

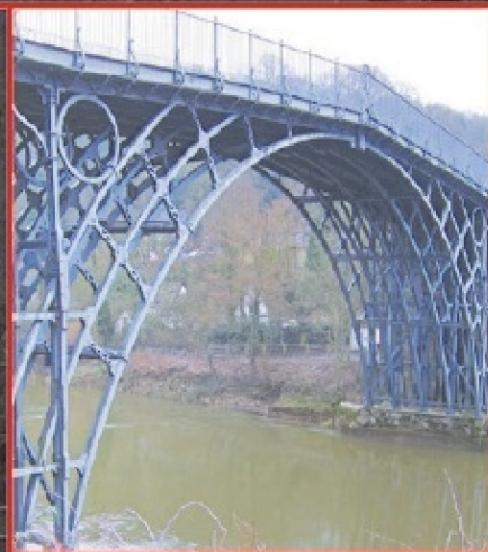
The Addison-Wesley Signature Series



A MARTIN FOWLER SIGNATURE Book

ПРЕДМЕТНО- ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

МАРТИН ФАУЛЕР
ПРИ УЧАСТИИ
РЕБЕККИ ПАРСОНС



**ПРЕДМЕТНО-
ОРИЕНТИРОВАННЫЕ
ЯЗЫКИ
ПРОГРАММИРОВАНИЯ**

DOMAIN- SPECIFIC LANGUAGES

Martin Fowler
With Rebecca Parsons



ADDISON-WESLEY

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ПРЕДМЕТНО- ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Мартин Фаулер
При участии Ребекки Парсонс



Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75

Ф28

УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция канд. техн. наук И.В. Красикова

Научный консультант докт. физ.-мат. наук Д.А. Клюшин

По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:

info@williamspublishing.com, http://www.williamspublishing.com

Фаулер, Мартин.

Ф28 Предметно-ориентированные языки программирования. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2011. — 576 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1738-6 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc,
Copyright © 2011 by Martin Fowler

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2011.

Научно-популярное издание

Мартин Фаулер

Предметно-ориентированные языки программирования

Литературный редактор *Л.Н. Красножон*

Верстка *О.В. Романенко*

Художественный редактор *В.Г. Паевутин*

Корректор *Л.А. Гордиенко*

Подписано в печать 30.06.2011. Формат 70x100/16.

Гарнитура Times. Печать офсетная.

Усл. печ. л. 46,44. Уч.-изд. л. 33,4.

Тираж 1000 экз. Заказ № 0000.

Отпечатано с готовых диапозитивов в ГУП “Типография «Наука»”
199034, Санкт-Петербург, 9-я линия В. О., 12.

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1738-6 (рус.)
ISBN 978-0-321-71294-3 (англ.)

© Издательский дом “Вильямс”, 2011
© Martin Fowler, 2011

Оглавление

Предисловие	17
Благодарности	24
Часть I. Описание	27
Глава 1. Вводный пример	29
Глава 2. Использование предметно-ориентированных языков	49
Глава 3. Реализация предметно-ориентированных языков	63
Глава 4. Реализация внутреннего DSL	85
Глава 5. Реализация внешнего DSL	105
Глава 6. Выбор между внутренними и внешними DSL	119
Глава 7. Альтернативные вычислительные модели	127
Глава 8. Генерация кода	135
Глава 9. Языковые инструментальные средства	143
Часть II. Общие вопросы	157
Глава 10. Зоопарк DSL	159
Глава 11. Семантическая модель	171
Глава 12. Таблица символов	177
Глава 13. Переменная контекста	187
Глава 14. Построитель конструкции	191
Глава 15. Макрос	195
Глава 16. Уведомление	205
Часть III. Вопросы создания внешних DSL	211
Глава 17. Трансляция, управляемая разделителями	213
Глава 18. Синтаксически управляемая трансляция	229
Глава 19. Форма Бэкуса–Наура	237
Глава 20. Лексический анализатор на основе таблицы регулярных выражений	247
Глава 21. Синтаксический анализатор на основе рекурсивного спуска	253
Глава 22. Комбинатор синтаксических анализаторов	263
Глава 23. Генератор синтаксических анализаторов	277
Глава 24. Построение дерева	289
Глава 25. Встроенная трансляция	305
Глава 26. Встроенная интерпретация	311
Глава 27. Внешний код	315

