт-кода использует ее неявно.

Соответствующая Java-реализация в файле *j0machine.java* сложнее. Переменная stackbuf, которую мы реализовали, была *байт-буфером (ByteBuffer)*, который был рассчитан на хранение большого числа 64-битных целочисленных значений, но теперь мы должны решить, как использовать ее для хранения

строк. Если мы храним фактическое содержимое строки в `stackbuf`, мы больше не реализуем стек – мы реализуем кучу, и это будет целое дело. Вместо этого мы будем хранить в `stackbuf` некоторый целочисленный код, который можем использовать для получения строки путем поиска ее в **строковом пуле**:

```
case Op.SADD: {
      String val1 = stringpool.get(stackbuf.getLong(sp--));
      String val1 = stringpool.get(stackbuf.getLong(sp--));
      long val3 = stringpool.put(val1 + val2);
      stackbuf.putLong(sp++, val3);
}
```

Этот код зависит от класса `stringpool`, который использует уникальные целые числа для хранения и извлечения строк. Эти уникальные числа являются ссылками на строковые данные, которые удобно хранить в `stackbuf`, но теперь для Java-реализации требуется класс `stringpool`, поэтому он находится здесь, в файле *stringpool.java*. Для любой строки способ получить ее уникальное целое число – это найти его в пуле. После того как оно будет получено таким образом, уникальное целое число может быть использовано для получения строки позже по запросу:

```
public class stringpool {
      static HashMap<String,Long> si;
      static HashMap<Long,String> is;
      static long serial;
      static { si = new HashMap<>(); is = new HashMap<>(); }
      public static long put(String s) { … }
      public static String get(long L) { … }
}
```

Для этого класса требуется следующая пара методов. Метод `put()` вставляет строки в пул. Если строка уже находится в пуле, возвращается ее существующий целочисленный ключ. Если строка еще не находится в пуле, порядковый номер инкрементируется, и этот номер ассоциируется со строкой:

```
public static long put(String s) {
      if (si.containsKey(s)) return si.get(s);
      serial++;
      si.put(s, serial);
      is.put(serial, s);
      return serial;
}
```

Метод `get()` извлекает `String` из `stringpool`:

```
public static String get(long L) {
        return is.get(L);
}
```

Теперь пришло время рассмотреть, как реализовать этот оператор для собственного кода.

Добавление конкатенации строк в собственную среду выполнения

Собственный код Jzero находится на гораздо более низком уровне, чем интерпретатор байт-кода. Реализация семантики класса `String` языка Jzero с нуля на языке C – это большая работа. Jzero использует чрезвычайно упрощенное подмножество класса `String` Java, для которого у нас есть место, чтобы описать только основные моменты. Вот базовое представление класса `String` на языке C для использования в Jzero:

```
struct String {
     struct Class *cls;
     long len;
     char *buf;
};
```

В этой структуре `cls` является указателем на еще не определенную структуру для информации о классе, `len` – длина строки, а `buf` – указатель на данные. Конкатенация строк Jzero может быть определена следующим образом:

```
struct String *j0concat(struct String *s1,
                        struct String *s2){
     struct string *s3 = alloc(sizeof struct String);
     s3->buf = allocstring(s1->len + s2->len);
     strncpy(s3->buf, s1->buf, s1->len);
     strncpy(s3->buf + s1->len, s2->buf, s2->len);
     return s3;
}
```

Этот код вызывает столько же вопросов, сколько дает ответов, например в чем разница между `alloc()` и `allocstring()`, мы вернемся к ним в ближайшее время. Но это функция, которую мы можем вызвать из сгенерированного собственного кода с помощью следующего дополнения к файлу *j0.icn*:

```
«SADD»: {
     bcode |||:= xgen("movq", instr.op2, "%rdi") |||
                 xgen("movq", instr.op3, "%rsi") |||
                 xgen("call", "j0concat") |||
                 xgen("movq", "%rax", instr.op1)
     }
```

Соответствующая Java-реализация в файле *j0.java* показана ниже:

```
case «SADD»: {
     rv.addAll(xgen("movq", instr.op2, "%rdi"));
     rv.addAll(xgen("movq", instr.op3, "%rsi"));
```

```
rv.addAll(xgen("call", "j0concat"));
rv.addAll(xgen("movq", "%rax", instr.op1));
break;
}
```

Здесь видно, что замена вызова функции для реализации инструкции непосредственного кода является простой. Давайте сравним это с кодом, который генерируется для встроенных функций, который мы представим далее.

Написание встроенных функций

Языки низкого уровня, такие как C, не имеют встроенных функций, они имеют стандартные библиотеки, которые содержат функции, доступные для всех программ. Подключение функции к вашей программе и вызов функции – это концептуально одно и то же действие, независимо от того, библиотечная это функция или пользовательская. Чем выше уровень языка, тем более заметна разница между тем, что написано для среды выполнения на языке реализации более низкого уровня, и тем, что написано конечными пользователями на самом языке. В этом разделе слова «*функция*» и «*метод*» используются как взаимозаменяемые. Рассмотрим, как реализовать встроенные функции в интерпретаторе байт-кода.

Добавление встроенных функций в интерпретатор байт-кода

Давайте реализуем *System.out.println()* в интерпретаторе байт-кода. Один из вариантов разработки – реализовать новую инструкцию машины байт-кода для каждой встроенной функции, включая *println()*. Это не очень хорошо подходит для тысяч встроенных функций. Мы могли бы реализовать инструкцию *callnative*, обеспечивающую способ определить, какую встроенную функцию мы хотим вызвать. Некоторые языки реализуют сложный интерфейс для вызова функций собственного кода и реализуют *println()* (или какую-либо функцию более низкого уровня для создания блока) как функцию-обертку, написанную на Jzero, которая использует собственный интерфейс вызова.

Для Jzero, как описано в разделе «*Запуск программы Jzero*» главы 11 «*Интерпретаторы байт-кода*», мы решили использовать для обозначения встроенных функций существующую инструкцию `call` со специальными значениями функций. В качестве специальных значений мы выбрали малые отрицательные целые числа, в которые обычно помещается адрес точки входа функции. Таким образом, механизм вызова функции должен быть построен так, чтобы искать малые отрицательные целые числа для различения типов методов и совершать корректные действия для определяемых пользователем и встроенных методов.

Рассмотрим метод `do_println()`, который мы предложили в главе 11«*Интерпретаторы байт-кода*». В Jzero этот метод среды выполнения предназначен для записи в стандартное устройство вывода, подобно *puts()* в C. Записываемая строка находится в стеке. Она больше не находится на вершине стека, куда инструкция вызова переместила адрес возврата функции. В Unicon метод `do_println()` может быть реализован следующим образом:

```
method do_println()
        write(stack[2])
end
```

В Java метод `do_println()` выглядел бы следующим образом:

```
public static do_println() {
      String s = stringpool.get(…);
      System.out.println(s);
}
```

Встроенные функции в байт-коде просты. Теперь давайте рассмотрим написание встроенных функций для собственного кода.

Написание встроенных функций для использования в реализации собственного кода

Пришло время реализовать `System.out.println()` для реализации собственного кода Jzero. В компиляторе Java это будет метод объекта `System.out`, но для Jzero мы можем делать все, что нам удобно. Мы можем написать собственную функцию с именем `System_out_println()` на ассемблере или, если наш сгенерированный собственный код тщательно придерживается соглашений о вызовах компилятора C на той же платформе, можем написать ее на C, поместить в нашу библиотеку среды выполнения Jzero и связать со сгенерированными ассемблерными модулями, чтобы сформировать исполняемый файл. Функция принимает один строковый аргумент, `struct String *`, как показано в предыдущем разделе. Вот реализация, которую вы можете поместить в файл *System_out_println.c*:

```
#include <stdio.h>
void System_out_println(struct String *s) {
      for(int i = 0; i < s->len; i++) putchar(s->buf[i]);
      putchar('\n');
}
```

Более интересной частью всего этого является то, как сгенерированный код получает доступ к этой и другим встроенным собственным функциям. Вы можете скомпилировать его с помощью следующей командной строки для `gcc`:

```
gcc -c System_out_println.c
```

Вы можете добавить выходной файл `System_out_println.o` в архивную библиотеку с именем `libjzero.a` с помощью следующей командной строки:

```
ar cr libjzero.a System_out_println.o
```

Две приведенные выше командные строки не выполняются в вашем компиляторе при каждой компиляции или линковке. Вместо этого они выполняются во время создания самого компилятора Jzero, наряду с потенциально многими

другими битами кода оператора или библиотеки встроенных функций. Они создают архивный файл библиотеки с именем *libjzero.a*. Этот архивный файл может быть подключен к сгенерированному Jzero коду с помощью команд `ld` или `gcc`, как описано в разделе «*Линковка, загрузка и включение среды выполнения*» главы 13 «*Генерация собственного кода*».

Опция командной строки *-lsomefile* расширяется до соответствия *libsomefile.a*, так что наша среда выполнения вызывается как *-ljzero*. Как же компилятор Jzero, который предположительно может быть установлен где угодно, находит библиотеку среды выполнения, которая предположительно может быть установлена где угодно? Ответ зависит от операционной системы, и некоторые из удобных опций требуют административных привилегий. Если вы можете скопировать *libjzero.a* в тот же каталог, который используется вашим линкером для других системных библиотек, например *C:\Mingw\lib* в Windows или */usr/lib64* в Linux, вы можете обнаружить, что все работает отлично. Если такой возможности нет, вы можете прибегнуть к переменным окружения или опциям командной строки, чтобы сообщить линкеру, где находится библиотека, либо сообщить самому компилятору Jzero в командной строке Jzero, где можно найти библиотеку среды выполнения. Добавление встроенных функций, подобных этой, важно потому, что не каждое дополнение языка может быть сделано в виде оператора. Аналогично не каждое дополнение языка лучше всего сформулировать в виде функции. Иногда такие операторы и встроенные функции более эффективны, когда они являются частью новых структур управления, которые поддерживают некоторую новую проблемную область. Давайте рассмотрим, как эти операторы и функции могут выиграть от интеграции с синтаксическими дополнениями в виде структур управления.

Интеграция встроенных функций со структурами управления

Структуры управления обычно являются более крупными элементами, чем выражения, например циклы. Они часто ассоциируются с новой семантикой языка программирования или новыми областями, в которых могут выполняться специализированные вычисления. Структуры управления обеспечивают контекст, в котором выполняется оператор (часто это составной оператор, состоящий из целого блока кода). Это может быть то, выполняется ли он (или сколько раз выполняется), в связи с какими данными он должен быть применен, или даже то, с какой семантикой должны интерпретироваться операторы. Иногда эти структуры управления явно и исключительно используются для ваших новых операторов или встроенных функций, но часто эти взаимодействия являются неявными побочными продуктами решения проблем, которые позволяет решать ваш язык.

Выполняется ли данный блок кода, какой из нескольких блоков выполнить или повторное выполнение кода – это наиболее традиционные структуры управления, такие как операторы *if* и циклы. Наиболее вероятные возможности взаимодействия операторов или функций с этими конструкциями включают **специальный синтаксис итератора**, который управляет циклами с исполь-

зованием ваших доменных значений, и **специальный синтаксис переключателя**, который позволяет выбрать блок(и) кода для выполнения.

Оператор *WITH* в языке Pascal – это старый, но хороший пример ассоциации некоторых данных с фрагментом кода, который использует эти данные. Его синтаксис *WITH r DO оператор*. Оператор *WITH* присоединяет некоторую запись *r* к оператору (обычно это составной оператор), в котором поля записи находятся в области видимости, а поле с именем *x* не обязательно должно иметь префикс в виде выражения доступа, такого как *r.x*. Это низкоуровневый строительный блок, который (и связанные с ним ссылки *self* или *this*) используется в объектном ориентировании. Но Pascal позволяет такие объектные вложения для отдельных операторов, более мелких, чем вызовы методов, а также позволяет ассоциировать несколько объектов с одним и тем же блоком кода.

Мы можем проиллюстрировать некоторые моменты взаимодействия со структурами управления на примере рассмотрения реализации цикла *for*, который выполняет итерацию по строкам. Поскольку Java несовершенен, вы не можете написать синтаксис, то есть *for(char c:s) statement*, чтобы выполнить *statement* один раз для каждого элемента *s*, но можете написать *for(char c:s.toCharArray()) statement*.

Итак, массивы Java хорошо взаимодействуют со структурой управления *for*, но класс Java *String* не так хорош. Существует интерфейс *Iterable*, однако строки не работают с ним без прыжков сквозь дополнительные обручи. Когда вы разрабатываете язык, постарайтесь сделать общие задачи простыми. Аналогичное замечание относится и к доступу к элементам *String*. Никто не хочет писать *s.charAt(i)*, когда можно было бы написать *s[i]*. Это хороший аргумент в пользу поддержки операторов. Пример интеграции встроенной функции со структурой управления путем предоставления параметров по умолчанию будет приведен в следующем разделе. Теперь давайте рассмотрим, как реализованы операторы и встроенные функции в Unicon.

Разработка операторов и функций для Unicon

Unicon – это очень высокоуровневый язык со множеством встроенных функций. Для таких языков имеет смысл провести некоторую инженерную работу, чтобы упростить создание среды выполнения. Цель этого раздела – рассказать немного о том, как это было сделано для Unicon, с целью сравнения. Операторы и встроенные функции Unicon реализованы с использованием языка **RTL**, что расшифровывается как **язык выполнения (Run Time Language)**. RTL – это надмножество языка C, разработанное Кеном Уолкером (Ken Walker) для облегчения сборки мусора и определения типов в среде выполнения. RTL записывает код на языке C, так что это почти специализированная форма препроцессора C, который обслуживает базу данных для поддержки определения типов.

Операторы и функции в RTL выглядят как код на языке C, со многими элементами специального синтаксиса. Существует поддержка синтаксиса для ассоциирования с различными частями кода C в зависимости от типа данных операндов. Чтобы обеспечить возможность определения типа, объявляется тип результата Unicon, который производится каждым фрагментом C-кода. Язык RTL также имеет поддержку синтаксиса, что позволяет легко указать,

когда должно происходить преобразование типа операнда. Кроме того, каждый фрагмент C-кода помечается синтаксисом, чтобы указать, следует ли его встроить в сгенерированный код или выполнить указанный код через вызов функции C. Сначала мы опишем, как писать операторы в RTL, а также их особенности. После этого мы узнаем, как писать функции Unicon в RTL, которые похожи на операторы, но немного более общие по своей природе.

Написание операторов в Unicon

После различных хитроумных расширений макросов и пропуска *#ifdef* оператор сложения в Unicon выглядит следующим образом. В следующем коде показаны три различные формы сложения – для целых чисел C (длинных), целых чисел произвольной точности и чисел с плавающей запятой. В фактической реализации существует четвертая форма сложения для параллельного добавления данных по массиву за раз:

```
        operator{1} + add(x, y)
                declare { C_integer irslt; }
                arith_case (x, y) of {
                        C_integer: { abstract { return integer }
                                inline { … }
                                }
                        integer: { abstract { return integer }
                                inline { … }
                                }
                        C_double: { abstract { return real }
                                inline { … }
                                }
                        }
end
```

В приведенном выше коде специальный RTL-оператор случая для арифметических операторов, называемый `arith_case`, выполняется во время компиляции оптимизирующим компилятором Unicon, в то время как в интерпретаторе байт-кода это фактический оператор `switch`, который выполняется во время выполнения. В `arith_case` скрыт набор стандартных для всего языка правил автоматического преобразования типов. Например, строки преобразуются в соответствующие им числа, если это возможно.

Случай для обычного сложения целых чисел в C проверяет валидность своего результата и запускает сложение произвольной точности, как в среднем случае целочисленного переполнения. Тело этого случая выглядит следующим образом: некоторые `#ifdef` были опущены для удобства чтения. Резюмируя, хотя синтаксис RTL встраивает этот код, код одного оператора плюс для целочисленного типа включает один, возможно, два вызова функций:

```
irslt = add(x,y, &over_flow);
if (over_flow) {
      MakeInt(x,&lx);
      MakeInt(y,&ly);
```

```
      if (bigadd(&lx, &ly, &result) == RunError)
              runerr(0);
      return result;
      }
else return C_integer irslt;
```

Функция `add()` вызывается для выполнения обычного целочисленного сложения. Если нет переполнения, целочисленный результат, возвращаемый функцией `add()`, является валидным и возвращается. По умолчанию RTL возвращает из функций операторов Unicon общее значение Unicon, которое может содержать любой тип. Если вместо этого возвращается примитивный тип языка C, его необходимо указать. В представленном выше коде `return` в конце помечен в RTL, чтобы указать, что возвращается целое число на языке C.

Если при вызове функции `add()` происходит переполнение, вызывается функция `bigadd()` для выполнения сложения с произвольной точностью. Вот реализация функции `add()` в среде выполнения Unicon, которая выполняет целочисленное сложение и проверяет переполнение. Здесь используются только ссылки на макросы для значений *2^63-1* и *-2^63*, а не расширенный синтаксис RTL. Вероятно, кто-то был достаточно внимателен, когда писал этот код:

```
word add(word a, word b, int *over_flowp)
{
      if ((a ^ b) >= 0 &&
          (a >= 0 ? b > MaxLong - a : b < MinLong - a)) {
         *over_flowp = 1;
         return 0;
         }
      else {
        *over_flowp = 0;
        return a + b;
        }
}
```

Это довольно простой код на C, за исключением исключающего или (`a^b`), которое является способом спросить, положительны оба значения или оба отрицательны. В дополнение к вычислению суммы эта функция записывает булево значение по адресу, указанному в ее третьем параметре, чтобы сообщить о том, произошло ли целочисленное переполнение.

Поскольку ему не нужно проверять переполнение, случай `arith_case` сложения вещественных чисел с плавающей запятой, обозначенный `C_double` в RTL, намного проще. Вместо вызова вспомогательной функции сложение вещественных чисел выполняется с использованием обычного оператора C +:

```
return C_double (x + y);
```

Мы опустили соответствующую реализацию функции сложения произвольной точности `bigadd()`, которая вызывается в этом операторе и занимает много страниц. Если вы хотите добавить в свой язык арифметику произвольной точности, вам следует прочитать о библиотеке **GNU Multiple Precision (GMP)**,

которая находится по адресу *https://gmplib.org/*. Теперь давайте рассмотрим несколько вопросов, которые возникают при написании встроенных функций для Unicon.

Разработка встроенных функций Unicon

Встроенные функции Unicon также написаны на RTL (и C), и, как и в случае с операторами, код каждой функции может быть предназначен как для встраивания, так и для вызова в качестве функции. Встроенные функции в среднем длиннее, чем операторы, но, возможно, в большинстве случаев синтаксис функций RTL существует как усовершенствованная форма обертки, которая позволяет вызывать функции языка C из Unicon с преобразованиями между представлениями типов значений Unicon и C по мере необходимости. В отличие от операторов, многие функции имеют несколько параметров, значения которых могут быть заданы по умолчанию с помощью специального синтаксиса. В качестве примера рассмотрим код функции анализа строк Unicon `any()`, которая истинна, если символ в текущей позиции строки является членом набора символов, указанного в ее первом параметре. Зарезервированное слово RTL `function` объявляет встроенную функцию Unicon вместо обычной функции языка C. Синтаксис `{0,1}` указывает, сколько результатов может выдать данная функция. Она может выдать как ноль результатов (отказ), так и один, она не является генератором. Оператор `if-then` указывает, что первый параметр способен быть преобразованным в `cset` (если это не так, произойдет ошибка выполнения), а зарезервированное слово `body` указывает, что сгенерированный код должен вызывать функцию, а не вставлять код:

```
function{0,1} any(c,s,i,j)
      str_anal( s, i, j )
      if !cnv:tmp_cset(c) then
              runerr(104,c)
      body {
              if (cnv_i == cnv_j)
                      fail;
              if (!Testb(StrLoc(s)[cnv_i-1], c))
                      fail;
              return C_integer cnv_i+1;
              }
end
```

Помимо битов синтаксиса RTL, огромную роль играют макросы. `str_anal` – это макрос, который устанавливает строку для анализа, задавая параметры 2–4 по умолчанию в соответствии с текущей средой сканирования строки. `str_anal` также гарантирует, что `s` – строка, а `i` и `j` – целые числа, при необходимости преобразуя их в эти типы и выдавая ошибку выполнения, если получено значение несовместимого типа. Среды сканирования строк создаются с помощью структуры управления сканированием строк, исследуемое местоположение в строке может быть перемещено другими функциями сканирования строк. Добавление специфичных для домена структур управления, таких как сканирование строк, будет представлено в следующей главе. Данный пример служит

для их мотивирования. Одна из причин использования новых управляющих структур заключается в том, чтобы сделать операторы и встроенные функции более мощными и лаконичными.

В этом разделе мы представили несколько основных моментов, показывающих, как реализованы операторы и встроенные функции языка Unicon. Многие проблемы в среде выполнения языка очень высокого уровня, как оказалось, вращаются вокруг большого семантического различия между исходным языком (Unicon) и языком реализации (в случае Unicon – C). В зависимости от уровня языка, который вы создаете, и языка, на котором вы пишете его реализацию, вы можете найти полезным прибегнуть к подобным техникам.

Заключение

В этой главе вы узнали, как писать высокоуровневые операторы и встроенные функции для среды выполнения вашего языка. Один из основных моментов, который вы должны усвоить, заключается в том, что реализация операторов и функций может варьироваться от совершенно разной до почти полностью одинаковой, в зависимости от языка, который вы разрабатываете.

Примеры в этой главе научили вас, как писать код в вашей среде выполнения, который будет вызываться из сгенерированного кода. Вы также узнали, как решить, когда следует сделать что-либо функцией среды выполнения, вместо того чтобы генерировать для этого код с помощью инструкций.

В следующей главе мы продолжим тему реализации встроенных функций, изучив структуры управления доменом.

Вопросы

1. Математически доказуемо, что каждое вычисление, которое можно реализовать в виде оператора или встроенной функции, можно реализовать в виде метода библиотеки. Тогда зачем реализовывать высокоуровневые операторы и встроенные функции?
2. Какие факторы вы должны учитывать, принимая решение между созданием нового оператора или новой встроенной функции?
3. Вероятно, были веские причины, по которым в Java решено предоставить строкам лишь частичную поддержку операторов и управляющих структур, несмотря на то что строки важны и поддерживаются лучше в таких языках, как Icon и Unicon (и Python, на который повлиял Icon). Можете ли вы назвать некоторые причины этого?

Глава 15

Структуры управления доменами

Генерация кода, представленная в предыдущих главах, охватывает основные условные и циклические структуры управления, но доменно-ориентированные языки часто имеют уникальную или специфическую семантику, которая требует введения новых структур управления. Добавление новой структуры управления обычно значительно сложнее, чем добавление новой функции или оператора. Однако когда они эффективны, добавление структур управления доменами является большой частью того, что делает доменно-ориентированные языки достойными разработки вместо простого написания библиотек классов.

В этой главе рассматриваются следующие основные темы:

- понимание необходимости новой структуры управления;
- обработка текста с помощью сканирования строк;
- рендеринг графических областей.

Первый раздел поможет вам узнать, как определить, когда необходима структура управления доменами. Во втором и третьем разделах будут представлены два примера структур управления доменом.

Эта глава даст вам лучшее представление о том, когда и как внедрять новые структуры управления, если это необходимо при разработке вашего языка. Что еще более важно, вы узнаете, как пройти по тонкой грани, которая разделяет необходимость придерживаться генерации знакомого кода для знакомых структур и необходимость снизить усилия программистов в новых областях применения за счет внедрения новой семантики.

Язык **Java** и его **подмножество Jzero** не имеют сопоставимых структур управления доменами, поэтому примеры в этой главе взяты из Unicon и его предшественника Icon. Хотя данная глава описывает их реализацию, иногда используя примеры кода, вы читаете эту главу скорее ради идей, чем для того, чтобы набрать код и увидеть его выполнение. Во-первых, давайте еще раз рассмотрим, когда новая структура управления является оправданной.

Понимание необходимости новой структуры управления

Новая структура управления нужна, когда она решает одну или несколько основных проблем программирования (**болевых точек**). Часто болевые точ-

ки возникают, когда люди начинают писать программное обеспечение для поддержки нового класса компьютерного оборудования или для новой области (домена) применения. Во время разработки языка осведомленность или знания о болевых точках домена применения могут существовать или не существовать, но чаще всего осознание болевых точек возникает в результате раннего существенного опыта попыток написания программного обеспечения для этого домена.

Болевые точки нередко возникают из-за сложности, частых и губительных ошибок, дублирования кода, других известных неприятных огрехов программирования, или **антипаттернов**. Некоторые характерные недостатки кода описаны в книге *Мартина Фаулера (Martin Fowler) «Рефакторинг. Улучшение существующего кода»*. Антипаттерны описаны на сайте *antipatterns.com* и в нескольких книгах, ссылки на которые приведены на этом сайте.

Отдельные программисты или программные проекты могут сгладить недостатки кода или избежать антипаттернов путем проведения **рефакторинга** кода, но когда использование прикладных библиотек домена влечет за собой то, что большинство или все приложения в этом домене сталкиваются с такими проблемами, становится целесообразным создание одной или нескольких структур управления доменом. Теперь давайте определим, что такое структуры управления, чтобы понять, о чем идет речь.

Определение структуры управления

Если вы наберёте в Google определение **структуры управления**, там будет написано что-то вроде «*структуры управления определяют порядок выполнения одного или нескольких фрагментов кода*». Это определение подходит для традиционных основных языков. Оно фокусируется на потоке управления, и в нем рассматриваются два вида структур управления – выбор того, что (или нужно ли) выполнять, и циклы, которые могут повторяться при некоторых условиях. Операторы *if* и циклы *while*, которые мы реализовали для Jzero в этой книге ранее, являются хорошими примерами этого.

Языки более высокого уровня, как правило, имеют более тонкий взгляд на структуры управления. Например, в языке со встроенным **перебором с возвратом** порядок выполнения фрагментов кода становится более сложным. В данной книге будет перефразировано определение Ральфа Грисвольда (Ralph Griswold) структуры управления в языке программирования Icon: структура управления – это выражение, содержащее два или более подвыражений, в котором одно подвыражение используется для управления выполнением другого подвыражения. Это определение является более общим и более мощным, чем традиционное определение, приведенное в предыдущем абзаце.

Выражение «управление выполнением» в определении Грисволда может быть истолковано так широко и так свободно, как вы хотите. Вместо того чтобы просто определить, выполняется ли фрагмент кода, или какой фрагмент кода, или сколько раз, структура управления может определить, как выполняется код. Это может означать введение новых областей, в которых имена интерпретируются по-другому, или добавление новых операторов. Далее в этой главе мы увидим интересные примеры структур управления, влияющих на выполнение кода. Давайте начнем с простого.

Устранение избыточных параметров

Многие библиотеки общего назначения имеют API с одними и теми же параметрами, повторяющимися в десятках или даже сотнях связанных функций. Приложения, использующие эти API, могут содержать множество вызовов, в которых библиотеке снова и снова передается одна и та же последовательность параметров. Хорошим примером этого является классический графический API Microsoft Windows. Такие вещи, как окна, контекст устройств, цвета, стили линий и шаблоны кистей предоставляются многократно для многих вызовов рисования. Вы можете написать любой код, но когда вы вызываете *GetDC()* для получения контекста устройства, лучше, чтобы был ровно один соответствующий вызов *ReleaseDC()*. Большая часть кода между этими двумя точками будет передавать этот контекст устройства в качестве параметра снова и снова.

Для уменьшения сетевого трафика аналог Win32 с открытым исходным кодом, известный как Xlib, библиотека C для написания приложений под X Window System, поместил несколько общих графических элементов рисования в объект **графического контекста**, который уменьшает избыточность параметров. Несмотря на это, API Xlib остается сложным и содержит значительную избыточность параметров.

Разработчики библиотек в некоторых случаях являются гениями, но API могут быть относительно враждебными для обычных разработчиков, имеющими крутые кривые обучения и множество ошибок. До появления построителей графических пользовательских интерфейсов, которые генерировали этот код для нас, создание графических интерфейсов непропорционально замедляло разработку и увеличивало стоимость многих приложений, и это соблазняло многих программистов на такую плохую практику, как копирование блоков и изменение огромных участков кода пользовательского интерфейса.

Для языка, где новая структура управления не является альтернативой, проблема избыточных параметров может быть неизбежной. Если вы создаете язык, структура управления является реальным вариантом решения этой проблемы.

Болевые точки становятся целью для новой структуры управления, когда они могут быть устранены в рамках домена, который вы поддерживаете, и побочным продуктом традиционных языков и отсутствия поддержки этого домена. Если в вашем прикладном домене существуют библиотеки и приложения, написанные на традиционном языке, вы можете изучить этот код, чтобы найти его болевые точки и создать структуры управления, которые устранят их в вашем языке программирования. Если ваш прикладной домен совсем новый и нет API и базы приложений на традиционном языке, для поиска болевых точек вы можете прибегнуть к догадкам или написанию примеров программ на вашем новом языке. Давайте рассмотрим новую структуру управления, где эти принципы были успешно применены, – сканирование строк в языках Icon и Unicon.

Сканирование строк в Icon и Unicon

Unicon унаследовал эту структуру управления доменом от своего непосредственного предшественника Icon. Сканирование строк вызывается синтаксисом `s ? expr`. `s` является *субъектом* сканирования внутри `expr`, и на него

ссылается глобальное ключевое слово *&subject*. Текущая *позиция* анализа в сканируемой строке, которая хранится в ключевом слове *&pos*, обозначает индекс места в строке, где в данный момент производится анализ. Текущая позиция начинается в начале строки и может быть перемещена назад и вперед, обычно двигаясь к концу строки. Например, в следующей программе строка `s` содержит `"For example, suppose string s contains"`:

```
procedure main()
      s := "For example, suppose string s contains"
      s ? {
              tab(find("suppose"))
              write("after tab()")
      }
end
```

Теперь, допустим, нам нужно добавить структуру управления сканированием:

```
s ? { … }
```

Здесь ключевые слова `&subject` и `&pos` будут иметь значения, показанные на рис. 15.1.

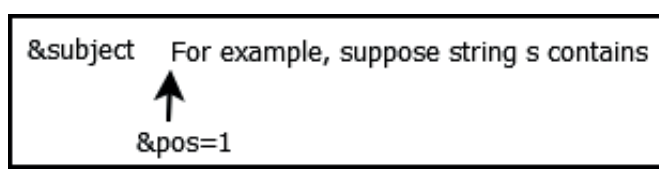


Рис. 15.1. Субъект и позиция в начале сканирования

После сканирования вперед, минуя запятую и пробел после нее, позиция сканирования строки будет установлена на 14. Последующий анализ начнется со слова `suppose`, как показано на рис. 15.2.

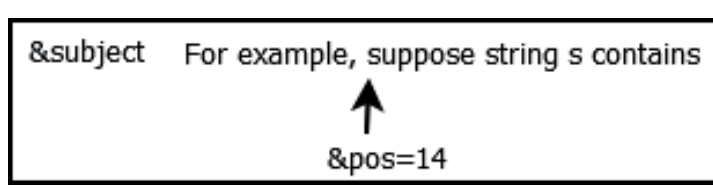


Рис. 15.2. Субъект и позиция после запятой и пробела после нее

Этот механизм является очень общим и позволяет использовать различные алгоритмы сопоставления паттернов. Теперь настало время погрузиться в детали того, как данная структура управления используется через свои операции.

Среды сканирования и их примитивные операции

Пара **(субъект, позиция)** называется **средой сканирования**. В структуре управления сканированием строк есть один оператор, две встроенные функ-

ции перемещения позиции и шесть встроенных функций анализа строк, которые выполняют вычисления, анализируя строку-субъект. Шесть встроенных функций анализа строк приведены в табл. 15.1. Более подробно они описаны в *приложении «Основы Unicon»*.

Таблица 15.1. Встроенные функции структуры управления строчным сканированием

Функция	Цель
any(C)	Является ли символ в данной позиции членом набора символов
many(C)	Являются ли 1+ символов в данной позиции членами набора символов
match(s)	Соответствуют ли символы в данной позиции строке поиска
find(s)	Выдать позицию(и), в которой символы совпадают со строкой поиска
upto(C)	Выдать позицию(и), в которой символ является членом набора символов
bal()	Выдать позицию(и), в которой символы сбалансированы по отношению к разделителям

Встроенными функциями двухпозиционного перемещения являются `move()` и `tab()`. Функция `move(n)` сдвигает индекс позиции на `n` букв относительно текущей позиции. Функция `tab(n)` – это абсолютное перемещение, устанавливающее позицию на индекс `n` в субъекте. Встроенные функции перемещения позиции обычно используются в сочетании с функциями анализа строк. Например, поскольку `find("suppose")` возвращает индекс, по которому может быть найдена строка «`suppose`", `tab(find("suppose"))` устанавливает позицию в это место. В примере, показанном на рис. 15.1, выполнение команды `tab(find("suppose"))` было бы одним из многих способов, с помощью которых среда сканирования может быть установлена в состояние, показанное на рис. 15.2. Другим способом достижения этого состояния является выполнение следующего кода:

```
tab(upto(',')) & move(1) & tab(match(« «))
```

Обычно примитивы анализа строк комбинируются таким образом для формирования более крупных и сложных паттернов. Встроенный в язык процесс перебора с возвратом, называемый **интегрированной целенаправленной оценкой**, означает, что предыдущие частичные совпадения будут отменены, если последняя часть конъюнкции завершится неудачей.

Комбинация `tab(match(s))` считается настолько полезной, что для нее определен унарный префиксный оператор =. Его не следует путать с бинарным оператором =, который выполняет числовое сравнение. В любом случае выражение `=s` эквивалентно `tab(match(s))`. Этот набор примитивов был придуман для Icon и сохранен в Unicon. Unicon добавляет сюда дополнительные механизмы (тип паттерна в стиле SNOBOL с регулярными выражениями в качестве литералов). Вам может быть интересно, добавили бы выразительности сканированию строк дополнительные операторы для других распространенных комбинаций анализа строк и функций перемещения позиций.

Структура управления сканированием строк в Icon и Unicon сильно контрастирует с монолитными операциями сопоставления с образцом, встречающимися в других языках обработки строк. Сканирование строк – это более общий механизм, чем регулярные выражения, в котором обычный код может вставляться в середине сопоставления с шаблоном. Следующий пример сканирования строки извлекает имена существительные собственные из строки *S* и сохраняет их в списке *L*:

```
S ? { L := []
      while tab(upto(&ucase)) do
              put(L, tab(many(&letters)))
}
```

Приведенный выше цикл `while` отбрасывает символы до тех пор, пока не будет найдена заглавная буква. Он рассматривает каждую заглавную букву как начало собственного имени и помещает это имя в список. Это действие не так лаконично, как хотелось бы, но оно очень общее и гибкое. Давайте посмотрим, как эта структура управления устраняет избыточные параметры функций анализа строк.

Устранение избыточных параметров с помощью структуры управления

Сканирование строк предоставляет стандартный набор значений параметров по умолчанию для встроенных в язык функций анализа строк. Все эти функции заканчиваются тремя параметрами – *строкой-субъектом, начальной позицией и конечной позицией*. Эти параметры по умолчанию соответствуют текущей среде сканирования, которая состоит из строки-субъекта `&subject`, текущей позиции `&pos` и конца строки-субъекта. Назначение параметров по умолчанию для структуры управления сокращает код и улучшает читаемость, решая одну из болевых точек, описанных в предыдущем разделе. Однако упрощение параметров – это еще не все влияние и цель сканирования строк.

Текущая среда сканирования строк видна в вызываемых функциях и имеет динамическую область видимости. Довольно просто написать вспомогательные функции, которые выполняют часть задачи анализа строк без необходимости передавать среду сканирования в качестве параметров.

Среды сканирования могут быть вложенными. В рамках сканирующего выражения или вспомогательной функции, когда подстрока требует дальнейшего анализа, это может быть выполнено путем введения другого выражения сканирования строки. Когда вводится новая среда сканирования, окружающая среда сканирования должна быть сохранена и восстановлена при выходе из вложенной подсреды. Такое поведение вложенности сохраняется в новой семантике выражений Icon и Unicon, ориентированной на цель, в которой выражения могут быть приостановлены, а затем возобновлены неявно. Среда сканирования сохраняется и восстанавливается в стеке. Эти операции являются более тонкими, но также зависят от активности процедур в стеке, например от вызовов процедур, приостановок, возобновления, возвратов и сбоев.

Для тех, кто хочет получить более подробную информацию, сканирование строк было подробно описано в других изданиях, например *Ralph E. Griswold and Madge T. Griswold. The Icon Programming Language, 3rd edition.* Реализация описана в разделе 9.6 «*The Implementation of Icon and Unicon*». Вкратце, в дополнение к сохранению и восстановлению окружения сканирования в стеке, две машинные инструкции байт-кода используются для упрощения генерации кода для этой структуры управления. Теперь давайте рассмотрим другую доменную структуру управления, которую мы введем в Unicon как часть средств трехмерной графики, – **рендеринг областей**.

Рендеринг областей в Unicon

В этом разделе описывается структура управления, называемая рендерингом (визуализацией) областей, которая была добавлена в Unicon во время написания этой книги. Поскольку эта особенность является новой, мы рассмотрим ее несколько подробнее. Структура управления рендерингом областей была в списке дел для Unicon в течение долгого времени, но добавление структуры управления может быть немного сложным, особенно если семантика нетривиальна, поэтому потребовалось написать данную главу, чтобы заняться этим. Сначала, однако, нам нужно подготовить сцену.

Отображение 3D-графики из списка отображения

Средства трехмерной графики Unicon определяют, что должно быть нарисовано, с помощью серии вызовов набора встроенных функций, а базовая среда выполнения отображает код, написанный на языках C и OpenGL, который рисует сцену столько раз в секунду, сколько возможно. Функции Unicon и код рендеринга на языке C взаимодействуют с помощью списка отображения. В основном функции Unicon помещают примитивы в конец списка отображения, а код рендеринга проходит по списку отображения и отрисовывает эти примитивы как можно быстрее.

В C API OpenGL существует похожий механизм отображения списков, который служит для предварительной упаковки и ускорения наборов примитивов путем их предварительного размещения на GPU, уменьшая узкое место между CPU и GPU. Однако Unicon – это динамический язык, в котором приоритет отдается гибкости, а не производительности. Чтобы управлять списком отображения на уровне кода приложения Unicon, список отображения Unicon представляет собой обычный список Unicon, а не список отображения C OpenGL.

Когда 3D-средства Unicon были впервые созданы, каждый графический примитив в списке отображения отображался каждый кадр. Это хорошо работало для небольших сцен. Для сцен с большим количеством примитивов становится непрактичным восстанавливать список отображения с нуля на каждом кадре. Требовались новые возможности, позволяющие приложениям быстро вносить изменения и выбирать примитивы из списка для отображения. Окончательная форма этих возможностей была неочевидной. Итак, давайте рассмотрим, как начинался рендеринг областей, как функциональный API.

Указание областей рендеринга с помощью встроенных функций

Выборочное отображение было введено в Unicon первоначально с помощью функции под названием `WSection()`. Символ W в этой функции означает окно и является общим префиксом для встроенных функций Unicon, связанных с графикой и оконными системами, поэтому это (оконная) функция секции. Два последовательных вызова `WSection()` определяют начало и конец секции, обычно называемой областью рендеринга. Области рендеринга позволяют легко включать и отключать коллекции 3D-примитивов в списке отображения между каждым кадром без необходимости перестраивать список отображения, вставлять или удалять элементы.

Первый вызов `WSection()` в паре представляет запись списка отображения с полем пропуска, которое можно включать и отключать. Второй вызов `WSection()` является маркером конца, который помогает определить, сколько примитивов списка отображения должно быть пропущено. Следующий пример рисует желтый ореол (изображающий доступный квест) над головой персонажа в виде тора:

```
questR := WSection(«Joe's halo»)
      Fg("diffuse translucent yellow")
      PushMatrix()
      npchaloT := Translate(0, h.headx+h.headr*3, 0)
      ROThalo := Rotate(0.0, 0, 1.0, 0)
      DrawTorus(0, 0, 0, 0.1, h.headr*0.3)
      PopMatrix()
WSection()
```

Вы не можете запустить этот пример в автономном режиме, поскольку он был взят из середины трехмерного приложения, которое занято рендерингом 3D-сцены. Отсутствующий контекст включает в себя открытое 3D-окно, в котором работают все эти функции, и текущий объект, в котором переменная класса `h` обозначает голову персонажа. Но, надеюсь, этот пример иллюстрирует, как вызовы `WSection()` используются в парах, определяющих начало и конец набора 3D-операций.

Большинство 3D-функций Unicon возвращают добавленный ими элемент списка отображения. Возвращаемое значение функции `WSection()` – это запись в списке отображения, которая влияет на поведение списка отображения для любого количества примитивов, входящих в данную секцию.

В приведенном выше примере кода ореол, будучи нарисованным, остается в списке отображения, но его можно сделать видимым или невидимым, отключив или установив флаг *skip*, присвоение значения *questR.skip := 1* приводит к исчезновению ореола. Рендеринг области вводит условный переход в структуру данных списка отображения.

Рендеринг областей также поддерживает выбор 3D-объектов. Параметр запуска `WSection()` задает строковое значение, которое возвращается, когда пользователь касается или кликает мышью на одном из 3D-примитивов данной секции.

Изменение графических уровней детализации с помощью вложенного рендеринга областей

Рендеринг областей поддерживает вложенность. В трехмерных сценах сложные объекты могут быть отображены путем обхода иерархической структуры данных, где самые крупные или наиболее важные графические элементы находятся в корне. Вложенный рендеринг областей поддерживает уровни детализации, где вторичные и третичные графические детали могут быть отображены в пределах подразделов и включены или отключены в зависимости от того, насколько близко или далеко объект находится от камеры. Уровни детализации могут быть важны для производительности, позволяя отображать детали пропорционально приблизительному расстоянию между зрителем и наблюдаемыми объектами. Существуют причудливые структуры данных, которые можно использовать для реализации этого уровня детализации, но рендеринг областей хорошо подходит для этого.

Код для рендеринга стула, например, может быть организован на трех уровнях детализации, используя три вложенные секции. Переменные `lod1`, `lod2` и `lod3` класса *Chair* будут связаны с тремя вложенными секциями в коде для полного рендеринга стула:

```
method full_render()
      lod1 := WSection()
      .   .   . render the big chair primitives
              lod2 := WSection()
              .   .   . render smaller chair primitives
                      lod3 := WSection()
                              .   .   . render tiny details in the chair
                      WSection()
              WSection()
      WSection()
end
```

После того как начальный `full_render()` вводит примитивы в список отображения, каждый раз, когда уровень рендеринга стула изменяется, метод `render()` в классе *chair* обновляет данные о том, что должно быть отображено и что должно быть пропущено, устанавливая флаги `skip`. Следующий код можно прочитать таким образом: если стул еще не был отрисован, выполните `full_render()`. Если он уже был отрисован, установите несколько флагов пропуска, чтобы указать степень детализации в зависимости от значения параметра `render_level` в диапазоне от `0` (невидимый) до `3` (полная детализация):

```
method render(render_level)
      if /rendered then return full_render()
      case render_level of {
              0: lod1.skip := 1
              1: { lod1.skip := 0; lod2.skip := 1 }
              2: { lod1.skip := lod2.skip := 0; lod3.skip := 1 }
              3: { lod1.skip := lod2.skip := lod3.skip := 0 }
              }
end
```

Этот механизм работает замечательно, но несколько болезненных поисков ошибок выявили проблему. В задуманном виде механизм секций был хрупким и подверженным ошибкам. Всякий раз, когда `WSection()` случайно помещается в неправильное место или неправильно вложена, программа работает ошибочно или возникают визуальные отклонения. Внедрение структуры управления упрощает использование рендеринга областей и снижает частоту ошибок, связанных с маркерами границ секций в списке отображения.

Создание структуры управления рендерингом областей

В этом подразделе будет описана реализация рендеринга областей в Unicon, чтобы дать представление о некоторых аспектах работы, связанной с внедрением новых структур управления для поддержки доменов приложений. В этой книге не описываются детали реализации Unicon, вместо этого в ней представлен минимум того, что необходимо сделать для сохранения удобочитаемости. За более подробной информацией о реализации Unicon вы можете обратиться к книге *«The Implementation of Icon and Unicon»*. Исходные файлы реализации, которые здесь изменяются, находятся в подкаталоге *uni/unicon* дистрибутива языка Unicon.

Чтобы добавить структуру управления, необходимо определить ее требования, синтаксис и семантику. Затем нужно добавить любые новые элементы в лексический анализатор, грамматику, деревья и таблицы символов. Могут потребоваться семантические проверки во время компиляции. После этого начинается основная работа по реализации структуры управления, которая заключается в добавлении правил к генератору кода для обработки любых новых форм, появляющихся в дереве синтаксиса для вашей структуры управления.

Дополнение должно быть максимально простым. Рендеринг областей требует наличия структуры управления, которая будет обеспечивать свойство совпадения пар при вызове функции `WSection()`.

Добавление зарезервированного слова для рендеринга областей

Для новой структуры управления мы добавляем в лексический анализатор Unicon новое зарезервированное слово *wsection*. Вы узнали, как добавлять зарезервированные слова в Jzero, в главе 3 «*Сканирование исходного кода*». Добавление одного из них в Unicon аналогично, поскольку лексический анализатор и парсер будут оба должны согласовать целочисленный код для нового зарезервированного слова, определяемый парсером.

Unicon был разработан до создания инструмента *uflex*, который был представлен в главе 3. В будущем Unicon может быть модифицирован для использования *uflex*, но в этом разделе описывается, как добавить зарезервированное слово к текущему рукописному лексическому анализатору Unicon, который в исходном коде Unicon называется *unilex.icn*. Зарезервированные слова хранятся в таблице, которая содержит для каждого зарезервированного слова два целых числа. Одно из них содержит пару булевых флагов для правил вставки точки с запятой, указывающих, является ли зарезервированное слово законным в начале (*Beginner*) и/или в конце (*Ender*) выражения. Другое

целое число содержит категорию терминального символа целого числа. Новое зарезервированное слово `wsection` будет *Beginner* в выражениях, поэтому точки с запятой могут быть вставлены в новые строки, которые непосредственно предшествуют ему. Запись таблицы для `wsection` в файле *unilex.icn* выглядит следующим образом:

```
t[«wsection»] := [Beginner, WSECTION]
```

Причина, по которой это дополнение лексического анализатора столь мало, заключается в том, что шаблон и код, которые необходимы для распознавания `wsection`, такие же, как и для других зарезервированных слов и идентификаторов. Чтобы этот код лексического анализатора работал, `WSECTION` должен быть объявлен в грамматике, как описано в следующем разделе, и файл *ytab_h.icn*, содержащий правила *#define* для терминальных символов, должен быть заново сгенерирован с помощью опции *-d* в *iyacc*.

Теперь пришло время использовать это новое зарезервированное слово в правиле грамматики.

Добавление правила грамматики

Добавление зарезервированного слова `wsection` позволяет использовать синтаксис, показанный здесь:

```
wsection expr1 do expr2
```

Это сделано для того, чтобы соответствовать остальному синтаксису Icon и Unicon. Зарезервированное слово `do` делает его похожим на цикл, но прецедентом является язык Pascal с оператором, который использует *do* и не является циклом. Добавление этого правила грамматики в *unigram.y* состоит из двух бит. В объявления терминальных символов добавляется следующее:

```
%token WSECTION /* wsection */
```

В основном разделе грамматики правила для добавления этой управляющей структуры в *unigram.y* следующие:

```
expr11 : wsection ;
wsection : WSECTION expr DO expr {
              $$ := node("wsection", $2, $4)
     };
```

Многие или большинство структур управления будут иметь семантические требования, например что первое выражение в предыдущем правиле – идентификатор секции – должно быть строкой. Поскольку Unicon является динамически типизированным языком, единственный способ, которым мы могли бы обеспечить выполнение такого правила во время компиляции, – это ограничить идентификаторы секций строковыми литералами. Мы решили не делать этого и применяем требование строки для первого выражения в сгенериро-

ванном коде. Но если ваш язык типизирован во время компиляции, вы могли бы добавить эту проверку в соответствующее место в обходе дерева, где выполняются иные проверки типов. Теперь давайте рассмотрим другие необходимые семантические проверки.

Проверка wsection на семантические ошибки

Цель структуры управления *wsection* – сделать рендеринг областей менее подверженным ошибкам. В дополнение к конструкции *wsection*, которая делает невозможным пропуск закрывающего вызова `WSection()` или случайное написание двух пересекающихся областей рендеринга, при каких еще обстоятельствах может произойти сбой рендеринга областей? Проблематичными являются операторы, которые выводят поток управления из области рендеринга неструктурированным образом. В Unicon к ним относятся *return*, *fail*, *suspend*, *break* и *next*. Однако если область рендеринга содержит циклы, то *break* или *next* внутри такого цикла вполне оправдан.

Таким образом, задача компилятора Unicon – решить, что делать в случае аномального выхода потока управления из области рендеринга. Для структуры управления сканированием строк правильным было бы реализовать сохранение и восстановление окружения сканирования в стеке, но рендеринг областей – это совсем другое.

Область рендеринга используется во время построения списка отображения, чтобы убедиться, что записи списка отображения хорошо сформированы. Список отображения используется в дальнейшем неоднократно в среде выполнения всякий раз, когда экран должен быть перерисован. Первоначальный поток управления, который использовался при построении списка отображения, не имеет к нему никакого отношения. По этой причине в *wsection* попытка преждевременного выхода, не достигнув конца области рендеринга, приводит к ошибке. Если программист хочет закодировать область рендеринга в неструктурированном виде, то на свой страх и риск он может явно вызывать `WSection()` в парах.

Обеспечение выполнения этих семантических правил требует, чтобы некоторая логика находилась в обходе (под)дерева всякий раз, когда в дереве синтаксиса встречается *wsection*. Обходы деревьев будут выглядеть несколько иначе в трансляторе Unicon, чем в Jzero, но в целом они похожи на Unicon-реализацию Jzero. Лучшее место для введения этой проверки – в методе *semantic()* класса *j0*, сразу после вызова метода *root.check_types()*, который выполняет проверку типов. Новая проверка в конце метода *semantic()* будет выглядеть следующим образом:

```
root.check_wsections();
```

В Unicon-файл *tree.icn* был добавлен следующий метод `check_wsections()`:

```
method check_wsections()
      if label == "wsection" then check_wsection()
      else every n := !children do
              n.check_wsections()
end
```

Вспомогательный метод, вызываемый для проверки правильности каждой конструкции *wsection*, называется `check_wsection()`. Это обход поддерева, ищущий узлы дерева, которые могли бы выйти из *wsection* неправильно, и сообщающий о семантической ошибке, если код пытается это сделать. Однако можно было бы сгенерировать код, который выполняет эти проверки во время выполнения, обеспечивая отложенное принудительное выполнение. Метод `check_wsection()` принимает необязательный параметр, который отслеживает вложенные циклы, содержащиеся в конструкции *wsection*, так что любые выражения *break* или *next*, вложенные в *wsection*, разрешены, если их действие не выходит за пределы *wsection*:

```
method check_wsection(loops:0)
      case label of {
              "return"| "Suspend0"| "Suspend1":
                      yyerror(label || " inside wsection")
              "While0"|"While1"|"until"|"until1"|
              "every"|"every1"|"repeat":
                      loops +:= 1
              "Next"|"Break":
                      if loops = 0 then
                              yyerror(label || " inside wsection")
                      else loops -:= 1
              "wsection": loops := 0
              }
      every n := !children do {
              if type(n) == "treenode" then
                      n.check_wsection(loops)
              else if type(n) == "token" then {
                      if n.tok = FAIL then
                              yyerror("fail inside wsection")
              }
      }
end
```

Приведенный выше код выполняет семантические проверки того, что структура управления *wsection* может обеспечить выполнение своего требования о том, что каждая открывающая `WSection(id)` имеет закрывающую `WSection()`. Теперь давайте рассмотрим генерацию кода для *wsection*.

Генерация кода для структуры управления `wsection`

Генерация кода для структуры управления *wsection* может быть смоделирована с помощью эквивалентных вызовов функции `WSection()`. В следующем примере использование *wsection* приведет к созданию кода, который соответствует примеру с ореолом, показанному ранее. Разница заключается в том, что, используя эту структуру управления, вы не можете забыть о закрывающей `WSection()`, случайно попытаться наложить их друг на друга и так далее:

```
questR := wsection select_id do {
      Fg("diffuse translucent yellow")
      PushMatrix()
      npchaloT := Translate(0, h.headx+h.headr*3, 0)
      ROThalo := Rotate(0.0, 0, 1.0, 0)
      DrawTorus(0, 0, 0, 0.1, h.headr*0.3)
      PopMatrix()
}
```

Чтобы понять генерацию кода для *wsection*, нам нужно семантическое правило для синтаксиса *wsection*, которое решает проблему в общем случае. Такое семантическое правило представлено в табл. 15.2. Вместо инструкций генерации промежуточного кода код представляется как преобразование источника в источник. Структура управления *wsection* реализуется с помощью некоторого полуфантастического кода Icon, который выполняет соответствующую пару вызовов функции `WSection()`, выдавая результат открывающего вызова `WSection()` в качестве результата всего выражения. Благодаря этому запись списка отображения может быть по желанию присвоена переменной в окружающем выражении:

Таблица 15.2. Семантическое правило для генерации кода структуры управления *wsection*

Продукт	Семантическое правило
`wsection: WSECTION expr1 DO expr2`	`wsection.code = “1(WSection(“ || expr1 || “),{“ || expr2|| “;WSection();1})”`

Код Icon в приведенном выше семантическом правиле требует некоторого пояснения. Выражение {$expr_2$; WSection(); 1} выполняет выражение $expr_2$, за которым следует закрывающая `WSection()`. Символ 1 после второй точки с запятой гарантирует, что все выражение успешно, поскольку оно оценивается окружающим выражением. Окружающее выражение имеет вид 1(WSection(...), {...}), которое сначала оценивает открывающую `WSection(...)`, а затем выполняет тело, но выдает возвращаемое значение открывающей `WSection()` как результат всего выражения.

Чтобы реализовать семантическое правило, показанное в табл. 15.2, и сделать фактический вывод кода, необходимо изменить процедуру генератора кода `yyprint()` в Unicon. `yyprint(n)` генерирует код для узла `n` дерева синтаксиса. `yyprint()` генерирует код как строку, выводимую в файл с именем `yyout`. В этом коде имеется множество различных переходов, практически по одному для каждого типа узла дерева, и эти переходы вызывают множество вспомогательных функций по мере необходимости. Для *wsection* функция `yyprint()` должна использовать следующий код, который можно добавить к случаю *treenode*:

```
else if node.label == «wsection» then {
      writes(yyout, "1(WSection("))
      yyprint(node.children[1])
      writes(yyout,"),{")
      yyprint(node.children[2])
      write(yyout, ";WSection();1})")
      fail
}
```

Причина, по которой это работает, когда структура управления доменом просто записывается как искусное расположение некоторых вызовов базовых функций, состоит в следующем. Основной компилятор Unicon является полутранспайлером, записывающим промежуточную форму, которая выглядит почти как исходный код. Точнее, промежуточная форма Unicon – это почти исходный код Icon. Очень много языков можно изобрести довольно быстро, если в качестве базового представления применяется другой язык очень высокого уровня, такой как Icon или Python.

Все это расширение языка Unicon, вероятно, побудило вас попробовать добавить свои структуры управления доменом. Надеюсь, что по мере того, как мы переходим к подведению итогов, у вас есть представление о том, как это сделать.

Заключение

В данной главе была рассмотрена тема структур управления доменами. Структуры управления доменами выходят далеко за рамки библиотек или даже встроенных функций и операторов в плане поддержки способности программистов решать проблемы в новых прикладных областях. В большинстве случаев структуры управления доменом упрощают код и уменьшают количество ошибок в программировании, которые были бы распространены, если бы программисты разрабатывали свой код, используя общие функции основного языка.

В следующей главе будет представлена сложная тема уборки мусора. Уборка мусора – это основная особенность языка, которая часто отличает низкоуровневые языки системного программирования от языков приложений более высокого уровня и доменно-ориентированных языков.

Вопросы

Ответьте на следующие вопросы, чтобы проверить свои знания по этой главе.

1. Структуры управления – это просто операторы `if` и циклы. Что в этом такого особенного?
2. Все, что позволяют вам сделать структуры управления, зависящие от домена приложения, – это предоставить некоторые значения по умолчанию для некоторых стандартных библиотечных функций. Зачем их использовать?

3. Какие дополнительные примитивы или семантика сделали бы структуру управления сканированием строк более полезной для программистов прикладной области?
4. Будет ли хорошей идеей для структуры управления *wsection* генерировать код, включая *PushMatrix()* и *PopMatrix()*, который окружает тело ее кода? Это позволило бы сделать пример более коротким и высокоуровневым.

Глава 16

Сборка мусора

Управление памятью – один из самых важных аспектов современного программирования, и почти любой язык, который вы изобретаете, должен обеспечивать автоматическое управление памятью через **сборку мусора**. В этой главе представлено несколько методов, с помощью которых вы можете реализовать сборку мусора в своем языке. Первый метод, называемый **подсчетом ссылок**, прост в реализации и имеет преимущество освобождения памяти по мере выполнения. Однако у подсчета ссылок есть фатальный недостаток. Второй метод, называемый **пометкой и освобождением**, представляет более надежный механизм, который гораздо сложнее реализовать, и его недостатком является то, что выполнение периодически приостанавливается, сколько бы времени ни занял процесс сборки мусора. Это два из многих возможных подходов к управлению памятью. Реализация сборщика мусора, не имеющего ни фатального недостатка, ни периодических пауз для сбора свободной памяти, может повлечь за собой другие связанные с этим расходы.

В этой главе рассматриваются следующие основные темы:

- оценка важности сборки мусора;
- подсчет ссылок на объекты;
- маркировка текущих данных и очистка остальных.

Цель этой главы – объяснить, почему сборка мусора важна, и показать, как это можно сделать. В число навыков, которые вы освоите, входят следующие: заставить объекты отслеживать количество ссылок, указывающих на них; идентифицировать все указатели на существующие данные в программах, включая указатели, расположенные внутри других объектов; освободить память и сделать ее доступной для повторного использования. Давайте начнем с обсуждения того, почему вы должны в любом случае беспокоиться обо всем этом.

Оценка важности сборки мусора

Вначале программы были небольшими, и **статическое распределение памяти** производилось при разработке программы. Код был не таким сложным, и программисты могли выделить всю память, которую они собирались использовать в течение всей программы, в виде набора глобальных переменных. Жизнь была хороша.

Затем произошел **закон Мура**[1], и компьютеры стали больше. Клиенты начали требовать, чтобы программы обрабатывали данные произвольного размера, вместо того чтобы соглашаться на фиксированные верхние пределы, присущие статическому распределению. Программисты изобрели **структурированное программирование** и использовали вызовы функций для организации программ большего размера, в которых большая часть памяти распределялась в **стеке**.

Стек – это форма **динамического распределения памяти**. Стеки хороши тем, что можно выделить большой кусок памяти при вызове функции и автоматически освободить память при возврате функции. Время жизни объекта локальной памяти привязано строго ко времени жизни вызова функции, в рамках которой он существует.

В конце концов люди заметили, что развитие программного обеспечения не поспевает за развитием аппаратного обеспечения. Мы столкнулись с кризисом программного обеспечения и попытались создать программную инженерию, чтобы справиться с этим кризисом. Иногда возникали ошибки, когда указатели на память, которая была освобождена в стеке, зависали, но это было редкостью и обычно являлось признаком начинающих программистов. Жизнь все еще была относительно хорошей. Затем закон Мура повторился.

Сейчас даже программы, работающие в наших наручных часах, слишком велики для понимания, и мы имеем программную среду, в которой во время выполнения программа может иметь миллиарды и миллиарды объектов. Клиенты ожидают, что вы сможете создать столько объектов, сколько они захотят, и эти объекты будут жить до тех пор, пока в них есть необходимость. Самой распространенной формой выделенной памяти является динамическая память, и она выделяется из области памяти, называемой **кучей**. Корректное использование области кучи является первоочередной задачей при разработке и реализации языков программирования.

В литературе по программной инженерии уже давно можно встретить утверждения, что от 50 до 75 % (или более) общего времени разработки программного обеспечения уходит на отладку. Это означает огромное количество времени и денег. По моему личному опыту нескольких десятилетий помощи студентам-программистам, в языках, где программисты сами управляют памятью, 75 или более процентов времени отладки тратится на ошибки управления памятью.

Это особенно актуально для новичков и программистов-неспециалистов, но случается как с новичками, так и с экспертами. Я рассматриваю C и C++. Теперь представьте, что проблема не только о том, сколько времени или денег отнимет управление памятью. По мере того как размер и сложность программ увеличиваются, вероятность того, что разработчики правильно управляют памятью вручную, падает, в результате возрастает вероятность того, что проект потерпит полный провал во время разработки или критический провал после развертывания.

---

[1] Закон Мура – *эмпирическое* наблюдение, изначально сделанное *Гордоном Муром*, согласно которому количество *транзисторов*, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца. – *Прим. перев.*

Какие виды ошибок управления памятью, спросите вы? Вы можете начать с этих: выделение недостаточного объема памяти; попытка использовать память сверх выделенного вами объема; вы забываете выделить память; не понимаете, когда вам нужно выделять память; забываете освободить память, чтобы ее можно было использовать повторно; освобождаете невыделенную память; попытка использовать память для определенной цели, после того как она была освобождена или повторно использована. Вот лишь несколько примеров.

Когда программы имеют скромный размер, а компьютеры, задействованные в них, ужасно дороги, имеет смысл максимизировать эффективность, тратя столько времени программиста на ручное управление памятью, сколько необходимо. Но по мере того, как программы становятся все длиннее, а компьютеры становятся все более дешевыми, с большим объемом памяти, практичность ручного управления памятью уменьшается. Автоматическое управление памятью неизбежно, и выполнение всего этого в стеке давно ушло в прошлое, когда структурированное программирование уступило место **объектно-ориентированной** парадигме.

Теперь у нас есть мир, в котором (для многих программ) большая часть интересующей памяти выделяется из кучи, где объекты живут произвольный промежуток времени, пока они не будут явно освобождены или неиспользуемая память не будет автоматически восстановлена. Вот почему сборщики мусора имеют первостепенное значение и заслуживают вашего внимания как разработчика языка.

Учитывая все это, реализация сборки мусора может быть сложной, а заставить ее работать хорошо – еще сложнее. Если вы перегружены, то, возможно, вам удастся отложить это до тех пор, пока успех вашего языка не потребует этого. Многие известные реализации языка (например, **Java** от **Sun**) обходились неадекватным сборщиком мусора или его отсутствием в течение многих лет. Но если вы серьезно относитесь к своему языку, то в конечном счете захотите иметь сборщик мусора для него. Давайте начнем с самого простого подхода, который называется *подсчетом ссылок*.

Подсчет ссылок на объекты

При **подсчете ссылок** каждый объект ведет подсчет того, сколько указателей ссылается на него. Это число начинается с 1, когда объект впервые выделяется и ссылка на него передается окружающему выражению. Счетчик ссылок инкрементируется, когда значение сохраняется в переменной, в том числе когда ссылка передается в качестве параметра или хранится в структуре данных. Счетчик декрементируется всякий раз, когда ссылка перезаписывается путем присвоения переменной для ссылки в другом месте или когда ссылка больше не существует (например, когда локальная переменная перестает существовать из-за возврата функции). Если счетчик ссылок достигает 0, память для этого объекта становится мусором, потому что на него ничего не указывает. Она может быть повторно использована для другой цели. Это кажется довольно разумным. Посмотрите, что нужно сделать, чтобы добавить подсчет ссылок в наш пример языка в этой книге, **Jzero**.

Добавление подсчета ссылок в Jzero

Jzero выделяет из кучи два типа объектов, которые могут быть собраны в мусор, – строки и массивы. Для таких объектов памяти, выделенных из кучи, представление Jzero в памяти включает размер объекта в его начальное слово. При подсчете ссылок второе слово в начале может содержать количество ссылок, указывающих на данный объект. Представление для строки показано на рис. 16.1.

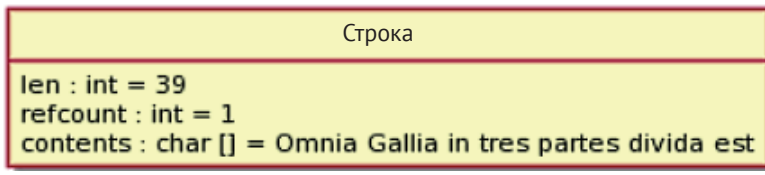


Рис. 16.1. Представление строки в памяти

В приведенном примере, если *len* и *refcount* равны 8 байтам каждый и имеется 39 байт строковых данных, *refcount* добавляет 8 к общему размеру в 55 байт (возможно, округляется до 56), так что добавление *refcount* – это всего лишь 14 % накладных расходов. Но если бы средняя длина строки составляла 3 байта и вам нужно было бы управлять миллиардами маленьких строк, добавление счетчика ссылок представляло бы значительные накладные расходы, которые могут ограничить масштабируемость вашего языка на больших данных. Учитывая такое представление, подсчет ссылок вступает в игру, когда объекты создаются в первую очередь. Поэтому давайте рассмотрим примеры операций, сгенерированный код которых включает использование кучи.

Генерация кода для распределения кучи

Когда создается такой объект, как `String`, для него должна быть выделена память. В некоторых языках объекты могут быть выделены в статической памяти, в стеке или в куче. Однако в Java (и Jzero) всем объектам выделяется память из кучи. Для строк это может вызвать недоумение, поскольку исходный код Java может включать строковые константы, которые обычно помещаются во время компиляции в статически выделенные адреса, а объекты кучи всегда распределяются во время выполнения. Предположим, что код выглядит следующим образом:

```
String s = «hello»;
```

С одной стороны, содержимое памяти строки *hello* может быть выделено в статической области памяти. С другой стороны, объект Jzero *String*, который мы присваиваем `String s`, должен быть экземпляром класса, выделенным из кучи, который содержит длину и счетчик ссылок вместе со ссылкой на символьные данные. Код, который мы генерируем в этом случае, может выглядеть следующим образом:

```
String s = new String(«hello»);
```

Если этот код будет выполняться миллиард раз, мы не хотим распределять миллиард экземпляров этой строки, нам нужен только один. В Java среда выполнения использует пул строк для строковых констант, так что ей нужно распределить только один экземпляр. Jzero не реализует классов Java *String* или *Stringpool*, но мы поместим статический метод с именем `pool()` в класс Jzero *String*, который возвращает ссылку на строку, распределяя экземпляр, если он еще не находится в пуле строк. Учитывая этот метод, сгенерированный код может выглядеть следующим образом:

```
String s = String.pool("hello");
```

Избежать выделения лишних строковых объектов позволяет тот факт, что объекты *String* являются неизменяемыми. Есть много способов генерации такого кода. Один из простых – это обход дерева, который заменяет узлы листьев со строковыми литералами поддеревьями, в которых вызывается метод *pool()*. Говоря конкретно, просто ищите узлы *STRINGLIT* и заменяйте их сконструированным набором узлов, показанным на рис. 16.2.

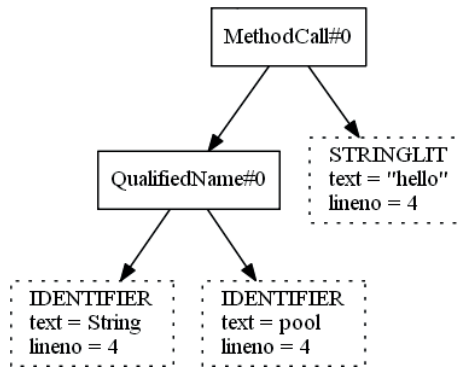


Рис. 16.2. Замена листа STRINGLIT для вызова метода pool

Код метода `poolStrings()`, который обходит дерево синтаксиса и выполняет эту подстановку, показан ниже. **Unicon**-реализация в файле *tree.icn* выглядит следующим образом:

```
method poolStrings()
     every i := 1 to *\kids do
            if type(\(\kids[i])) == "tree__state" then {
                   if kids[i].nkids>0 then kids[i].poolStrings()
                   else kids[i] := kids[i].internalize()
            }
end
```

Этот метод проходит по дереву, вызывая метод `internalize()` для замены всех листьев. Java-реализация `poolStrings()` в файле *tree.java* показана здесь:

```
public void poolStrings() {
      if (kids != null)
      for (int i = 0; i < kids.length; i++)
              if ((kids[i] != null) && kids[i] instanceof "tree") {
                      if (kids[i].nkids>0) kids[i].poolStrings();
                      else kids[i] = kids[i].internalize();
              }
}
```

Метод дерева с именем `internalize()` в этом обходе строит и возвращает поддерево, которое вызывает метод `String.pool()`, если он вызван на `STRINGLIT`. В противном случае он просто возвращает узел. В Unicon код выглядит следующим образом:

```
method internalize()
      if not (sym === "STRINGLIT") return self
      t4 := tree("token",parser.IDENTIFIER,
              token(parser.IDENTIFIER,"pool", tok.lineno, tok.colno))
      t3 := tree("token",parser.IDENTIFIER,
              token(parser.IDENTIFIER,"String", tok.lineno,
                      tok.colno))
      t2 := j0.node("QualifiedName", 1040, t3, t4)
      t1 := j0.node("MethodCall",1290,t2,self)
      return t1
end
```

Соответствующий код в Java выглядит так:

```
public tree internalize() {
      if (!sym.equals("STRINGLIT")) return this;
      t4 = tree("token",parser.IDENTIFIER,
              token(parser.IDENTIFIER,"pool", tok.lineno,
                      tok.colno));
      t3 = tree("token",parser.IDENTIFIER,
              token(parser.IDENTIFIER,"String", tok.lineno,
                      tok.colno));
      t2 = j0.node("QualifiedName", 1040, t3, t4);
      t1 = j0.node("MethodCall",1290,t2,this);
      return t1;
}
```

Этот код в компиляторе зависит от функции среды выполнения, которая реализует метод `String.pool()`, используя хеш-таблицу, чтобы избежать дубликатов. Теперь давайте рассмотрим изменения в генерации кода, которые необходимы для оператора присваивания.

Изменение сгенерированного кода для оператора присваивания

Подсчет ссылок основывается на изменении поведения присваивания, чтобы позволить объектам отслеживать ссылки, которые на них указывают. В про-

межуточном коде для Jzero была инструкция *ASN*, которая выполняла такое присваивание. Наша новая семантика подсчета ссылок для присваивания *x* = *y* могла бы состоять в следующем:

- если старое назначение (*x*) указывает на объект, декрементируйте его счетчик. Если счетчик равен нулю, освободите старый объект;
- выполните присваивание. Переменная *x* теперь ссылается на новое назначение;
- если новое назначение (*x*) указывает на объект, инкрементируйте его счетчик.

Интересным является вопрос, следует ли реализовать эту последовательность операций путем генерации множества трехадресных инструкций для присваивания или же семантика инструкции *ASN* должна быть изменена, чтобы выполнять перечисленные пункты автоматически при выполнении инструкции A*SN*. Часть ответа может зависеть от того, как вы относитесь к добавлению новых опкодов для *ASN*, когда задействованы объекты, возможно, с использованием *OASN* для присваивания объектов.

Учет недостатков и ограничений, связанных с подсчетом ссылок

Подсчет ссылок имеет несколько недочетов и один фатальный недостаток. Одним из недостатков является то, что оператор присваивания работает медленнее, поскольку он декрементирует счетчики объектов, удерживаемых до присваивания, и инкрементирует счетчики присваиваемых объектов. Это серьезный недостаток, так как присваивание является очень частой операцией. Другим недостатком является то, что размер объектов становится больше для хранения счетчиков ссылок, что очень неприятно, особенно для многочисленных небольших объектов, для которых дополнительный счетчик является значительным накладным расходом.

Фатальный недостаток возникает, если цепочка ссылок на объект может иметь цикл. Это очень распространенная практика в структурах данных. В случае цикла объекты, указывающие друг на друга, никогда не достигнут числа ссылок 0, даже если они недоступны для остальной части программы. Диаграмма на рис. 16.3 иллюстрирует циклический связанный список после того, как он стал мусором. Ни один внешний указатель никогда не сможет добраться до этой структуры, но, согласно подсчету ссылок, память, используемая этими объектами, не подлежит восстановлению.

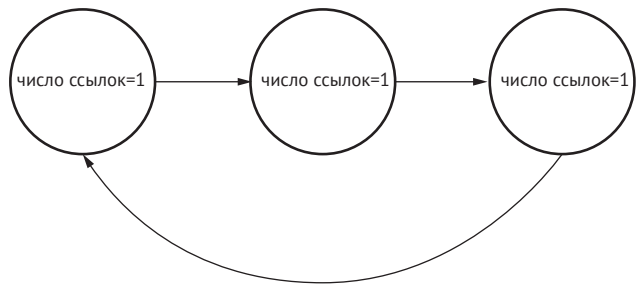


Рис. 16.3. Циклический связанный список не может быть обработан при подсчете ссылок

Несмотря на эти недостатки, подсчет ссылок относительно прост и легок, и он работает настолько хорошо, что это был, по-видимому, единственный метод сборки мусора в первой версии Python. В конечном итоге Python реализовал настоящий сборщик мусора в дополнение к подсчету ссылок, хотя после внедрения настоящего сборщика мусора подсчет ссылок становится ненужным и нерациональным расходом времени и места. В любом случае, из-за его фатальных недостатков большинство языков общего назначения не сочтут подсчет ссылок достаточным, поэтому давайте рассмотрим пример более надежного сборщика мусора, а именно реальный сборщик мусора методом сбора меток и развертки, используемый в языке программирования Unicon.

Пометка реальных данных и очистка остальных

В этом разделе дается обзор сборщика мусора Unicon, который представляет собой сборщик мусора типа mark-andsweep, который был разработан для языка Icon и затем расширен. Он написан на (расширенном диалекте) C, как и остальные части среды выполнения Icon и Unicon. Поскольку Unicon унаследовал этот сборщик мусора от Icon, многое из того, что вы видите здесь, связано с этим языком. Другие аспекты этого сборщика мусора описаны в книге *«Реализация Icon и Unicon: сборник» (The Implementation of Icon and Unicon: a Compendium)*.

Почти во всех сборщиках мусора, кроме использующих подсчет ссылок, подход заключается в том, чтобы найти все существующие указатели, доступные из всех переменных в программе, все остальное в куче является мусором. В сборщике с пометкой и освобождением (mark-and-sweep) существующие (живые) данные помечаются, когда они найдены, а затем все живые данные перемещаются в верхнюю часть кучи, оставляя большой красивый пул доступной памяти на дне. Функция `collect()` на языке C из среды выполнения Unicon представлена в общих чертах следующим образом:

```
int collect(int region) {
      grow_C_stack_if_needed();
      markprogram();
      markthreads();
      reclaim();
}
```

Интересно, что процесс сбора мусора в куче начинается с того, что мы убеждаемся, что у нас есть достаточно памяти в области стека C для выполнения этой задачи. В Unicon есть два стека – стек интерпретатора VM и стек, используемый C-реализацией VM. Необходимость увеличения стека C была обнаружена трудным путем. Причина в том, что алгоритм сборки мусора является рекурсивным, особенно операция обхода живых данных и пометки всего, на что они указывают. В некоторых компиляторах C и операционных системах стек C может увеличиваться автоматически по мере необходимости, но в других его размер может быть установлен явно. Код сборщика мусора делает это с помощью функции операционной системы, называемой `setrlimit()`. Код для увеличения стека C выглядит следующим образом:

```
void grow_C_stack_if_needed() {
      struct rlimit rl;
      getrlimit(RLIMIT_STACK , &rl);
      if (rl.rlim_cur < curblock->size) {
              rl.rlim_cur = curblock->size;
              if (setrlimit(RLIMIT_STACK , &rl) == -1) {
                      if (setrlimit_count != 0) {
                              fprintf(stderr,"iconx setrlimit(%lu) failed
                                      %d\n", (unsigned long)(rl.rlim_
                              cur),errno);
                              fflush(stderr);
                              setrlimit_count--;
                              }
              }
      }
}
```

Приведенный выше код проверяет, насколько велик стек C, и если область текущего блока больше, он запрашивает пропорциональное увеличение стека C. Это излишне для большинства программ, но примерно соответствует наихудшим требованиям. К счастью, память стоит недорого.

Фундаментальная предпосылка сборщика мусора Unicon заключается в том, что частые операции должны быть быстрыми, даже если это происходит за счет нечастых операций. В моем присутствии знаменитый компьютерный ученый Ральф Грисволд (Ralph Griswold) неоднократно замечал, что большинство программ никогда не собирают мусор, а завершают выполнение до того, как это произойдет. Это верно с определенной точки зрения. Это верно в различных областях применения, таких как утилиты обработки текста, и неверно в других прикладных областях, таких как серверы и любые другие приложения, которые выполняются в течение длительного времени.

Согласно доктрине быстрых частых операций, присваивания выполняются чрезвычайно часто и должны совершаться как можно быстрее – по этой причине подсчет ссылок является очень плохой идеей. Аналогично выделение памяти происходит довольно часто и должно быть настолько быстрым, насколько это возможно. Сборка мусора происходит нечасто, и вполне нормально, если ее стоимость пропорциональна затраченной работе.

Что еще более интересно, Icon и Unicon являются специализированными языками обработки строк и текстов, и строковый тип данных целиком и полностью специализирован в реализации. Оптимальная эффективность для строкового типа может заставить некоторые программы, в которых много строк, работать на этих языках очень хорошо, а другие программы – нет.

Организация областей памяти кучи

В связи с важным особым случаем строк Unicon имеет два вида кучи. Общая куча, называемая **областью блоков**, позволяет выделять любые типы данных, кроме строк. А отдельная куча, называемая **областью строк**, предназначена для строковых данных.

Блоки являются самоописывающимися для целей сборки мусора. Схема области блоков представляет собой последовательность блоков. Каждый блок начинается с заглавного слова, которое идентифицирует его тип. Многие типы блоков имеют фиксированный размер, типы блоков с изменяющимся размером имеют поле `size` в слове после заглавного слова. Область блоков показана на рис. 16.4. Прямоугольник слева – это область `struct`, которая управляет областью блоков (показанной прямоугольником справа). Управляемая область блоков может быть размером во много мегабайт и содержать тысячи или миллионы блоков.

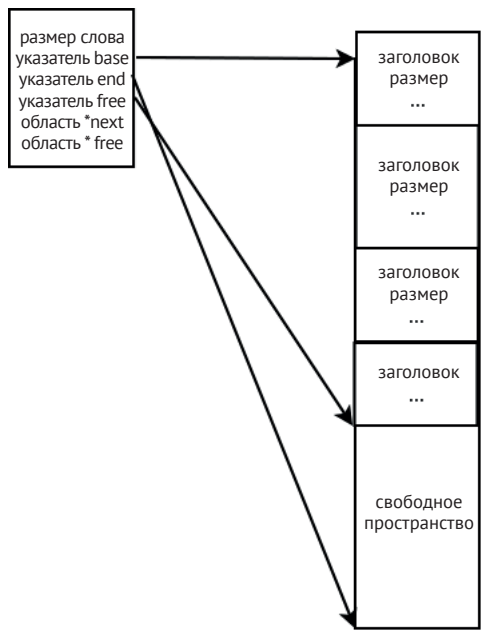


Рис. 16.4. Область блоков

В области блоков выделение происходит очень быстро. Чтобы выделить блок размером *n* для экземпляра класса или другой структуры, такой как список или таблица, просто убедитесь, что *n* меньше, чем оставшееся свободное пространство между указателями *free* и *end*. В этом случае новый блок размещается по указателю *free*, и область обновляется для его учета путем добавления *n* к указателю *free*.

В отличие от области блоков, область строк представляет собой необработанные неструктурированные строковые данные. Области строк организованы так, как показано на рис. 16.4, за исключением того, что фактические строковые данные не имеют никаких заголовков, размеров или другой структуры – это необработанный текст. Из-за того, что строки не выделяются в виде блоков, как все остальное, некоторые распространенные операции со строками, такие как срезы, не выполняются. Аналогично область строк может быть выровнена по байтам без потери места, когда выделяется много маленьких строк, в отличие от области блоков, которая выравнивается по словам. Кроме того, данные в области строк никогда не содержат ссылок на другую живую память, поэтому

отделение строк от области блоков уменьшает общий объем памяти, в которой должны быть найдены ссылки.

В любой момент времени существует одна текущая область блока и одна текущая область строки, из которых может быть выделена память. Каждая программа и каждый поток имеют свои текущие области блоков и строк, которые являются активными областями в двунаправленном связанном списке всех областей кучи, выделенных для данной программы или потока.

Когда область заполнена и запрашивается больше памяти, происходит сборка мусора из текущей кучи. Более старые области в связанном списке являются областями-срочниками, и в них мусор собирается только тогда, когда сборка мусора в текущей области не позволяет освободить достаточно памяти для выполнения запроса. Когда ни одна область в списке не может удовлетворить запрос, должна быть выделена новая область. Когда сборке мусора не удается освободить достаточно места для удовлетворения запроса на память, создается новая текущая область того же типа, обычно в два раза больше, чем предыдущая, и добавляется в связанный список.

Обход базиса для пометки живых данных

При первом проходе сборки мусора помечаются живые данные. Должны быть найдены все указатели на память кучи в программе. Это начинается с **базисного набора** переменных, состоящего из глобальной и статической памяти, и включает все локальные переменные в стеке, которые должны быть пройдены. Объекты кучи, на которые указывают все эти глобальные и локальные переменные, помечаются.

Задача маркировки живых данных в среде выполнения Unicon в общих чертах представлена в следующем примере кода. Первые два элемента базисного набора состоят из переменных, выделяемых на основе каждой программы и каждого потока. В Icon они первоначально были глобальными переменными, но по мере развития виртуальной машины глобальные переменные превратились в поля структуры, и поиск всех базисных переменных в этих категориях превратился в серию обходов структуры данных, чтобы добраться до них всех:

```
static void markprogram(struct progstate *pstate) {
      struct descrip *dp;
      PostDescrip(pstate->K_main);
      PostDescrip(pstate->parentdesc);
      PostDescrip(pstate->eventmask);
      PostDescrip(pstate->valuemask);
      PostDescrip(pstate->eventcode);
      PostDescrip(pstate->eventval);
      PostDescrip(pstate->eventsource);
      PostDescrip(pstate->AmperPick);
      PostDescrip(pstate->LastEventWin);/* last Event() win */
      PostDescrip(pstate->Kywd_xwin[XKey_Window]);/*&window*/
      postqual(&(pstate->Kywd_prog));
      for (dp = pstate->Globals; dp < pstate->Eglobals; dp++)
              if (Qual(*dp)) postqual(dp);
              else if (Pointer(*dp)) markblock(dp);
```

```
        for (dp = pstate->Statics; dp < pstate->Estatics; dp++)
                if (Qual(*dp)) postqual(dp);
                else if (Pointer(*dp)) markblock(dp);
        }
```

Пометить все глобальные переменные в программе – задача несложная:

```
static void markthreads() {
        struct threadstate *t;
        markthread(&roottstate);
        for (t = roottstate.next; t != NULL; t = t->next)
                if (t->c && (IS_TS_THREAD(t->c->status))) {
                        markthread(t);
                }
}
```

Каждый поток помечается следующим вызовом функции `markthread()`. Некоторые фрагменты состояния потока содержат вещи, которые, как известно, не содержат ссылок на переменные кучи, но те поля, которые могут содержать указатели на кучу, должны быть помечены:

```
static void markthread(struct threadstate *tcp) {
        PostDescrip(tcp->Value_tmp);
        PostDescrip(tcp->Kywd_pos);
        PostDescrip(tcp->ksub);
        PostDescrip(tcp->Kywd_ran);
        PostDescrip(tcp->K_current);
        PostDescrip(tcp->K_errortext);
        PostDescrip(tcp->K_errorvalue);
        PostDescrip(tcp->T_errorvalue);
        PostDescrip(tcp->Eret_tmp);
}
```

Фактический процесс маркировки отличается для *строк* и *объектов*. Поскольку переменные Unicon могут содержать значения любого типа, макрос `PostDescrip()` используется для определения того, является значение строкой или указателем другого типа, или ни тем, ни другим. Ссылки на строки называются *квалификаторами*, и они помечаются с помощью функции `postqual()`. Указатели других типов помечаются с помощью функции `markblock()`:

```
#define PostDescrip(d) \
        if (Qual(d)) postqual(&(d)); \
        else if (Pointer(d)) markblock(&(d));
```

Чтобы интерпретировать этот макрос, вам нужно больше, чем просто `postqual()` и `markblock()`. Вам также необходимо знать, что делают макросы тестирования `Qual()` и `Pointer()`. Краткий ответ заключается в том, что они выполняют побитовое *AND* для проверки значения одного бита в дескрипторном слове значения Unicon. Значение является строкой, если самый верхний (знаковый) бит дескрипторного слова, называемый `F_Nqual`, равен 0. Если этот

бит равен 1, то оно не является строкой, а остальные биты флага могут быть использованы для проверки других свойств, из которых флаг указателя `F_Ptr` указывает на то, что значение содержит указатель, возможно, указатель на значение в куче:

```
#define Qual(d) (!((d).dword & F_Nqual))
#define Pointer(d) ((d).dword & F_Ptr)
```

Эти тесты быстрые, но они выполняются много раз во время сборки мусора. Если бы мы придумали более быстрый способ, чем показанный в макросе `Post-Descrip()`, для определения значений для потенциальной маркировки живых строк и блоков, это могло бы значительно повлиять на производительность сборки мусора.

Маркировка области блоков

Для блоков метка перезаписывает часть объекта с указателем на переменную, которая указывает на объект. Если на объект указывает более одной переменной, строится связанный список из этих живых ссылок по мере их нахождения. Этот связанный список необходим для того, чтобы все эти указатели могли быть обновлены, дабы указывать на новое местоположение, если объект будет перемещен. Функция `markblock()` состоит из более чем 200 строк кода. В обобщенном виде она представлена в следующем примере кода:

```
void markblock(dptr dp) {
      dptr dp;
      char *block, *endblock;
      word type;
      union block **ptr, **lastptr;
      block = (char *)BlkLoc(*dp);
      if (InRange(blkbase, block, blkfree)) {
              type = BlkType(block);
              if ((uword)type<=MaxType)endblock=
                      block+BlkSize(block);
              BlkLoc(*dp) = (union block *)type;
              BlkType(block) = (uword)&BlkLoc(*dp);
              if ((uword)type <= MaxType) {
                      ... пройти по любым указателям в блоке
                      }
      else ... обрабатывать другие типы блоков, которые не будут перемещаться
}
```

Обход указателей внутри блока зависит от того, как организованы блоки в языке. Указатели внутри блока всегда представляют собой непрерывный массив. Глобальная таблица в сборщике мусора с именем `firstp` указывает, по какому смещению байта для каждого типа блока могут быть найдены его вложенные указатели. Вторая глобальная таблица с именем `firstd` сообщает, по какому смещению байта для каждого типа блока найдены его дескрипторы (вложенные значения, которые могут быть чем угодно, а не только указателем блока). Эти таблицы просматриваются следующим кодом:

```
        ptr = (union block **)(block + fdesc);
        numptr = ptrno[type];
        if (numptr > 0) lastptr = ptr + numptr;
        else
                lastptr = (union block **)endblock;
        for (; ptr < lastptr; ptr++)
                if (*ptr != NULL)
                        markptr(ptr);
        }
    if ((fdesc = firstd[type]) > 0)
        for (dp1 = (dptr)(block + fdesc);
                        (char *)dp1 < endblock; dp1++) {
                if (Qual(*dp1)) postqual(dp1);
                else if (Pointer(*dp1)) markblock(dp1);
                }
```

Указатели вложенных блоков посещаются путем обхода с переменной *ptr* с вызовом `markptr()` для каждого из них. Функция `markptr()` аналогична функции `markblock()`, но может посещать другие типы указателей, кроме блоков. Вложенные дескрипторы посещаются путем прохода с помощью переменной `dp1` и вызова `postqual()` для строк и `markblock()` для блоков.

Для строк массив с именем `quallist` строится из всех указателей живых строк (включая их длины), которые указывают на текущую область строк. Функция с именем `postqual()` добавляет строку в массив `quallist`:

```
    void postqual(dptr dp) {
        if (InRange(strbase, StrLoc(*dp), strfree)) {
                if (qualfree >= equallist) {
                        newqual = (char *)realloc((char *)quallist,
                                (msize)(2 * qualsize));
                        if (newqual) {
                                quallist = (dptr *)newqual;
                                qualfree = (dptr *)(newqual + qualsize);
                                qualsize *= 2;
                                equallist = (dptr *)(newqual + qualsize);
                                }
                        else {
                                qualfail = 1;
                                return;
                                }
                        }
                *qualfree++ = dp;
        }
    }
```

Большая часть приведенного выше кода состоит из расширения размера массива, если это необходимо. Размер массива удваивается каждый раз, когда требуется дополнительное пространство.

Кроме того, если объект содержит какие-либо другие указатели, их необходимо посетить, а то, на что они указывают, должно быть помечено и пройдено, рекурсивно следуя за всеми указателями на все, до чего можно добраться.

Восстановление живой памяти и размещение ее в непрерывных фрагментах

Во время второго прохода процесса сборки мусора кучи обходятся сверху вниз, и все живые данные перемещаются в верхнюю часть. Общая стратегия восстановления памяти показана в следующем коде. Обратите внимание, что сборка мусора осложняется параллельными потоками – здесь мы не рассматриваем параллелизм во всех подробностях:

```
static void reclaim()
      {
      cofree();
      if (!qualfail)
              scollect((word)0);
      adjust(blkbase,blkbase);
      compact(blkbase);
      }
```

Восстановление памяти заключается в освобождении статической памяти без ссылок, состоящей из совместных выражений, которые стали мусором в функции `cofree()`, затем перемещении всех живых строковых данных вверх в функции `scollect()`, а потом перемещении блочных данных вверх путем вызова функции `adjust()`, за которым следует `compact()`.

Функция `cofree()` проходит по каждому блоку совместных выражений. Эти блоки не могут быть выделены в области блоков, потому что они содержат переменные, которые нельзя перемещать. Эта функция состоит из следующего кода:

```
void cofree() {
      register struct b_coexpr **ep, *xep;
      register struct astkblk *abp, *xabp;
      ep = &stklist;
      while (*ep != NULL) {
              if ((BlkType(*ep) == T_Coexpr)) {
                      xep = *ep;
                      *ep = (*ep)->nextstk;
                      for (abp = xep->es_actstk; abp; ) {
                              xabp = abp;
                              abp = abp->astk_nxt;
                              if ( xabp->nactivators == 0 )
                                      free((pointer)xabp);
                              }
                      free((pointer)xep);
                      }
              else {
                      BlkType(*ep) = T_Coexpr;
                      ep = &(*ep)->nextstk;
                      }
              }
      }
```

Приведенный выше код проходит по связанному списку блоков совместных выражений. Когда код посещает блок совместных выражений, в заголовке которого все еще написано `T_Coexpr`, это указывает на то, что блок не был помечен как живой. В этом случае совместное выражение и связанные с ним блоки памяти освобождаются с помощью стандартной библиотечной функции `free()`.

Функция `scollect()` собирает область строк, используя список всех живых указателей на нее. Она сортирует массив `quallist` с помощью стандартной библиотечной функции `qsort()`. Затем проходит по списку и копирует живые строковые данные в верхнюю часть области, обновляя указатели на область строк по мере определения новых местоположений строк. При этом уделяется внимание указателям на перекрывающиеся строки, чтобы такие строки оставались смежными:

```
static void scollect(word extra) {
     char *source, *dest, *cend;
     register dptr *qptr;
     if (qualfree <= quallist) { strfree = strbase; return; }
     qsort((char *)quallist,
            (int)(DiffPtrs((char *)qualfree,(char *)quallist)) /
                   sizeof(dptr *), sizeof(dptr),
                          (QSortFncCast)qlcmp);
     dest = strbase;
     source = cend = StrLoc(**quallist);
     for (qptr = quallist; qptr < qualfree; qptr++) {
            if (StrLoc(**qptr) > cend) {
                   while (source < cend) *dest++ = *source++;
                   source = cend = StrLoc(**qptr);
                   }
            if ((StrLoc(**qptr) + StrLen(**qptr)) > cend)
                   cend = StrLoc(**qptr) + StrLen(**qptr);
            StrLoc(**qptr) = StrLoc(**qptr) +
                              DiffPtrs(dest,source)+(uword)extra;
            }
     while (source < cend) *dest++ = *source++;
     strfree = dest;
     }
```

Функция `adjust()` является первой частью сбора области блоков. Она проходит по области блоков, перемещая указатели в области блоков в те места, куда будут указывать блоки. Во время разметки был построен связанный список всех указателей на каждый блок, он используется для обновления всех этих указателей на новое местоположение блоков. Исходный код для функции `adjust()` показан ниже:

```
void adjust(char *source, char *dest) {
     register union block **nxtptr, **tptr;
     while (source < blkfree) {
            if ((uword)(nxtptr = (union block **)BlkType(source))>
                   MaxType) {
```

```
                while ((uword)nxtptr > MaxType) {
                        tptr = nxtptr;
                        nxtptr = (union block **) *nxtptr;
                        *tptr = (union block *)dest;
                        }
                BlkType(source) = (uword)nxtptr | F_Mark;
                dest += BlkSize(source);
                }
        source += BlkSize(source);
        }
}
```

Функция `compact()` является заключительным шагом в сборе области блоков, как показано в следующем блоке кода. Она заключается в перемещении самих блоков памяти вверх на их новое место. Заглавные слова живых блоков очищаются, когда блок перемещается на свое новое место:

```
void compact(char *source) {
     register char *dest;
     register word size;
     dest = source;
     while (source < blkfree) {
             size = BlkSize(source);
             if (BlkType(source) & F_Mark) {
                     BlkType(source) &= ~F_Mark;
                     if (source != dest)
                             mvc((uword)size,source,dest);
                     dest += size;
                     }
             source += size;
             }
     blkfree = dest;
     }
```

Из этого раздела вы должны сделать вывод, что сборщик мусора для маркировки и зачистки является нетривиальной и относительно низкоуровневой задачей. Если вам нужно стимулирование, подумайте вот о чем: работа по созданию сборщика мусора направлена на благое дело – он сэкономит бесчисленные усилия программистов, использующих ваш язык, и они будут вам благодарны за это. Многие изобретатели языков до вас успешно реализовали сборку мусора, и вы можете это сделать.

Заключение

В данной главе вы многое узнали о *сборке мусора*. Вы узнали, что такое мусор, как он возникает, и увидели два совершенно разных способа борьбы с ним. Простой способ, популяризированный в некоторых ранних системах Lisp и ранних версиях Python, называется *подсчетом ссылок*. При подсчете ссылок сами выделенные объекты становятся ответственными за их сбор. Обычно это работает.

Более сложная форма сборки мусора включает в себя поиск всех живых данных в программе и обычно перемещение их, чтобы избежать фрагментации памяти. Поиск живых данных, как правило, является рекурсивным, требует обхода стека для поиска ссылок в параметрах и локальных переменных и обычно представляет собой обременительную и низкоуровневую задачу. Было реализовано множество вариаций этой общей идеи. Одно из основных наблюдений, которым пользуются некоторые сборщики мусора, заключается в том, что большинство выделенных объектов используются лишь в течение короткого времени, а затем почти сразу становятся мусором.

Любой метод, который вы используете, чтобы избавить программистов от необходимости управлять собственной памятью, вероятно, будет высоко оценен.

В следующей главе мы завершим книгу размышлениями о том, чему мы научились.

Вопросы

1. Предположим, что конкретное значение Unicon, например значение *null*, особенно часто встречается при маркировке живых данных. При каких обстоятельствах имело бы смысл модифицировать макрос `PostDescrip()` для проверки этого значения, чтобы увидеть, можно ли избежать тестов в макросах `Qual()` и `Pointer()`?
2. Каковы преимущества и недостатки создания отдельной области кучи для каждого типа класса?
3. Операция `reclaim()` сборщика мусора Unicon перемещает всю живую немусорную память в верхнюю часть области. Будет ли полезно модифицировать этот сборщик, чтобы живые данные не перемещались?

Глава 17

Заключительные размышления

Узнав так много о создании языка программирования, вы, возможно, захотите поразмышлять о том, что вы узнали, и подумать о том, какие области хотели бы изучить более глубоко. В этой главе отражены основные темы, представленные в книге, и дана пища для размышлений по следующим направлениям:

- размышления о том, чему автор научился в ходе написания этой книги;
- решение, куда двигаться дальше;
- изучение ссылок для дальнейшего чтения.

Давайте начнем с того, какие дополнительные бонусы можно извлечь из данной книги.

Размышления о том, что изучено при написании этой книги

В ходе написания этой книги автор узнал несколько полезных вещей. Среди прочего он пришел к выводу, что на данный момент Java очень подходит для написания компиляторов. Конечно, Эндрю Аппель (Andrew Appel) мог в 1997 году опубликовать книгу *«Современная реализация компиляторов на Java» (Modern Compiler Implementation in Java)*, и другие книги по написанию компиляторов на Java тоже существуют. Они могут быть замечательными, но многие авторы компиляторов не будут рассматривать возможность использовать Java, если это означает отказ от lex и YACC. Использование стандартного набора инструментов lex/YACC для Java делает его более совместимым с кодовыми базами компиляторов, созданными для других языков.

Я хочу выразить свою признательность сопровождающему Byacc/J *Томашу Хурке (Tomas Hurka)* за принятие и улучшение моего патча *static import*, чтобы Byacc/J лучше работал с Jflex и подобными инструментами (включая мой инструмент Merr, как описано в главе 4 «*Парсинг*»), которые генерируют *yylex()* или *yyerror()* в отдельных файлах. Поддержка *yylex()* и *yyerror()* в отдельных файлах устраняет необходимость в глупых обходных путях, таких как написание метода заглушки *yylex()* внутри класса парсера, который вызывает *yylex()*, созданный в другом файле. Кроме того, различные небольшие усовершенствования Java после его первоначального выпуска, такие как возможность использо-

вать значения *String* для случаев *switch*, имеют большое значение для удобства составителя компилятора. На данный момент удобства и преимущества Java по сравнению с C практически перевешивают его недостатки, которых много. Не будем делать вид, что жесткая структура пакетов и классов для каталогов и файлов Java и отсутствие механизмов *#include* или *#ifdef* не имеют издержек.

Я написал эту книгу не для того, чтобы решить, хорош ли Java для компиляторов. Я написал эту книгу, чтобы сделать Unicon отличным для компиляторов. Маленьким чудом данной книги стало нахождение способа использовать одни и те же лексические и синтаксические спецификации для Unicon и Java. В итоге я действительно счастлив, что смог использовать оба языка так же, как я бы традиционно писал компилятор на C. После этого замечательного совместного использования спецификаций lex/YACC Unicon не предоставил столько дополнительных преимуществ, сколько я ожидал, по сравнению с Java. Unicon оставляет многие болевые точки Java, он несколько более лаконичен и легче работает с неоднородными типами структур. В конечном счете оба языка отлично подошли для написания компилятора Jzero, и я позволю вам судить о том, какой код был более читабельным и удобным. Теперь давайте рассмотрим, куда вы можете пойти дальше.

Решение о том, куда двигаться дальше

Вы можете захотеть изучить более сложные работы в любой из областей. К ним относятся дизайн языка программирования, реализация байт-кода, оптимизация кода, мониторинг и отладка выполнения программ, среды программирования, такие как IDE, и построители GUI. В этом разделе мы более подробно рассмотрим лишь некоторые из этих возможностей. Данный раздел отражает многие из моих личных предубеждений и приоритетов.

Изучение дизайна языков программирования

Выявить сильные работы в области дизайна языков программирования, вероятно, сложнее, чем в большинстве других технических тем, упомянутых в этом разделе. Гарольд Абельсон (Harold Abelson) и Джеральд Сассман (Gerald Sussman) когда-то написали книгу под названием «*Структура и интерпретация компьютерных программ*» (*Structure and Interpretation of Computer Programs*), которая широко известна как полезная. Хотя это не совсем книга по проектированию языков программирования, ее содержание затрагивает эту тему.

При случайном поиске вы можете найти множество книг по языкам программирования общего назначения. «*Продвинутый дизайн языков программирования*» (*Advanced Programming Language Design*) Рафаэля Финкеля (Rafael Finkel) – одна из них, охватывающая целый ряд продвинутых тем. Что касается других источников, то книги по дизайну языков и статьи, написанные настоящими изобретателями языков, могут быть более реальными и полезными, чем книги, написанные третьими лицами.

Одно из светил дизайна языков Никлаус Вирт (Niklaus Wirth) написал множество важных книг. «*Алгоритмы и структуры данных*» (*Algorithms and Data Structures*), а также «*Проект Оберон*» (*Project Oberon*) содержат ценные сведения относительно дизайна языков, и их реализации. Как разработчик нескольких

успешных языков, включая Pascal и Modula-2, Никлаус Вирт имеет большой авторитет в аргументации за простоту языковых конструкций, которые защищают программистов от самих себя. Он – гигант, на чьи плечи вам будет полезно встать.

Язык программирования **Prolog** породил богатую литературу, описывающую многие из проблем дизайна и реализации, которые были рассмотрены для этого языка и логического программирования в целом. Prolog важен тем, что в нем реализован *обширный неявный перебор с возвратом*. Одной из важных работ по Prolog является «*Искусство Prolog*» (*Art of Prolog*) Леона Стерлинга (Leon Sterling) и Эхуда Шапиро (Ehud Shapiro). Другим важным вкладом является **модель «ящика Берда»** (**Byrd box model**) для функций, в которой вместо понимания публичного интерфейса функции как вызова, за которым следует возврат, функция рассматривается как имеющая вызов, производящая результат и возобновляемая неоднократно, вплоть до окончательного отказа.

Следующее великое семейство языков программирования, которое заслуживает внимания, – это **SmallTalk**. SmallTalk не открыл объектно-ориентированную парадигму, но очистил и популяризировал ее. Краткое изложение некоторых принципов его проектирования было опубликовано в журнале Byte в статье Дэна Ингаллса (Dan Ingalls) под названием «*Принципы дизайна Smalltalk*» (*Design Principles Behind Smalltalk*). Рассматривая объектно-ориентированные языки, разумно также рассмотреть полуобъектно-ориентированные языки, такие как C++, в области которых ценна книга Бьярне Страуструпа (Bjarne Stroustrup) «*Дизайн и эволюция C++*» (*Design and Evolution of C++*).

Резкий рост популярности языков высокого уровня, таких как Python и Ruby, является одним из самых важных событий последних десятилетий. Удручает то, насколько слабо представлены многие чрезвычайно популярные языки в литературе по дизайну языков программирования. Изобретатель TCL Джон Оустерхаут (John Ousterhout) написал две важные работы на темы, связанные с дизайном языков очень высокого уровня. «*Скриптинг: высокоуровневое программирование для XXI века*» (*Scripting: Higher-Level Programming for the 21st Century*) – хорошая работа, хотя и отражающая предвзятость ее автора. Оустерхаут также выступил с важным докладом под шутливым названием «*Почему потоки – плохая идея*» (*Why Threads Are a Bad Idea*), в котором привел доводы в пользу событийно-ориентированного программирования и синхронных сопрограмм вместо потоков для большинства параллельных нагрузок.

Языки Icon и Unicon являются двумя более хорошо документированными примерами языков очень высокого уровня. Дизайн языка Icon описан в книге Грисвольда (Griswold) «*История языка программирования Icon*» (*History of the Icon Programming Language*). Рассмотрев некоторые прекрасные возможности изучения дизайна языков, давайте разберем варианты изучения их реализации.

Изучение реализации интерпретаторов и машин байт-кода

Темы реализации продвинутых языков программирования должны включать реализацию всех типов интерпретаторов и сред выполнения для продвинутых языков программирования с новой семантикой. Первым языком очень высокого уровня был Lisp. Его изобретателю Джону Маккарти (John McCarthy) приписывают изобретение математической нотации, которая может быть вы-

полнена на компьютере, одного из первых интерактивных интерпретаторов и, возможно, первого компилятора «точно в срок». Другие разработчики Lisp написали заметные книги. Особого внимания заслуживает «*Анатомия Lisp*» (*Anatomy of Lisp*) Джона Аллена (John Allen).

Любое описание машин байт-кода было бы небрежным, если бы в нем не были упомянуты машины байт-кода Pascal. Многие основополагающие работы по реализации языка Pascal собраны в книге «*PASCAL: язык и его реализация*» (*PASCAL: The Language and Its Implementation*) под редакцией Дэвида Баррона (David Barron). Система UCSD Pascal, которая популяризировала машины байт-кода, была основана на работе Урса Амманна (Urs Ammann) в Швейцарском федеральном технологическом институте (Цюрих), которая хорошо представлена в книге Баррона. Другой значительной работой над Pascal является работа Стивена Пембертона (Steven Pemberton) и Мартина Дэниелса (Martin Daniels) «*Реализация Pascal: компилятор и интерпретатор*» (*Pascal Implementation: Compiler and Interpreter*), которая представляет собой общедоступный ресурс.

В подборке книг, авторами которых являются Адель Голдберг (Adele Goldberg) и ее соавторы, описан SmallTalk, очень продвинутый язык, лучше, чем почти любой другой. Сюда входит «*SmallTalk-80: язык и его реализация*» (*SmallTalk-80: The Language and its Implementation*).

В мире логического программирования **абстрактная машина Уоррена (Warren Abstract Machine — WAM)** является одним из главных средств рассуждения о базовой семантике языка Prolog и о том, как ее реализовать. Она описана в книге «*Абстрактный набор инструкций PROLOG*» (*An Abstract PROLOG Instruction Set*).

Реализация Unicon описана в *сборнике «Реализация Icon и Unicon» (The Implementation of Icon and Unicon: a Compendium)*. Эта книга объединяет и обновляет несколько предыдущих работ по реализации языка Icon, а также описания реализации различных подсистем, которые были добавлены в Unicon.

Приобретение опыта в оптимизации кода

Оптимизация кода обычно является темой учебников по компиляторам для продвинутых выпускников. Классический учебник «*Компиляторы: принципы, методы и инструменты*» (*Compilers: Principles, Techniques, and Tools*) содержит обширную информацию по различным оптимизациям. Более современной работой является «*Проектирование компилятора*» (*Engineering a Compiler*) Купера (Cooper) и Торчона (Torczon).

Оптимизация кода для языков более высокого уровня часто требует более новых методов. Различные работы по оптимизации компиляторов для языков очень высокого уровня, кажется, предполагают, что потребуется 20 лет, чтобы люди поняли, как исполнять такие языки эффективно. В качестве намека на это я сошлюсь на такие примеры, как «*T: диалект Lisp*» (*T: a Dialect of Lisp*) и «*Разработка и реализация компилятора SELF*» (*The Design and Implementation of the SELF Compiler*), которые вышли через 20 лет после языков Lisp и Smalltalk. Конечно, то, сколько времени это займет, зависит от размера и сложности языка. Я необъективен, но одной из моих любимых работ по таким языкам является диссертация «*Реализация оптимизирующего компилятора для Icon*» (*The*

implementation of an optimizing compiler for Icon), которая входит в сборник реализаций Icon и Unicon. Она вышла всего через дюжину, или около того, лет после изобретения Icon, так что, вероятно, оптимизация здесь возможна.

Мониторинг и отладка выполнения программ

Существует много книг об отладке кода конечного пользователя, но мало книг о том, как писать мониторы и отладчики программ. Отчасти проблема заключается в том, что методы их реализации являются низкоуровневыми и сильно зависят от платформы, поэтому многое из того, что написано про реализацию отладчика, может быть справедливо только для одной конкретной операционной системы и может стать неверным через 5 лет.

Что касается общей картины, вы, возможно, захотите подумать о том, как разработать свой отладчик и какой интерфейс он должен предоставлять конечному пользователю. Помимо подражания интерфейсу основных отладчиков, вы должны рассмотреть понятие отладки на основе запросов, описанное в книге Раймондаса Ленцевичуса (Raimondas Lencevicius) «*Продвинутые методы отладки*» (*Advanced Debugging Methods*). Вам также следует рассмотреть понятия относительной отладки и дельта-отладки, которые были популяризированы в работах Дэвида Абрамсона (David Abramson) и др. и Андреаса Целлера (Andreas Zeller).

Одна из вещей, о которых вы, возможно, захотите прочитать, если хотите узнать больше о реализации отладчика, – это форматы исполняемых файлов и, в частности, информация об их отладочных символах. Переносимый формат исполняемых файлов Microsoft Windows задокументирован на сайте Microsoft.

Одним из наиболее известных соответствующих форматов UNIX является **формат линковки исполняемых файлов (executable linking формат – ELF)**, который хранит отладочную информацию в формате, называемом **отладкой с произвольными форматами записи (Debugging With Arbitrary Record Formats – DWARF)**.

Отладчик GNU, известный как GDB, является достаточно популярным, он имеет руководство «*Внутренние компоненты GDB*» (*GDB Internals*) и часто используется в качестве основы, на которой разрабатываются отладочные возможности. На сайте *https://aarzilli.github.io/debuggerbibliography/* перечислено несколько других ресурсов по реализации отладчика, в основном ориентированных на язык **Go**.

Для содержательного обсуждения классической литературы по мониторингу выполнения программ вы можете прочитать «*Мониторинг выполнения программ: обзор*» (*Monitoring Program Execution: A Survey*) или соответствующую главу в книге «*Мониторинг и визуализация программ*» (*Program Monitoring and Visualization*).

Проектирование и реализация IDE и построителей GUI

Одним из основных элементов успеха языков программирования является степень, в которой их среда программирования поддерживает написание и отладку кода. В этой книге лишь вкратце затронуты эти темы, и вы, возможно, захотите больше узнать о том, как реализуются IDE и их построители пользовательского интерфейса.

Здесь есть хорошие и плохие новости. Плохая новость заключается в том, что почти никто не написал книгу по созданию собственной интегрированной среды разработки. Если бы вы собирались создать ее с нуля, то могли бы начать с самообразования по написанию текстового редактора, а затем добавить другие возможности. В этом случае вам стоит обратиться к книге Крейга Финсета (Craig Finseth) «*Искусство редактирования текста*» (*The Craft of Text Editing*). Эта книга была написана человеком, который изучал реализацию текстового редактора Emacs для своей бакалаврской работы. Там также есть глава под названием *GNU Emacs Internals*, написанная в качестве приложения к руководству по GNU Emacs.

Хорошей новостью является то, что почти никто не должен писать текстовый редактор в интегрированной среде разработки. Каждая из основных графических вычислительных платформ поставляется с библиотекой пользовательского интерфейса, которая включает текстовый редактор в качестве одного из своих виджетов. Вы можете собрать интерфейс интегрированной среды разработки, используя инструмент построителя графического интерфейса. К сожалению, библиотеки графических пользовательских интерфейсов обычно не переносимы и недолговечны. Это означает, что работа по программированию на них почти обречена на то, чтобы быть выброшенной в течение десятилетия или двух. Требуются огромные усилия, чтобы написать код, который работает на всех платформах и живет вечно в интернете.

Итак, этот раздел должен быть посвящен многоплатформенным переносимым библиотекам графического пользовательского интерфейса и их использованию для написания интегрированных сред разработки и построителей пользовательских интерфейсов. Java – один из самых переносимых языков, и даже после нескольких неудачных стартов все еще вероятно, что некоторые из лучших и наиболее многоплатформенных переносимых библиотек пользовательского интерфейса могут быть библиотеками Java.

Изучение ссылок для дальнейшего чтения

Здесь представлена подробная библиография работ, которые мы обсуждали в предыдущем разделе. Внутри каждого подраздела работы перечислены в алфавитном порядке по авторам.

Изучение дизайна языков программирования

В области дизайна языков программирования вам могут быть интересны следующие материалы:

- Harold Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs*, Second edition, MIT Press, 1996.
- Rafael Finkel, *Advanced Programming Language Design*, Pearson, 1995.
- Ralph Griswold, *History of the Icon Programming Language*, Proceedings of HOPL-II, ACM SIGPLAN Notices 28:3 March 1993, pages 53–68.
- Daniel H. H. Ingalls, *Design Principles Behind Smalltalk*, Byte Magazine August 1981, pages 286–298.
- John Ousterhout, *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer 31:3, March 1998, pages 23–30.

- John Ousterhout, *Why Threads Are a Bad Idea (for most purposes)*, Invited talk, USENIX Technical Conference, September 1995 (available at *https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf*).
- Leon Sterling and Ehud Shapiro, *The Art of Prolog*, MIT Press, 1986.
- Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994.
- Niklaus Wirth, *Algorithms and Data Structures*, Prentice Hall, 1985.
- Niklaus Wirth, *Project Oberon: The Design of an Operating System and Compiler*, Addison Wesley/ACM Press 1992.

Это крошечная выборка лучших работ из богатого литературного наследия, и она, возможно, содержит множество печальных упущений. Теперь давайте посмотрим на аналогичный список по реализации.

Изучение реализации интерпретаторов и машин байт-кода

В области реализации интерпретаторов и машин байт-кода вы можете найти следующие материалы, представляющие интерес:

- John Allen, *Anatomy of Lisp*, McGraw Hill, 1978.
- Urs Ammann, *On Code Generation in a PASCAL Compiler*, Software Practice and Experience 7 (3), 1977, pages 391–423.
- David W. Barron, ed., *PASCAL-The Language and Its Implementation*, John Wiley, 1981.
- Adele Goldberg, David Robson, *SmallTalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
- Clinton Jeffery and Don Ward, eds., *The Implementation of Icon and Unicon: a Compendium*, Unicon Project, 2020 (available at *http://unicon.org/book/ib.pdf*).
- A. B. Vijay Kumar, *Supercharge Your Applications with GraalVM*, Packt, 2021.
- Steven Pemberton and Martin Daniels, *Pascal Implementation: The P4 Compiler and Interpreter*, Ellis Horwood, 1982 (available at *https://homepages.cwi.nl/~steven/pascal/*).
- David Warren, *An Abstract PROLOG Instruction Set*, Technical Note 309, SRI International, 1983 (available at *http://www.ai.sri.com/pubs/files/641.pdf*).

Теперь давайте рассмотрим аналогичный список для собственного кода и оптимизации кода.

Приобретение опыта работы с собственным кодом и оптимизации кода

Что касается собственного кода и оптимизации кода, то вам могут быть интересны следующие материалы:

- Al Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman, *Compilers: Principles Techniques and Tools*, Second edition, Addison Wesley, 2006.
- Craig Chambers, *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Stanford dissertation, 1992.
- Keith Cooper and Linda Torczon, *Engineering a Compiler*, Second edition, Morgan Kaufmann, 2011.

- Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, March 2004. Available at *https://llvm.org/pubs/2004-01-30-CGO-LLVM.html*
- Jonathan Rees and Norman Adams, *T: a dialect of Lisp*, Proceedings of the 1982 ACM symposium on LISP and functional programming, pages 114–122.
- Kenneth Walker, *The implementation of an optimizing compiler for Icon*, Arizona dissertation, 1991.

После оптимизации вы, возможно, захотите более подробно изучить такую узкоспециализированную область, как мониторинг и отладка выполнения программ.

Мониторинг и отладка выполнения программ

В области мониторинга и отладки вам могут быть интересны следующие материалы:

- David Abramson, Ian Foster, John Michalakes, and Roc Sosic, *Relative Debugging: A new methodology for debugging scientific applications*, Communications of the ACM 39 (11), November 1996, pages 69–77.
- DWARF Debugging Information Format Committee, *DWARF Debugging Information Format Version 5* (*http://www.dwarfstd.org*), 2017.
- John Gilmore and Stan Shebs, *GDB Internals*, Cygnus Solutions, 1999. The most recent copy is in wiki format and available at *https://sourceware.org/gdb/wiki/Internals*.
- Clinton Jeffery, *Program Monitoring and Visualization*, Springer, 1999.
- Raimondas Lencevicius, *Advanced Debugging Methods*, Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.
- Microsoft, PE Format, available at *https://docs.microsoft.com/en-us/windows/win32/debug/pe-format*.
- Bernd Plattner and J. Nievergelt, *Monitoring Program Execution: A Survey*. IEEE Computer, Vol. 14. November 1981, pages 76–93.
- Andreas Zeller, Why Programs Fail: *A Guide to Systematic Debugging*, Second edition, Morgan Kaufmann, 2009.

Наряду с мониторингом и отладкой было бы полезно рассмотреть интегрированные инструменты программирования для вашего языка.

Проектирование и реализация IDE и построителей GUI

В области сред разработки и построителей пользовательского интерфейса вам могут быть интересны следующие материалы:

- Craig Finseth, *The Craft of Text Editing: Emacs for the Modern World*. Springer, 1990.
- Bill Lewis, Dan LaLiberte, Richard Stallman, the GNU Manual Group, et al., GNU Emacs Internals, *Appendix E within the GNU Emacs Lisp Reference Manual*, GNU Project, 1990–2021, pages 1208–1250.

Честно говоря, я бы хотел, чтобы у меня было больше хорошей литературы, которую я мог бы порекомендовать в области IDE и построителей GUI. Если вы

знаете хорошие работы на эту тему, присылайте мне свои предложения. Итак, давайте подведем итоги.

Заключение

Из этой книги вы узнали кое-что о создании языков программирования. Мы сделали это, показав вам реализацию игрушечного языка под названием Jzero. Однако Jzero сам по себе неинтересен, интересны инструменты и методы, используемые при его реализации. Мы даже реализовали его дважды!

Если вы думали, что разработка и внедрение языков программирования – это, возможно, плавательный бассейн, которым можно наслаждаться, то ваш новый вывод может быть таким: это больше похоже на океан. Если это так, то инструменты, предоставленные в ваше распоряжение в этой книге, включая версии flex и YACC для использования с Unicon или Java, являются роскошным круизным лайнером, способным доставить вас по этому океану туда, куда вы хотите попасть.

Говорят, что на создание первого компилятора языка высокого уровня ушло 18 лет. Возможно, сейчас это задача нескольких месяцев, хотя это все еще бессрочная задача, на которую вы можете потратить столько времени, сколько сможете выделить, чтобы улучшить любой компилятор или интерпретатор, который захотите написать.

Святым Граалем компиляторов уже давно является высокоуровневая декларативная спецификация проблемы генерации кода, соответствующего декларативной спецификации правил лексики и синтаксиса. Несмотря на серьезную работу многих людей, гораздо более умных, чем я, этот ожидаемый прорыв не состоялся. Вместо него появилось несколько костылей. Само понятие машины байт-кода, реализованной в переносимом системном языке, таком как C, сделало многие языки переносимыми на огромное количество процессоров… как только кто-то перенесет на них компилятор языка C. Это стало частью мейнстрима благодаря таким технологиям, как .NET CLR и машины байт-кода Java JVM и GraalVM. Аналогичным образом получили широкое распространение транспайлеры, которые преобразуют исходный код в код на другом языке высокого уровня (например, C).

Третья форма повышения переносимости, которая доступна изобретателям языков программирования, – это распространение форматов целевых инструкций промежуточного уровня, таких как LLVM. Все эти широко используемые способы сделать ваш язык программирования переносимым уклоняются от задачи генерации кода для совершенно нового процессора. Возможно, четвертая форма увеличения переносимости связана с тем, что в настоящее время создается мало новых наборов инструкций для процессоров, поскольку промышленность коллективно инвестировала так много в небольшое количество аппаратных наборов инструкций, для которых доступны генераторы оптимизирующего кода.

Спасибо, что прочитали эту книгу. Я надеюсь, что, несмотря на ее многочисленные недостатки, вы смогли насладиться моей книгой и нашли ее полезной. Я с нетерпением жду, какие новые языки программирования вы изобретете в будущем!

Часть IV

Приложение

В эту часть войдут материалы, которые помогут читателям понять основной текст. Данная часть включает в себя следующий материал:

- приложение «Основы Unicon».

Приложение

Основы Unicon

В этом приложении представлено достаточно информации о языке Unicon, чтобы помочь вам понять примеры кода на нем в нашей книге. Данное приложение предназначено для опытных программистов и не тратит время на изложение основных концепций программирования. Вместо этого представление Unicon фокусируется на его интересных или необычных особенностях по сравнению с распространенными языками.

Если вы знаете Java, то большинство кода Unicon в этой книге можно понять, посмотрев соответствующий код Java. Здесь вы можете посмотреть все, что не является очевидным или не объясняется сравнением с Java. Это приложение не является полным справочником по языку Unicon, для этого смотрите *приложение A* (*Appendix A*) к книге «*Программирование с Unicon*» (*Programming with Unicon*), которая доступна в отдельном виде в открытом доступе в *Unicon Technical Report #8*. *Programming with Unicon* и *Unicon Technical Report #8* размещены на сайте *unicon.org*.

> **Сокращенные обозначения**
>
> В данном приложении используются квадратные скобки [] для обозначения необязательных объектов и звездочки * для обозначения объектов, которые могут встречаться ноль или более раз. Если квадратные скобки или звездочки выделены, это означает, что они встречаются в коде Unicon, а не как необязательные или повторяющиеся объекты.

В этом приложении рассматриваются следующие темы:

- запуск Unicon;
- использование деклараций и типов данных Unicon;
- оценка выражений;
- проблемы отладки и среды выполнения;
- мини-справочник по функциям;
- избранные ключевые слова.

Для начала приведем расширенное обсуждение запуска программ Unicon.

Запуск Unicon

Unicon вызывается для компиляции и запуска либо из командной строки, либо изнутри IDE. Имена исходных файлов Unicon заканчиваются расширением

.icn, а имена объектных файлов Unicon заканчиваются расширением *.u*. Вот несколько примеров вызова транслятора Unicon:

- `unicon mainname [ filename(s) ]`
  Компилирует и линкует *mainname.icn* и другие файлы, чтобы сформировать исполняемый файл с именем *mainname.exe* в Windows или просто *mainname* в большинстве других платформ.
- `unicon -o exename [ filename(s) ]`
  Компилирует и линкует исполняемый файл с именем *exename* или, в Windows, *exename.exe*.
- `unicon -c filename(s)`
  Компилирует файлы *.icn* в файлы *.u*, но не линкует их.
- `unicon -u filename(s)`
  Предупреждает о необъявленных переменных.
- `unicon -version`
  Выводит версию Unicon.
- `unicon -features`
  Выводит характеристики данной сборки Unicon.
- `unicon foo -x`
  Компилирует и запускает *foo.icn* за один шаг.

Более подробное описание того, как запустить Unicon под Windows, можно прочитать на сайте *http://unicon.org/utr/utr7.html*. Полный список опций командной строки можно посмотреть на сайте *http://unicon.org/utr/utr11.html*.

Если вам не нравится работать из командной строки, вы можете попробовать IDE Unicon под названием *ui*. Программа *ui* имеет опции для компиляции и выполнения программ в графическом интерфейсе. Пример этого показан на рис. А.1.

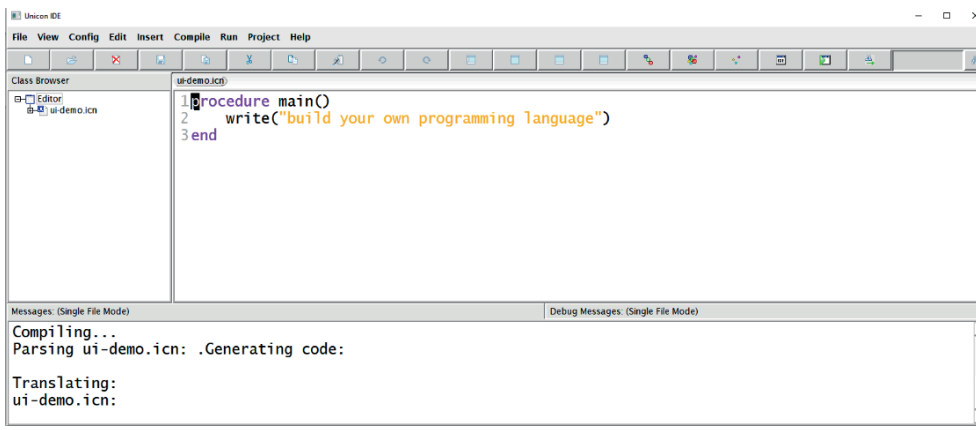


Рис. А.1. Скриншот IDE Unicon ui

Создатели Unicon используют множество различных сред программирования, и IDE Unicon – это скорее демонстрация технологии, чем производственный инструмент, но вы можете найти его полезным, хотя бы из-за его любимого меню **Help**. Оно написано примерно в 10 000 строк Unicon, не считая

библиотеки классов GUI. Теперь давайте рассмотрим, какого типа декларации допускаются в Unicon и какие типы данных он поддерживает.

Использование объявлений и типов данных Unicon

Вы не можете написать программу Unicon, не объявив ничего. Объявление чего-либо – это акт связывания имени, видимого в некоторой области видимости и времени жизни, с некоторым фрагментом кода или памяти, способным хранить значение. Далее давайте узнаем, как могут быть объявлены различные компоненты программы.

Объявление различных типов компонентов программы

Программы Unicon состоят из одной или нескольких процедур, начинающихся с *main()*. Структура программы часто также включает классы. Unicon отличает определяемые пользователем процедуры от функций, которые встроены в язык. Следующие образцы показывают структуру синтаксиса первичных объявлений тел кода в процедурах и методах Unicon.

Объявление процедуры:

`procedure` *X* (*params*) [*locals*]* [*initial*] [*exprs*]* **`end`**

Объявление метода:

method *X* (*params*) [*locals*]* [*initial*] [*exprs*]* **`end`**

Процедура или метод имеет имя, параметры и тело, заканчивающееся словом *end*. Тело может начинаться с локальных и статических объявлений и начальной секции, за которыми следует последовательность выражений. Методы могут быть объявлены только внутри классов, внутри которых они имеют доступ к дополнительному набору имен полей класса и других методов. В Unicon нет статических методов, и все методы являются публичными.

Объявление параметров:

[*var* [**`:`** *expr*] [**`,`** *var* [**`:`** *expr*]]* [**`,`** *variable* **`[]`**]]

Объявление имен полей:

[*var* [**`,`** *var*]*]

Параметры имеют ноль или более имен, разделенных запятыми. Каждый параметр может опционально включать двоеточие, за которым следует значение по умолчанию или имя функции принуждения типа. За последним параметром могут следовать квадратные скобки, указывающие на то, что переменное количество аргументов будет передано в виде списка. Параметры используются для процедур и методов, включая методы *initially*. Имена полей, которые объявляются в объявлениях записей и классов, проще, чем список параметров, состоящий из списка идентификаторов, разделенных запятыми.

Объявление глобальных переменных:

`global` *variable* [**`,`** *variable*]*

Объявление локальных переменных:

local *variable* [**:=** *expr*] [**,** *variable* [**:=** *expr*]]*

Объявление сатических переменных:

static *variable* [**:=** *expr*] [**,** *variable* [**:=** *expr*]]*

Переменные могут быть введены с помощью списка имен, разделенных запятыми, в одной из трех областей видимости – **глобальной**, **локальной** или **статической**. Локальные имена могут содержать присваивание для инициализации переменной. Глобальные переменные живут в течение всего времени выполнения программы. Локальные переменные живут в течение времени выполнения одной процедуры или вызова метода. Статические переменные живут в течение всего времени выполнения программы, и одна копия каждой статической переменной используется всеми вызовами этой процедуры или метода.

Объявление типа записи:

record *R* (*fields*)

Объявление класса:

class *C* [**:** *super*]* (*fields*) [*methods*]* [*initially*] **end**

Запись или класс объявляется именем, за которым следует список имен полей, разделенных запятыми, окруженный круглыми скобками. В имени записи или класса объявляется глобальная переменная, которая содержит функцию-конструктор, которая создает экземпляры. Класс также может иметь список имен суперклассов, разделенных двоеточием. Объявление класса содержит ноль или более методов и опциональную секцию *initially*, за которой следует зарезервированное слово *end*.

Объявление метода initially:

initially [(*params*)] [*locals*]* [*initial*] [*exprs*]*

Секция *initially* – это специальный, опциональный метод инициализации, который вызывается автоматически конструктором класса. Если секция *initially* присутствует, она должна быть после всех других методов, непосредственно перед концом класса. Ей не предшествует слово *method*, и ее список параметров является необязательным.

Ссылочные библиотечные модули:

link *module* [**,** *module*]*

Программы Unicon могут включать несколько файлов в командной строке, но используемые файлом модули могут быть также объявлены в исходном коде. Модули могут быть строковыми константными именами файлов или идентификаторами, используемыми в качестве имен файлов.

Использование пакетов:

import *package* [**,** *package*]*

Пространство глобальных имен Unicon может состоять из нескольких именованных пакетов, которые можно импортировать, указав имя (имена) пакета.

Теперь давайте рассмотрим типы данных Unicon.

Использование атомарных типов данных

Unicon имеет богатый набор типов данных. Атомарные типы неизменяемы, а структурные типы изменяемые. Они появляются непосредственно в исходном коде в виде литеральных константных значений или вычисляются и вводятся в программу операторами либо функциями.

Числовые

Целые – это целые числа со знаком произвольной точности. Целые числа являются наиболее распространенным типом и работают очевидным образом. Существует множество литеральных форматов с основаниями от 2 до 36, а набор суффиксов, таких как K и M, умножает числа на тысячи или миллионы. Например, целочисленный литерал *4G* обозначает значение в четыре миллиарда. Целые числа в Unicon в основном просто работают, не привлекая нашего внимания. Предусмотрены все обычные арифметические операторы наряду с удобным оператором возведения в степень *x^y*. Интересный унарный оператор *?n* выдает случайное число от 1 до `n`. Унарный оператор `!n` генерирует целые числа от 1 до *n*.

Вещественный тип данных обеспечивает аппроксимацию чисел с плавающей точкой. Вещественные константы должны содержать десятичную мантиссу, показатель степени, либо и то, и другое. Удивительно, сколько проблем доставляли программистам вещественные значения и как они теперь воспринимаются как должное – вещественные значения имеют тот же размер, что и 64-битные целые числа, хотя двоичный формат отличается. Одна из проблем, с которой вы иногда сталкиваетесь, – это преобразование туда и обратно между целыми и вещественными числами. Преобразование выполняется автоматически по мере необходимости, но отнимает время, если вы делаете это многократно и без необходимости.

Текстовые

Unicon имеет несколько встроенных типов для работы с текстом, включая строки, **наборы символов (character sets – csets)** и удивительный тип шаблона, заимствованный из SNOBOL4. В этой книге используются строки и csets, но вместо шаблонов применяются Flex и Yacc, поскольку они более переносимы. По этой причине мы не будем представлять тип шаблона или его формат литерала, основанный на регулярных выражениях.

Строки – это упорядоченные последовательности из нуля или более символов. Строковый литерал окружен двойными кавычками и может включать в себя управляющие (escape-) символы. Эти escape-последовательности приведены в табл. А.1.

Csets – это неупорядоченные совокупности из нуля или более не дублирующих друг друга символов. Литерал cset окружен одинарными кавычками и может включать в себя управляющие символы. В Unicon имеется множество констант ключевых слов cset для предопределенных наборов символов, которые встречаются в виде макросов или тестовых функций в других языках. Оказы-

вается, что наличие полного набора типов данных для символов полезно при обработке текста. Тип cset поддерживает обычные операторы множеств, такие как *c1*++*c2*, который вычисляет объединение, или *c1*--*c2* для символов в *c1*, но не в *c2*. Теперь давайте перейдем к рассмотрению структурных типов Unicon.

Таблица А.1. escape-символы строк и cset

Код	Символ	Код	Символ	Код	Символ	Код	Символ
\b	удаление символа перед курсором	\d	удалить	\e	escape	\f	ввод формы
\l	перевод строки	\n	новая строка	\r	возврат каретки	\t	табуляция
\v	вертикальная табуляция	\’	кавычка	\”	двойная кавычка	\\	обратный слеш
ooo	восьмеричный	\x*hh*	шестнадцатеричный	\^x	удаление выделенного в буфер		

Организация нескольких значений с помощью структурных типов

Структурные типы – это значения, которые состоят из нескольких значений. Структурные типы являются изменяемыми, что означает, что значения могут быть изменены или заменены. Обычно они создаются во время выполнения программы действием, которое выделяет память и инициализирует ее значениями их компонентов. Многие структурные типы являются контейнерами, позволяющими вставлять и удалять значения. Первый структурный тип, который следует рассмотреть, – это класс, который представляет собой определяемый пользователем структурный тип.

Классы

Если ваши данные не числовые и не текстовые, вы, вероятно, захотите написать для них класс в Unicon. Каждый класс Unicon – это новый тип данных. Типы данных класса используются для конструирования объектов из домена приложения. Обычно они используются для объектов, которые содержат несколько частей информации, имеющих сложное поведение.

Unicon определяет семантику множественного наследования интересным способом, называемым **наследование на основе замыкания**, который позволяет циклы в схеме наследования. Тот факт, что все классы Unicon являются публичными и виртуальными, упрощает ситуацию и фокусирует на выразительных возможностях, а не на защите программистов от самих себя. Теперь давайте рассмотрим другие структурные типы Unicon, которые часто используются для обеспечения связей между типами различных классов. Первый встроенный структурный тип, который мы рассмотрим, – это тип списка.

Списки

Я представил классы перед списками, чтобы немного подразнить вас. Списки являются наиболее распространенным структурным типом. В этой книге показано лишь небольшое представление о том, на что способны списки. Списки Unicon – это нечто среднее между связанным списком и массивом, которое может увеличиваться и уменьшаться, может использоваться как стек, очередь или двусторонняя очередь. Тип списка поддерживает различные представления для массивов целых чисел и массивов действительных чисел, которые оптимизируют их пространственное представление, совместимое с языком C.

Помимо использования в качестве массивов, стеков и т. п., списки обычно используются в качестве «клеящих» структур данных в классах для реализации агрегации и множественности. Одним из немногих предупреждений для обычных программистов о списках Unicon является то, что их первый индекс равен 1, а не 0. Давайте сравним тип списка с удивительно полезным типом данных – таблицей.

Таблицы

Таблица, которую иногда называют **ассоциативным массивом**, является чрезвычайно гибкой структурой, которая отображает индексы произвольных типов для значений произвольных типов. Таблица называется в честь ее реализации, которая обычно представляет собой хеш-таблицу. Таблица похожа на массив, индексы которого не ограничены непрерывными целыми числами, начинающимися с 1. Ключи таблицы могут быть несмежными, разреженными, отрицательными, а также могут быть строками или любыми другими типами. Строки и целые числа – практически единственные типы, которые используются в качестве хеш-ключей. Конечно, вы можете использовать вещественные числа, но ошибки округления делают последующий поиск сложным. И вы можете использовать csets в качестве ключей таблицы, просто это редкость. Если вы используете значения других структур в качестве ключей таблицы, все работает, но вы не можете вычислить их хеш из их содержимого, потому что содержимое является изменяемым.

Файлы

Тип файлов Unicon – это то, чего вы ожидаете. Файлы обычно имеют доступ к постоянному хранилищу, управляемому операционной системой. Существуют удобные функции для обработки строк за один раз. Большинство форм ввода и вывода являются расширениями типа файла, поэтому функции файла применяются к сетевым соединениям, графическим окнам и так далее.

Другие типы

Unicon имеет множество других мощных встроенных типов для таких вещей, как окна, сетевые соединения и потоки. В отличие от некоторых языков, в нем нет глобальной блокировки интерпретатора, замедляющей параллелизм потоков. Учитывая значения в этом богатом наборе типов данных, тела программ Unicon собираются в вычисления с помощью различных выражений.

Оценка выражений

Выражения Unicon ориентированы на достижение цели. Когда они могут, они вычисляют результат, и это называется **успехом**. Выражения, которые не имеют результата, называются **неудачными**. Неудача обычно предотвращает выполнение окружающего выражения и может вызвать возвращение к более ранней части выражения, если есть такая часть, которая может дать дополнительные результаты.

Эта семантика оценки, ориентированной на цель, устраняет необходимость в булевом типе данных, который обычно встречается в других языках. Это также значительно увеличивает выразительные возможности языка, устраняя множество утомительных проверок на наличие сторожевых значений или написания явных циклов для поиска вещей, которые могут быть найдены путем оценки, ориентированной на цель, и переборов с возвратом. Требуется время, чтобы привыкнуть к этой возможности, но после ее освоения код становится короче и быстрее в написании.

Формирование базовых выражений с помощью операторов

Многие из операторов Unicon знакомы вам по другим языкам, в то время как иные являются уникальными. Ниже приводится краткое описание операторов Unicon. При объединении в цепочку порядок выполнения операторов определяется их приоритетом, который в целом такой же, как и в основных языках. Унарные операторы имеют более высокий приоритет, чем бинарные, умножение происходит перед сложением, и так далее. В случае сомнений установите приоритет с помощью круглых скобок.

Принудительный приоритет:

(*exp* **)**

Круглые скобки, перед которыми нет выражения, усиливают приоритет оператора и в противном случае не имеют никакого эффекта.

Размер:

***** *x* : int

Унарная звездочка – это оператор размера, который возвращает количество элементов в строке, cset, очереди или структуре *x*.

Is-null:

/ *x*

Is-nonnull:

**** *x*

Эти предикаты просто производят *x*, если тест истинен, но если тест ложен, то они терпят неудачу.

Инверсия:

- *num*

Унарный плюс:

+ *num*

Инверсия числа – это изменение его знака с положительного на отрицательный или наоборот. Унарный оператор плюс заставляет операнд стать числом, но не изменяет его числовое значение.

Отрицание результата оценки:

not *exp*

not преобразует успех выражения *exp* в неудачу, и наоборот. Когда завершается успешно, результатом является нулевое значение.

Tabmat:

= *str*

Когда операндом является строка, унарное равенство соответствует *tab(match(s))*.

Двоичная арифметика:

num1 **^** *num2*

num1 **%** *num2*

num1 ***** *num2 num1* **+** *num2*

num1 **/** *num2 num1* **–** *num2*

Сразу за обычными двоичными числовыми операторами, включая каретку для возведения в степень, может следовать `:=` для выполнения расширенного присваивания, например `x +:= 1`, чтобы прибавить единицу к `x`. Почти все двоичные операторы могут использоваться с `:=` для выполнения расширенного присваивания.

Конкатенация:

str1 **||** *str2*

Конкатенация списков:

lst1 **|||** *lst2*

Конкатенация – это соединение первого и второго операндов по порядку и получение результата.

Присваивание значения:

variable **:=** *expr*

При присваивании значение справа хранится в переменной слева.

Сравнение:

num1 **=** *num2*

str1 **==** *str2num1* **~=** *num2*

str1 **~==** *str2num1* **<** *num2*

str1 **<<** *str2num1* **<=** *num2*

str1 **<<=** *str2num1* **>** *num2*

str1 **>>** *str2num1* **>=** *num2*

str1 **>>=** *str2ex1* **===** *ex2*

ex1 **~===** *ex2*

Предоставляются обычные числовые операторы сравнения, а также строковые версии, которые обычно повторяют символ оператора. Тильда означает НЕ. Оператор эквивалентности === и оператор неэквивалентности ~=== не выполняют преобразования типов, в то время как остальные операторы обычно приводят операнды к числовому или строковому типу по мере необходимости. Операторы сравнения возвращают свой второй операнд, если только они не ошибаются.

И:

ex1 **&** *ex2*

Бинарный оператор амперсанд проверяет *ex1*, и если оно успешно, то результатом всего выражения является результат *ex2*. Если *ex1* не удается, *ex2* не оценивается.

Создание пустого списка:

[ ]

Создание инициализированного списка:

[*ex* [, *ex*]* **]**

Создание списка результатов выражения:

[: *ex* **:]**

Создание инициализированной таблицы:

[*ex* **:** *ex* [**;** *ex* **:** *ex*]* **]**

Когда в скобки заключены ноль или более элементов, создаются списки и таблицы. Элементы инициализатора разделяются запятыми. Элементы таблицы состоят из **пар ключ-значение.**

Выбор подэлемента:

ex1 **[** *ex2* [**,** ex]* **]**

Срез:

ex1 **[** *ex2* **:** *ex3* **]**

Плюсовой срез:

ex1 **[** *ex2* **+:** *ex3* **]**

Минусовой срез:

ex1 **[** *ex2* **-:** *ex3*]

Для списков и строк, когда скобки имеют выражение слева от них, берется элемент или фрагмент из этого выражения. Выражение L[1,2] эквивалентно L[1][2]. Ссылка на регулярный элемент выбирает элемент из значения, такого как строка или список. Элемент может быть прочитан и использован в окружающем выражении или записан и заменен присваиванием. Индексы обычно начинаются с 1 для первого элемента. Индексы списков и строк не работают при выходе за пределы диапазона. Срез определен как для списков, так и для строк. Срез строки может быть проведен, если исходная строка является переменной. Срез списка создает список, который содержит копию выбранных элементов базового списка.

Индексы таблиц являются ключами и могут быть любого типа. При поиске неизвестного ключа индексы таблиц дают значение таблицы по умолчанию. Записи принимают как строки, так и целочисленные индексы, как если бы они были и таблицами, и списками.

Доступ к полю:

x **.** *name*

Оператор точка (.) выбирает поле *name* из записи или экземпляра класса *x*.

Вызов процедур, функций и методов

Одна из самых фундаментальных абстракций во всем программировании – это действие, заключающееся в том, чтобы попросить другую часть кода в другом месте вычислить значение, которое требуется в выражении.

В Unicon вы можете обратиться или вызвать написанную пользователем процедуру, встроенную функцию или метод класса, заключив в круглые скобки ее имя или ссылку, заключив в них ноль или более значений.

Вызов:

f ([*expr1* [**,** [$expr_i$]]*])

Вызов метода:

object **.** *method* ([*expr1* [**,** [$expr_i$]]*])

Процедура или функция вызывается с помощью круглых скобок, заключающих ноль или более выражений аргументов, разделенных запятыми. Пропуск аргумента приводит к передаче нулевого значения в этой позиции. Выполнение переходит к этой процедуре или функции и возвращается обратно, когда получен результат или результат невозможен. Метод вызывается путем обращения к имени метода через объект.

Завершение вызова:

`return` [*expr*]

return выдает *expr* в качестве результата метода или процедуры. Вызов не может быть возобновлен. Если результат *expr* не получен, выражение возвращает *null*.

Выдача результата:

`suspend` [*expr*]

suspend выдает *expr* в качестве результата метода или процедуры. Вызов будет возобновлен, чтобы попытаться получить другой результат, если выражение, в котором был сделан вызов, терпит неудачу. Если результат *expr* не получен, выражение возвращает *null*.

Завершение вызова без результата:

```
fail
```

fail завершает вызов процедуры или метода без результата. Вызов не может быть возобновлен.

Итерации и выбор того, что и как выполнять

Несколько структур управления Unicon охватывают традиционные операции потока управления. К ним относятся последовательность, циклы и выбор фрагментов кода для выполнения.

Последовательное выполнение:

{ *expr1* **;** *expr2* }

Фигурные скобки обозначают выражения, которые должны быть вычислены последовательно. Точки с запятой завершают каждое выражение в последовательности. В Unicon предусмотрена автоматическая вставка запятой, поэтому точка с запятой нужна редко, за исключением случаев, когда два или более выражений находятся в одной строке.

If-then:

if *ex1* **then** *ex2* [**else** *ex3*]

При успехе *ex1* выполняется *ex2*, в противном случае оценивается *ex3*.

Оценка до тех пор, пока не произойдет неудача:

while *ex1* [**do** *ex2*].

Цикл *while* выполняет итерации до тех пор, пока *ex1* не завершится неудачей.

Уничтожение генератора:

every *ex1* [**do** *ex2*]

Цикл *every* просто терпит неудачу, несмотря ни на что. Это заставляет получить все результаты из *ex1*. Эта штука пожирает генераторы.

Тело цикла:

do *ex*

do обычно является необязательным и предоставляет тело для выполнения итераций цикла.

Оценка до бесконечности:

repeat *ex*

Выражение *repeat* – это цикл, который переоценивает *ex* снова и снова. Помимо прочего, *ex* может выйти из цикла, используя *break*, *return*, *fail* или останавливая выполнение программы.

Выход из цикла:

break [*ex*]

break завершает цикл в текущей процедуре или методе – всегда ближайшем. Выражение *ex* оценивается после завершения цикла. Вы можете написать *break break* для выхода из двух циклов, *break break break* для выхода из трех циклов и так далее.

Сканирование строк:

str **?** *ex*

Эта структура управления выполняет *ex*, проверяя наличие *&subject* в *str*. Ключевое слово *&pos* начинается с 1. Сканирование строк может быть вложенным. Оно имеет динамическую область видимости.

Выполнение одной ветви:

case *ex* **of {** [*ex1* **:** *ex2*] * ; [**default :** exN] **}**

case оценивает выражение и сравнивает результат с последовательностью ветвей случая, проверяемых по порядку. Если выражение равно определению ===, то есть без преобразования типов, одному из выражений слева от двоеточия, то выполняется выражение справа от двоеточия и случай завершается.

Запуск по первому вызову:

initial *ex*

initial оценивает выражение в начале процедуры или метода, но только когда эта процедура или метод будут вызваны в первый раз.

Генераторы

Некоторые выражения в Unicon могут давать несколько результатов. Особенность генераторов заключается в том, что если генератор возобновляется для получения второго или последующих результатов, окружающее выражение может быть выполнено повторно и в результате может выдать несколько результатов для вложенного в него выражения. Например, рассмотрим вызов *ord("="|"+"|"|"-")*. Функция *ord(s)*, которая возвращает ASCII-код для *s*, не является генератором, но если ее параметрическое выражение является генератором, то все выражение *ord()* является генератором. В данном случае *"="|"+"|"|"-"* является генератором, который может дать три результата. Если вложенному выражению нужны все из них, *ord()* может быть вызван три раза и выдать три результата для вложенного выражения. В качестве другого примера этой очень хорошей особенности рассмотрим следующее выражение:

```
\kids[1|2].first | genlabel()
```

Этот генератор может создавать поле .first либо из kids[1], либо из kids[2], при условии что kids не является null или пустым, но если эти значения не встречаются или не удовлетворяют окружающему выражению, это выражение вызовет genlabel() и выдаст свой результат(ы), если таковой имеется.

Альтернатива:

ex1 **|** *ex2*

Альтернатива генерирует результаты сначала из *ex1*, а затем из *ex2*.

Генерация компонентов:

! *ex* : *any**

Унарный оператор восклицания порождает составные части значения в определенном порядке. Целые числа генерируются путем подсчета от 1 до числа. Строки, csets, списки, записи и объекты создаются путем получения их значений по одному за раз по порядку. Таблицы и наборы ведут себя аналогично, но порядок не определен. Файлы генерируют свое содержимое по строке за раз.

Конечная числовая последовательность:

ex1 **to** *ex2* [**by** *ex3*]

to генерирует числа от *ex1* до *ex2*. Шаг по умолчанию равен 1, но если указано значение *by*, то последовательность будет увеличиваться на эту величину каждый раз.

Отладка и вопросы окружения

Этот раздел содержит информацию, которая может оказаться полезной при программировании в Unicon. Сюда входят краткое введение в отладчик Unicon, некоторые переменные окружения, которые вы можете установить для изменения поведения Unicon во время работы программы, и простой препроцессор, предоставляемый Unicon.

Изучение основ отладчика UDB

Отладчик Unicon на уровне исходного кода называется *udb* и описан в UTR 10, с которым можно ознакомиться по адресу *http://unicon.org/utr/utr10.html*. Набор команд *udb* основан на наборе команд *gdb*, который находится по адресу *https://www.gnu.org/software/gdb/*.

Когда вы запускаете *udb*, то указываете программу для отладки в качестве аргумента командной строки. Также из отладчика можно выполнить команду *load*, чтобы указать программу для отладки. Выход из отладчика обычно осуществляется с помощью команды *quit* (или *q*).

Подсказка *udb* распознает множество команд, часто в сокращенном виде. Возможно, после команды *quit* следующей по важности командой является *help* (или *h*).

Следующей по важности командой является команда *run* (или *r*). Она может быть использована для перезапуска выполнения программы с начала.

Чтобы установить точку прерывания на номер строки или процедуры, вы можете использовать команду *break* (или *b*) с указанием номера строки или имени процедуры. Когда выполнение достигнет этого места, вы вернетесь в командную строку *udb*. В этот момент вы можете использовать команду *step*

(или *s*) для выполнения одной строки за раз, *next* (или *n*) для перехода к следующей строке, пропуская при этом любые вызванные процедуры или методы, *print* (или *p*) для получения значений переменных или *cont* (или *c*) для продолжения выполнения на полной скорости.

Переменные окружения

Несколько переменных окружения управляют или изменяют поведение программ Unicon или компилятора Unicon. Здесь приведены наиболее важные из них. По умолчанию куча области блоков и куча области строк Unicon имеют размер, пропорциональный физической памяти, но вы можете явно задать некоторые параметры памяти среды выполнения, пользуясь табл. A.2.

Таблица A.2. Переменные окружения и их описания

Переменная окружения	Описание
BLKSIZE	Байтов в куче области блоков
IPATH	Список каталогов для поиска ссылок
LPATH	Список каталогов для поиска includes
MSTKSIZE	Байтов в основном стеке
STKSIZE	Байтов в стеках совместных выражений
STRSIZE	Байтов в куче области строк
TRACE	Начальное значение *&trace*

IPATH также используется для поиска суперклассов и импорта пакетов. Теперь давайте рассмотрим препроцессор Unicon, который немного похож на упрощенный препроцессор C.

Препроцессор

Препроцессор Unicon выполняет включение файлов и заменяет символьные константы значениями. Цель препроцессора – позволить включать или отключать фрагменты кода во время компиляции. Это облегчает, например, создание различных кодов для разных операционных систем.

Команды препроцессора

Следующие директивы препроцессора – это строки, начинающиеся со знака доллара:

- `$define` *sym text*
  Символ *sym* заменяется на *text*. В этой конструкции нет макропараметров.
- `$include` *filenam*
  Файл с именем *filenam* включается в исходный код, в котором был найден `$include`.
- `$ifdef` *sym*

`$ifndef` *sym*
`$else`
`$endif`

Строки внутри `$ifdef` передаются компилятору, если *sym* был представлен предыдущим `$define`. `$ifndef` передает исходный код, если символ не был определен. Эти две директивы принимают необязательный `$else`, за которым следует дополнительный код, и завершаются `$endif`.

- `$line` *num* [*filenam*]
  Следующая строка должна быть представлена как начинающаяся со строки *num* из файла *filenam*.
- `$undef` *sym*
  Определение *sym* стирается. Последующие вхождения ничем не заменяются.

Определения встроенных макросов

Эти символы определяют плаформы и функции, которые могут присутствовать и влиять на возможности языка. Определения встроенных макросов приведены в табл. А.3.

Таблица А.3. Встроенные макросы

Макрос	Содержание	Макрос	Содержание
`_CO_EXPRESSIONS`	синхронные потоки	`–MESSAGING`	HTTP, SMTP и т. д.
`_CONSOLE WINDOW`	эмулированный терминал	`_MS WINDOWS`	Microsoft Windows
`_DBM`	DBM	`_MULTITASKING`	load() и т. д.
`_DYNAMTC_LOADING`	код может быть загружен	`_POSIX`	POSIX
`_EVENT_MONITOR`	код обрабатывется	`__PIPES`	однонаправленная передача
`_GRAPHICS`	графика	`SYSTEMFUNCTION`	система ()
`_KEYBOARD_FUNCTIONS`	`kbhit (), getc ()` и т. д.	`_UNIX`	UNIX, Linux, …
`_LARGE INTEGERS`	произвольная точность	`_WIN32`	графика Win32
`_MACINTOSH`	Macintosh	`_X_WINDOW-SYSTEM`	графика X Windows

Эти символы, которые вы можете проверить во время компиляции с помощью *$ifdef*, имеют соответствующие строки функций, которые можно проверить во время выполнения с помощью *&features*. Более подробную информацию вы можете посмотреть в книге *Programming with Unicon*. Теперь давайте рассмотрим встроенные функции Unicon.

Мини-справочник функций

В этом разделе описывается подмножество встроенных функций Unicon, которые, вероятно, являются актуальными для разработчиков языков программирования. Полный список см. в *Appendix A* к книге *Programming with Unicon*. Необходимые типы параметров в этом разделе указаны по их именам. Имена *c* или *cs* указывают на *набор символов*. Имена *s* или *str* обозначают *строку*. Имена *i* или *j* обозначают *целые числа*. Имя *x* или *any* указывает на то, что *параметр* может быть любого типа. Такие имена могут быть дополнены номером, чтобы отличить их от других параметров того же типа. Двоеточие и типы после параметров указывают на *возвращаемые типы*, а также количество *возвращаемых значений*. Как правило, функция имеет ровно одно возвращаемое значение. Знак вопроса указывает на то, что функция является *предикатом*, который может завершиться неудачно с нулем или одним возвращаемым значением. Звездочка указывает на то, что функция является *генератором* с нулем или более возвращаемых значений.

Многие функции также имеют значения по умолчанию для параметров, которые указываются в ссылке с помощью двоеточия и значения после их имени. Функции с параметрами, заканчивающимися на `s`, `i` и `j`, являются функциями анализа строк. Последние три параметра функций анализа строк по умолчанию имеют значения `&subject`, `&pos` и `0`. Параметры `i` и `j` меняются местами, если `i` больше `j`, поэтому не имеет значения, в каком порядке заданы индексы, и анализ всегда будет проводиться слева направо.

- `abs(n) : num`
  `abs(n)` возвращает `-n`, если `n` отрицательно. В противном случае возвращается `n`.
- `any(cs, s, i, j) : integer?`
  `any(cs,s,i,j)` выдает `i+1`, если `s[i]` является членом `cset`, `cs`, и завершается неудачей в противном случае.
- `bal(c1:&cset, c2:'(', c3:')', s, i, j) : integer*`
  `bal(c1,c2,c3,str,i1,i2)` производит индексы в `str`, где член `c1` в `str[i:j]` имеет те же начальные символы, что `c2`, и те же конечные символы, что `c3`.
- `char(i) : str`
  `char(i)` возвращает однобуквенную строку кодировки `i`.
- `close(f) : file`
  `close(f)` освобождает ресурсы операционной системы, связанные с файлом `f`, и закрывает его.
- `copy(any) : any`
  `copy(y)` создает `y`. Для структур возвращает физическую копию. Для вложенных структур копия находится на один уровень глубже.
- `delay(i) : null`
  `delay(i)` ожидает не менее указанного количества миллисекунд.
- `delete(y1, y2, ...) : y1`
  `delete(y1,y2)` удаляет значения на месте ключа `y2` и последующие элементы структуры `y1`.

- `exit(i:0) :`
  `exit(i)` завершает выполнение программы и выдает `i` в качестве статуса выхода.
- `find(str1, str2, i, j) : int*`
  `find(str1,str2,i1,i2)` выдает индексы, в которых `str1` встречается в `str2`, учитывая только индексы между `i1` и `i2`.
- `getenv(str) : str?`
  `getenv(str1)` выдает значение с именем `str1` из окружения.
- `iand(i1, i2) : int`
  `iand(i1,i2)` возвращает `i1`, объединенное побитовым И с `i2`.
- `icom(i) : int`
  `icom(i)` превращает единицы в нули, а нули в единицы.
- `image(x) : str`
  `image(x)` выдает строку, представляющую содержимое `x`.
- `insert(x1, x2, x3:&null) : x1`
  `insert(x1,x2,x3)` помещает `x2` в структуру `x1`. Если `x1` – список, то `x2` – позиция, в противном случае это ключ. Если `x1` – таблица, ключ `x2` связан со значением `x3`. `insert()` создает структуру.
- `integer(x) : int?`
  `integer(x)` преобразует `x` к целочисленному типу. Терпит неудачу, если преобразование невозможно.
- `ior(i1, i2) : int`
  `ior(i1,i2)` возвращает `i1`, объединенное побитовым ИЛИ с `i2`.
- `ishift(i1, i2) : int`
  `ishift(i1,i2)` сдвигает битовые позиции `i2` в пределах `i1` и возвращает результат. Сдвиг происходит вправо, если `i2 < 0`, или влево, если `i2 > 0`. Нули `i2` поступают в направлении, противоположном направлению сдвига.
- `ixor(i1, i2) : int`
- `ixor(i1,i2)` возвращает `i1`, объединенное побитовым исключающим ИЛИ с `i2`.
- `kbhit() : ?`
- `kbhit()` возвращает, было нажатие клавиатуры или нет.
- `key(y) : any*`
- `key(y)` производит ключи/индексы, с помощью которых можно получить доступ к элементам структуры `y`.
- `list(i,x) : list`
- `list(i,x)` создает список с `x` элементами, каждый из которых содержит `x`. `x` не копируется для каждого элемента списка, поэтому вам, возможно, придется выделить их все отдельно, если вы хотите, например, получить список списков.
- `many(cs,str,i,j) : int?`
- `many(cs,str,i,j)` выдает позицию в `str`, которая следует за столькими смежными членами `cs` в пределах `str[i:j]`, насколько это возможно.

- `map(str1,str2,str3) : str`
  `map(str1,str2,str3)` возвращает `str1`, преобразованную таким образом, что там, где символы `str1` могут быть найдены в `str2`, они заменяются соответствующими символами в `str3`. `str2` и `str3` должны быть одинаковой длины.
- `match(str, s, i, j) : int?`
  `match(str1,s,i,j)` возвращает `i+*str1`, если `str1==s[i+:*str1]`. Функция терпит неудачу, если совпадения нет.
- `max(num,...) : num`
  `max(...)` выдает числовой максимум своих параметров.
- `member(y,...) : y?`
  `member(y,...)` выдает `y`, если остальные параметры находятся в `y`; в противном случае – неудача.
- `min(num,...) : num`
  `min(...)` выдает числовой минимум своих параметров.
- `move(i) : str`
  `move(i)` увеличивает или уменьшает `&pos` на `i` и возвращает подстроку от старой до новой позиции в пределах `&subject`. Позиция сбрасывается, если эта функция возобновляется.
- `open(str1, str2, ...) : file?`
  `open(str1,str2,...)` просит операционную систему открыть имя файла `str1`, используя режим `str2`. Последующие аргументы являются атрибутами, которые могут влиять на определенные файлы. Распознаваемые функцией режимы, которые указываются в аргументе `str2`, приведены в табл. А.4.

Таблица А.4. Режимы и их описания

Режим	Описание	Режим	Описание
`a`	добавлять/дополнять	`nl`	прослушивать TCP-порт
`b`	открывать для чтения и записи	`nu`	подключаться к порту UDP
`c`	создать новый файл	`m`	подключение к серверу обмена сообщениями
`d`	база данных GDBM	`o`	соединение ODBC (SQL)
`g`	окно 2D-графики	`p`	выполнить командную строку и передать ее
`g1`	окно 3D-графики	`r`	читать
`n`	TCP-клиент	`t`	переводить новые строки
`na`	принимать TCP-соединение	`u`	использовать двоичный непереведенный режим
`nau`	принимать UDP-дейтаграммы	`w`	записывать

- `ord(s) : integer`
  `ord(s)` возвращает порядковый номер (например, код ASCII) однобуквенной строки `s`.
- `pop(L) : any?`
  `pop(L)` возвращает значение из начала `L` и удаляет его из списка.
- `pos(i) : int?`
  `pos(i)` возвращает, находится ли сканирование строки в позиции `i`.
- `proc(any, i:1) : procedure?`
  `proc(str,i)` производит процедуру, которая называется `s`. Если `i` равно 0, то производится встроенная функция с именем `s`, если таковая существует.
- `pull(L, i:1) : any?`
  `pull(L)` возвращает последний элемент `L` и удаляет его. Может быть удалено `i` элементов.
- `push(L, y, ...) : list`
  `push(L,y1,...,yN)` вставляет один или несколько элементов в список `L` спереди. `push()` возвращает свой первый параметр с добавленными новыми значениями.
- `read(f:&input) : str?`
  `read(f)` вводит следующую строку `f` и возвращает ее без новой строки.
- `reads(f:&input, i:1) : str?`
  `reads(f,i)` вводит *i* байт из файла `f` и терпит неудачу, если больше нет байт. `reads()` возвращает доступный вход, даже если он меньше `i` байт. Если запрошено –1 байт, `reads()` возвращает строку, содержащую все оставшиеся байты в файле.
- `ready(f:&input, i:0) : str?`
  `ready(f,i)` вводит `i` байт из файла `f`, обычно через сетевое соединение. Возврат производится без блокировки, и если это означает, что доступно меньше `i` байт, то так тому и быть. Терпит неудачу, если входные данные еще не поступили.
- `real(any) : real?`
- `real(x)` преобразует `x` в его эквивалент с плавающей точкой. Терпит неудачу, если преобразование невозможно.
- `remove(str) : ?`
  `remove(str)` удаляет файл с именем `str` из файловой системы.
- `rename(str1, str2) : ?`
  `rename(str1,str2)` изменяет имя файла `str1` на `str2`.
- `repl(y, i) : x`
  `repl(x,i)` создает `i` конкатенированных экземпляров `x`.
- `reverse(y) : y`
  `reverse(y)` выдает список или строку, расположенную в обратном порядке по отношению к `y`.
- `rmdir(str) : ?`
  `rmdir(str)` удаляет папку с именем `str` или терпит неудачу, если ее невозможно удалить.

- `serial(y) : int?`
  `serial(y)` выдает идентифицирующее целое число для структуры `y`. Эти номера присваиваются при выделении структур. Отдельные счетчики используются для каждого типа структуры. Идентифицирующее целое число обеспечивает хронологический порядок выделения экземпляров каждого типа.
- `set(y, ...) : set`
  `set()` выделяет набор. Параметры являются начальными значениями нового набора, за исключением случаев, когда они являются списками. В этом случае содержимое параметров является начальными значениями нового множества.
- `sort(y, i:1) : list`
  `sort()` выделяет список, в котором сортируются элементы *y*. Когда сортируются таблицы, ключи сортируются, когда `i` равно одному или трем, а значения сортируются, когда `i` равно двум или четырем. Когда `i` равно одному или двум, элементами возвращаемого списка являются двухэлементные подсписки ключ-значение. Когда *i* равно трем или четырем, элементы возвращаемого списка чередуются между ключами и значениями.
- `stat(f) : record?`
  `stat(f)` выдает информацию об `f`. Аргументом может быть строковое имя файла или открытый файл. Три переносимых поля – размер `size` в байтах, разрешения доступа `mode` и время последнего изменения `mtime`. Строка режима напоминает длинный листинг из команды `ls(1)`. `stat(f)` терпит неудачу, если нет имени файла или пути *f*.
- `stop(s, ...) :`
  `stop(args)` записывает свои аргументы в `&errout`, за которым следует новая строка, а затем выходит из программы.
- `string(any) : str?`
  `string(y)` преобразует *y* в соответствующую строку. Она терпит неудачу, если преобразование невозможно.
- `system(x, f:&input, f:&output, f:&errout, s) : int`
  `system(x)` запускает программу, заданную в виде строковой командной строки или списка аргументов командной строки. Программа запускается как отдельный процесс. Необязательные аргументы предоставляют стандартные файлы ввода-вывода. Возвращается статус завершения процесса. Если пятым параметром является «*nowait*», функция немедленно возвращается с новым идентификатором процесса, вместо того чтобы ждать его завершения.
- `tab(i:0) : str?`
  `tab(i)` присваивает местоположению `i` значение *&pos*. В результате получается подстрока между новым и прежним расположением. Ключевое слово `&pos` возвращается в прежнее положение, если функция возобновляется.
- `table(k,v, ..., x) : table`
  `table(x)` строит таблицу, значения которой по умолчанию равны `x`. `table(k,v,...x)` инициализирует таблицу из чередующихся аргументов ключа и значений.

- `trim(str, cs:' ', i:-1) : str`
  `trim(str,cs,i)` выдает подстроку `str` с удаленными членами `cset`, `cs` спереди (когда `i` = 1), сзади (когда `i` = –1) или с обеих сторон (когда `i` = 0). По умолчанию удаляются пробелы в конце.
- `type(x) : str`
  `type(x)` выдает тип `x` в виде строки.
- `upto(cs, str, i, j) : int*`
  `upto(cs,str, i,j)` генерирует индексы в `str`, где член `cset`, `cs` может быть найден в `str[i:j]`. В противном случае терпит неудачу.
- `write(s|f, ...) : str|file`
  `write(...)` отправляет один или несколько строковых аргументов, дополненных новой строкой, в файл, по умолчанию в `&output`. `write()` выдает последний параметр.
- `writes(s|f, ...) : str|file`
  `writes(...)` отправляет один или несколько строковых аргументов в файл, по умолчанию в `&output`. `writes()` выдает последний параметр.

Избранные ключевые слова

Unicon имеет около 75 ключевых слов. Ключевые слова – это глобальные имена, начинающиеся с амперсанда, с заранее определенным значением. Многие ключевые слова являются встроенными в язык постоянными значениями, в то время как другие связаны со встроенными средствами языка, специфичными для конкретной области, такими как сканирование строк или графика. В этом разделе перечислены наиболее важные ключевые слова, многие из которых встречаются в примерах, приведенных в этой книге:

- `&clock : str`
  Ключевое слово `&clock`, доступное только для чтения, выдает текущее время суток.
- `&cset : cset`
  Постоянное ключевое слово `&cset` обозначает cset, содержащий все.
- `&date : str`
  Ключевое слово `&date`, доступное только для чтения, выводит текущую дату.
- `&digits : cset`
  Постоянное ключевое слово `&digits` обозначает cset, содержащий от 0 до 9.
- `&errout : file`
  Ключевое слово `&errout` обозначает стандартное местоположение для вывода ошибок.
- `&fail`
  Ключевое слово `&fail` – это выражение, которое терпит неудачу при создании результата.
- `&features : str*`
  Ключевое слово `&features`, доступное только для чтения, выдает то, что среда выполнения Unicon может делать в виде строк. Например, если Unicon построен с графическими средствами, то они будут обобщены.

- `&input : file`
  Постоянное ключевое слово `&input` обозначает стандартное местоположение для ввода.
- `&lcase : cset`
  Ключевое слово `&lcase` обозначает cset, содержащий буквы от a до z.
- `&letters : cset`
  Ключевое слово `&letters` обозначает cset, содержащий буквы от A до Z и от a до z.
- `&now : int`
  Ключевое слово `&now`, доступное только для чтения, выдает секунды с 1/1/1970 GMT.
- `&null : null`
  Ключевое слово `&null` обозначает значение, которое не является никаким другим типом. Оно является значением по умолчанию во многих конструкциях языка для объектов, которые не были инициализированы или были опущены.
- `&output : file`
  Ключевое слово `&output` обозначает стандартное местоположение для вывода.
- `&pos := int`
  Ключевое слово *&pos* обозначает позицию внутри `&subject`, в которой выполняется анализ строки. Оно начинается с 1 в каждой среде сканирования строк, и его значение всегда является допустимым индексом в `&subject`.
- `&subject := str`
  Ключевое слово `&subject` относится к анализируемой строке в структуре управления сканированием строк.
- `&ucase : cset`
  Постоянное ключевое слово `&ucase` обозначает cset, содержащий буквы от A до Z.
- `&version : str`
  Ключевое слово `&version` сообщает версию Unicon в виде строки.

Оценки

Глава 1

1. Сгенерировать код на языке C гораздо проще, чем машинный код, но полученный код может быть больше или медленнее, чем собственный код, а транспайлер зависит от базового компилятора, который может быть чем-то вроде движущейся цели.
2. Лексический, синтаксический и семантический анализ, за которым следует генерация промежуточного и окончательного кода.
3. Классические болевые точки включают в себя чрезмерную сложность ввода/вывода, особенно на новых аппаратных средствах, параллельное выполнение и обеспечение работы программы в различных операционных системах и процессорах. Одной из особенностей, которую языки использовали для упрощения ввода/вывода, было уменьшение проблемы взаимодействия с новым оборудованием через набор строк в человекочитаемых форматах, например для воспроизведения музыки или чтения сенсорного ввода. Параллельное выполнение было упрощено в языках со встроенными потоками и мониторами. Переносимость была упрощена в языках, которые предоставляют свою собственную высокоуровневую реализацию виртуальной машины.
4. Это зависит от интересующей вас области применения, вот одна из них. Язык будет вводить программы, написанные на Java-подобном синтаксисе, хранящиеся в файлах с расширением *.j0*, и генерировать целевой код, запускаемый на веб-сайтах, в виде HTML5+JavaScript. Язык будет поддерживать JDBC и сокетные коммуникации через веб-сокеты, а также 2D- и 3D-графику с помощью OpenGL. Язык будет поддерживать интуитивно понятный синтаксис квадратных скобок для доступа к элементам строк и ключам HashMap. Язык будет поддерживать синтаксис JSON в исходном коде в виде литерала HashMap.

Глава 2

1. Зарезервированные слова способствуют как удобочитаемости для человека, так и простоте разбора для реализации языка, но они также иногда исключают возможность использования наиболее естественных имен для переменных в программе. Слишком большое количество зарезервированных слов может усложнить изучение языка программирования.
2. Целые числа в C или Java, например, могут быть знаковыми или беззнаковыми, в десятичном, восьмеричном, шестнадцатеричном или даже двоичном формате, в виде малых, средних, больших или сверхразмерных слов.

3. Некоторые языки реализуют механизм вставки точки с запятой, который делает точку с запятой необязательной. Часто это включает использование символа новой строки вместо точки с запятой для завершения или разделения высказываний.
4. Хотя большинство программ Java не используют эту возможность, размещение *main()* в нескольких (или во всех) классах может быть очень полезно при модульном и интеграционном тестировании.
5. Хотя вполне возможно предоставить предварительно открытые средства ввода/вывода, они могут потребовать значительных ресурсов и затрат на инициализацию, которые программы не должны затрачивать, если данное средство ввода/вывода не будет использоваться в программе. Если вы разрабатываете язык, который специально нацелен на область, где гарантирована одна из этих форм ввода/вывода, имеет смысл подумать о том, как сделать доступ к ней как можно более простым.

Глава 3

1. В первом приближении регулярное выражение имеет вид [0-3][0-9]"/"[01][0-9]"/"[0-9]{4}. Хотя можно написать регулярное выражение, которое соответствует только законным датам, такое выражение непрактично длинное, особенно учитывая високосные годы. В подобных случаях имеет смысл использовать регулярное выражение, которое обеспечивает наиболее простое приближение к корректности, а затем проверить корректность в семантическом действии или на последующем этапе семантического анализа.
2. *yylex()* возвращает целочисленную категорию для использования в синтаксическом анализе, в то время как *yytext* является строкой, содержащей совпадающие символы, а *yylval* содержит объект, называемый токеном, который содержит все лексические атрибуты данной лексемы.
3. Когда регулярное выражение не возвращает значение, символы, которые оно сопоставило, отбрасываются, и функция *yylex()* продолжает работу с новым выражением, начиная со следующего символа на входе.
4. Flex сопоставляет самую длинную строку, какую только может, он разрывает связи с несколькими регулярными выражениями, выбирая то из них, которое соответствует самой длинной строке. Когда два регулярных выражения совпадают по длине в данной точке, Flex выбирает то из них, которое встречается первым в файле спецификации lex.

Глава 4

1. Терминальный символ не определяется правилом производства в терминах других символов. Это противоположность нетерминальному символу, который может быть заменен или построен из последовательности символов в правой части правила производства, которое определяет этот нетерминальный символ.
2. Сдвиг удаляет текущий символ из входных данных и помещает его в стек разбора. Сокращение извлекает ноль или более символов из верх-

ней части стека разбора, которые соответствуют правой части правила производства, и помещает соответствующий нетерминал из левой части правила производства на их место.
3. YACC дает вам возможность выполнять некоторый код семантическх действий только тогда, когда выполняется операция сокращения.
4. Целочисленные категории, возвращенные из *yylex()* в предыдущей главе, – это в точности последовательность терминальных символов, которые парсер видит и сдвигает во время разбора. Успешный разбор сдвигает все доступные входные символы и постепенно сводит их к начальному нетерминалу грамматики.

Глава 5

1. Лексический анализатор *yylex()* выделяет лист и сохраняет его в *yylval* для каждого терминального символа, который он возвращает в *yyparse()*.
2. Когда правило производства в грамматике сокращается, код семантического действия в парсере выделяет внутренний узел и инициализирует его дочерние узлы, чтобы они ссылались на листья и внутренние узлы, соответствующие символам в правой части этого правила производства.
3. *yyparse()* поддерживает стек значений, который увеличивается и уменьшается синхронно со стеком разбора во время парсинга. Листья и внутренние узлы хранятся в стеке значений до тех пор, пока они не будут вставлены в качестве дочерних в содержащий их внутренний узел.
4. Стек значений является полностью общим и может содержать значения любого типа. В языке C это делается с помощью типа объединения, который небезопасен для типов. В Java для этого используется класс *parserVal*, который содержит узлы дерева в общем виде. В Unicon и других динамических языках нет необходимости в обертывании или разворачивании.

Глава 6

1. Таблицы символов позволяют на этапах семантического анализа и генерации кода быстро искать символы, объявленные далеко в дереве синтаксиса, следуя правилам области видимости языка.
2. Синтезированные атрибуты вычисляются с использованием информации, расположенной непосредственно в узле, или информации, полученной от его дочерних узлов. Унаследованные атрибуты вычисляются с использованием информации из других мест в дереве, например из родительских или родственных узлов. Синтезированные атрибуты, как правило, вычисляются с помощью восходящего обратного обхода дерева синтаксиса, в то время как унаследованные атрибуты обычно вычисляются с помощью прямого обхода. Оба вида атрибутов хранятся в узлах дерева синтаксиса в переменных, добавленных к типу данных узла.
3. Язык Jzero предусматривает глобальную область видимости, область видимости класса и одну локальную область видимости для каждой функции-члена. Таблицы символов обычно организованы в виде древовидной структуры, соответствующей правилам определения области види-

мости языка, с дочерними таблицами символов, прикрепленными или связанными с соответствующими записями таблицы символов в окружающей области видимости.
4. Если бы Jzero разрешил использовать несколько классов в отдельных файлах, то таблицы символов нуждались бы в механизме, позволяющем знать о таких классах. В Java это может повлечь за собой чтение других исходных файлов во время компиляции данного файла. Это подразумевает, что классы должны легко находиться без ссылки на их имя файла. Отсюда требование Java о том, что классы должны размещаться в файлах, базовое имя которых совпадает с именем класса.

Глава 7

1. Проверка типов обнаруживает множество ошибок, которые могут помешать корректному выполнению программы. Но она также помогает определить, сколько памяти потребуется для хранения переменных и какие именно инструкции понадобятся для выполнения различных операций в программе.
2. Тип структуры необходим для представления произвольно глубоких составных структур, включая рекурсивные структуры, такие как связанные списки. В любой заданной программе имеется лишь конечное число таких типов, поэтому можно было бы перечислить их и представить с помощью целочисленных индексов, поместив их в таблицу типов, но ссылки на структуры обеспечивают более прямое представление.
3. Если бы настоящие компиляторы выводили строку OK для каждой успешной проверки типа, то неигрушечные программы выдавали бы тысячи таких проверок при каждой компиляции, что затрудняло бы фиксацию случайных ошибок.
4. Придирчивые средства проверки типов могут быть неприятны для программистов, но они помогают избежать непреднамеренных преобразований типов, скрывающих логические ошибки, а также уменьшают тенденцию языка к медленной работе из-за молчаливого и автоматического повторного преобразования типов во время выполнения программы.

Глава 8

1. Для доступа к любому массиву результатом оператора индекса будет тип элемента массива. При обращении к структуре или классу имя поля (члена) в структуре должно быть использовано для определения его типа с помощью поиска в таблице символов или чего-то эквивалентного.
2. Тип возврата функции может храниться в таблице символов функции и просматриваться из любого места в теле функции. Один из простых способов сделать это – хранить тип возврата под символом, который не является легальным именем переменной, например return. Альтернативой может быть помещение типа возврата функции в тело функции как наследуемого атрибута. Это может быть относительно просто, но окажется пустой тратой места в узлах дерева разбора.

3. Как правило, такие операторы, как плюс и минус, имеют фиксированное количество операндов и фиксированное число типов, для которых они определены, что позволяет хранить правила проверки типов в таблице или в каком-либо операторе переключателя. Вызовы функций должны выполнять проверку типов над произвольным числом аргументов, которые могут быть произвольного типа. Параметры функции и возвращаемый тип хранятся в записи ее таблицы символов. Они просматриваются и используются для проверки типа в каждом месте, где эта функция вызывается.
4. Помимо доступа к членам, проверка типов происходит, когда создаются, присваиваются, передаются в качестве параметров и, в некоторых языках, уничтожаются составные типы.

Глава 11

1. Сложные наборы инструкций требуют больше времени и логики для декодирования и могут сделать реализацию интерпретатора байт-кода более сложной или менее переносимой. С сдругой стороны, чем ближе окончательный код будет напоминать промежуточный код, тем проще становится этап генерации окончательного кода.
2. Реализация адресов байт-кода с использованием аппаратных адресов обеспечивает наилучшую производительность, на которую вы можете рассчитывать, но это может сделать реализацию более уязвимой к проблемам безопасного хранения в памяти и защищенного кода. Интерпретатор байт-кода, который реализует адреса, используя смещения в массиве байтов, может обнаружить, что у него меньше проблем с памятью, производительность при этом может быть проблемой, а может и не быть.
3. Некоторые интерпретаторы байт-кода могут выиграть от возможности модифицировать код во время выполнения. Например, байт-код, который был связан с использованием информации о смещении байтов, может быть преобразован в код, использующий указатели. Неизменяемый код делает этот тип самомодифицирующегося поведения более сложным или невозможным.

Глава 12

1. Операнды многооперационных инструкций помещаются в стек с помощью инструкций *push*. При выполнении операции вычисляется результат. Результат сохраняется в памяти с помощью инструкции *pop*.
2. Строится таблица, в которой каждая из меток сопоставлена со смещением байта 120. Использование меток, встречающхся после того, как появилась их запись в таблице, просто заменяется значением 120. Использованием меток, встреченных до появления их записи в таблице, являются прямые ссылки. Таблица должна содержать связанный список прямых ссылок, которые сопоставляются при обнаружении метки.
3. В стековой машине байт-кода Jzero операнды могут уже находиться в стеке и PARM-инструкции могут быть избыточными, что позволяет су-

щественную оптимизацию. Кроме того, в машине Jzero последовательность вызовов функций вызывает ссылку на (адрес) вызываемый(ого) метод(а), которая должна быть передана перед операндами. Это совершенно иное соглашение о вызовах, чем то, которое используется в трехадресном промежуточном коде.

4. Статические методы не вызываются на экземпляре объекта. В случае статического метода без параметров вам может потребоваться вставить адрес процедуры в инструкцию *CALL*, поскольку ей не предшествует ни одна инструкция *PARM*.
5. Если вы определите, что ваш трехадресный код для вложенных вызовов действительно приводит к вложенным последовательностям *PARM*... *CALL*, вам понадобится стек инструкций *PARM* для управления ими, и нужно будет тщательно искать корректную инструкцию *CALL*, пропуская любые вложенные инструкции *CALL*, в которых инструкции *PARM* были помещены в стек после инструкции *PARM*, которую вы ищете. Развлекайтесь!

Глава 13

1. В собственном коде есть много новых понятий. К ним относятся многие виды и размеры регистров и режимы доступа к оперативной памяти. Также важен выбор из многих возможных базовых последовательностей инструкций.
2. Даже с учетом необходимых дополнений в процессе выполнения, адреса, которые хранятся как смещения относительно указателя инструкций, могут быть более компактными и использовать преимущества предварительной выборки инструкций в конвейерной архитектуре, чтобы обеспечить более быстрый доступ к глобальным переменным, чем указывать их с помощью абсолютных адресов.
3. Скорость вызова функций важна, поскольку современное программное обеспечение часто состоит из множества нередко вызываемых крошечных функций. Архитектура x64 обеспечивает быстрый вызов функций, если функции используют преимущества передачи первых шести параметров в регистрах. Некоторые аспекты архитектуры x64, по-видимому, потенциально могут снизить скорость выполнения, например необходимость сохранения и восстановления большого количества регистров в памяти до и после вызова.

Глава 14

1. Несмотря на то что библиотеки – это здорово, у них есть и минусы. Библиотеки, как правило, имеют больше проблем с совместимостью версий, чем встроенные в язык функции. Библиотеки не могут предоставить нотацию, которая была бы краткой и читабельной, как встроенные модули. Наконец, библиотеки не позволяют взаимодействовать с новыми структурами управления для поддержки новых областей применения.

2. Если ваше новое вычисление требует только одного или двух параметров, встречается много раз в типичных приложениях в вашей области и вычисляет новое значение без побочных эффектов, то это хороший кандидат на превращение в оператор. Оператор ограничен двумя операндами или, самое большее, тремя, в противном случае он не обеспечит никакого преимущества в читабельности по сравнению с функцией.
3. В конечном счете мы должны прочитать книги, написанные изобретателями языка Java, чтобы узнать их причины, но одной из них может быть то, что разработчики Java хотели использовать строки как класс и решили, что классы не будут свободно реализовывать операторы ради ссылочной прозрачности.

Глава 15

1. Структуры управления в очень высокоуровневых и доменно-ориентированных языках должны быть гораздо более мощными, чем просто операторы *if* и циклы, в противном случае программистам было бы лучше просто кодировать на распространенном языке.
2. Мы привели несколько примеров, в которых структуры управления предоставляли значения по умолчанию для 0-параметров или обеспечивали закрытие открытого ресурса. Доменно-специфические структуры управления, безусловно, могут предоставлять дополнительную высокоуровневую семантику, такую как выполнение специфического для домена ввода/вывода или доступ к специальному оборудованию таким способом, который трудно реализовать в контексте обычного потока управления.
3. Область применения – анализ строк. Возможно, некоторые дополнительные операторы или встроенные функции улучшили бы выразительные возможности Unicon для анализа строк. Можете ли вы назвать кандидатов, которых вы могли бы добавить к функциям анализа шести строк или к двум функциям перемещения позиций? Вы можете легко провести статистику по распространенным приложениям Icon и/или Unicon и выяснить, какие комбинации *tab()* или *move()* и функций анализа шести строк встречаются в коде наиболее часто и являются кандидатами на то, чтобы стать оператором, помимо *tab(match())*. Я сомневаюсь, что *tab(match())* является самой частой. Но остерегайтесь – если вы добавляете слишком много примитивов, это делает структуру управления более сложной для изучения и освоения. Кроме того, идеи из этой структуры управления могут быть применены к анализу других последовательных данных, таких как массивы/списки числовых значений или значения экземпляров объектов.
4. Очень заманчиво вписать как можно больше дополнительной семантики в структуру управления доменом, чтобы сделать код более лаконичным. Однако если значительное количество конструкций *wsection* не основаны на иерархической 3D-модели и не будут использовать встроенную функциональность *PushMatrix()* и *PopMatrix()*, включение этой структуры в *wsection* может без необходимости замедлить скорость выполнения конструкции.

Глава 16

1. Можно изменить макрос *PostDescrip()*, чтобы он проверял значение null перед проверкой того, является значение классификатором или указателем. Окупится ли такая проверка, зависит от того, насколько дорогостоящим является оператор побитового *И*, а также от фактической частоты различных типов данных, встречающихся во время этих проверок, которую можно измерить, но она может значительно изменяться в зависимости от приложения.

2. Если бы у каждого типа класса была своя область кучи, экземплярам не нужно было бы отслеживать свой размер, что потенциально экономит память для классов, имеющих много маленьких экземпляров. Освобожденными мусорными экземплярами можно было бы управлять в связанном списке и сравнивать их с помощью пометки и освобождения, и экземпляры, возможно, никогда не нужно было бы перемещать или обновлять указатели, что упростило бы сборку мусора. С другой стороны, некоторые программы могут использовать только несколько различных классов, и выделение специальной области кучи для таких классов может оказаться бесполезным.

3. Хотя некоторое время может быть сэкономлено за счет отсутствия перемещения данных во время сборки мусора, со временем значительный объем памяти может быть потерян из-за фрагментации. Небольшие участки свободной памяти могут остаться неиспользованными, потому что последующие запросы на выделение памяти будут запрашивать участки большего размера.

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Клинтон Л. Джеффери

Создай свой собственный язык программирования

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Зам. главного редактора *Сенченкова Е. А.*
Перевод *Минц С. В.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 33,15. Тираж 200 экз.

Веб-сайт издательства: *www.dmkpress.com*

Книга рассказывает о том, как разрабатывать уникальные языки программирования, чтобы сократить время и стоимость создания приложений для новых или специализированных областей применения вычислительной техники. Вы начнете с реализации интерфейса компилятора для вашего языка, включая лексический и синтаксический анализатор, а к концу чтения сможете разрабатывать и воплощать в коде свои собственные языки, позволяющие компилировать и запускать любые программы.

Вы научитесь:

- анализировать требования к новому языку и разрабатывать его синтаксис и семантику;
- создавать лексические и контекстно-независимые грамматические правила для общих выражений и управляющих структур;
- разрабатывать парсер исходного кода и генерировать анализатор синтаксиса;
- создавать ключевые структуры данных в компиляторе и использовать свой компилятор для поддержки синтаксической раскраски кода в редакторе;
- реализовать интерпретатор байт-кода и запускать байт-код, сгенерированный вашим компилятором;
- выполнять обходы дерева и добавлять информацию в дерево синтаксического разбора;
- реализовывать сборщик мусора на вашем языке.

Издание предназначено для разработчиков программного обеспечения, заинтересованных в разработке собственного языка программирования или адаптации существующего языка для той или иной предметной области. Студентам, изучающим информатику, книга пригодится как практическое руководство по разработке языка.

Для изучения материала понадобятся базовые знания и опыт работы с языком высокого уровня, таким как **Java** или **C++**.

Интернет-магазин:
www.dmkpress.com

Оптовая продажа:
КТК «Галактика»
books@alians-kniga.ru

Packt>

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-93700-140-5

9 785937 001405 >