Глава 28. Альтернативная токенизация	325
Глава 29. Вложенные операторные выражения	333
Глава 30. Символ новой строки в качестве разделителя	339
Глава 31. Прочие вопросы	343
Часть IV. Вопросы создания внутренних DSL	347
Глава 32. Построитель выражений	349
Глава 33. Последовательность функций	357
Глава 34. Вложенные функции	361
Глава 35. Соединение методов в цепочки	375
Глава 36. Перенос области видимости в объект	387
Глава 37. Замыкание	397
Глава 38. Вложенные замыкания	403
Глава 39. Список литералов	415
Глава 40. Ассоциативные массивы литералов	417
Глава 41. Динамический отклик	423
Глава 42. Аннотации	439
Глава 43. Работа с синтаксическим деревом	449
Глава 44. Класс таблицы символов	461
Глава 45. Шлифовка текста	469
Глава 46. Расширение литералов	473
Часть V. Альтернативные вычислительные модели	477
Глава 47. Адаптивная модель	479
Глава 48. Таблицы принятия решений	485
Глава 49. Сеть зависимостей	495
Глава 50. Система правил вывода	503
Глава 51. Конечный автомат	517
Часть VI. Генерация кода	521
Глава 52. Генерация с помощью преобразователя	523
Глава 53. Шаблонная генерация	529
Глава 54. Встроенный помощник	537
Глава 55. Генерация, осведомленная о модели	545
Глава 56. Генерация, игнорирующая модель	557
Глава 57. Отделение генерируемого кода с помощью наследования	561
Список литературы	569
Предметный указатель	570

Содержание

Предисловие	17
Благодарности	24
Часть I. Описание	27
Глава 1. Вводный пример	29
1.1. Готическая безопасность	29
1.1.1. Контроллер мисс Грант	30
1.2. Модель конечного автомата	31
1.3. Программирование контроллера мисс Грант	34
1.4. Языки и семантическая модель	41
1.5. Использование генерации кода	43
1.6. Использование языковых инструментальных средств	46
1.7. Визуализация	48
Глава 2. Использование предметно-ориентированных языков	49
2.1. Определение предметно-ориентированных языков	49
2.1.1. Границы DSL	51
2.1.2. Фрагментарные и автономные DSL	53
2.2. Зачем используется DSL	54
2.2.1. Повышение производительности разработки	54
2.2.2. Общение с экспертами в предметной области	55
2.2.3. Изменения в контексте выполнения	56
2.2.4. Альтернативные вычислительные модели	57
2.3. Проблемы с DSL	57
2.3.1. Языковая какофония	58
2.3.2. Стоимость построения	58
2.3.3. Язык гетто	59
2.3.4. Ограниченная абстракция	59
2.4. Более широкая обработка языка	60
2.5. Жизненный цикл DSL	60
2.6. Что такое хороший дизайн DSL	62
Глава 3. Реализация предметно-ориентированных языков	63
3.1. Архитектура обработки DSL	63
3.2. Работа синтаксического анализатора	67
3.3. Грамматики, синтаксис и семантики	68
3.4. Анализ данных	69
3.5. Макросы	72
3.6. Тестирование DSL	72
3.6.1. Тестирование семантической модели	73
3.6.2. Тестирование синтаксического анализатора	76
3.6.3. Тестирование сценариев	80

3.7. Обработка ошибок	80
3.8. Миграция DSL	82
Глава 4. Реализация внутреннего DSL	85
4.1. Свободные API и API командных запросов	85
4.2. Необходимость в слое синтаксического анализа	88
4.3. Использование функций	89
4.4. Коллекции литералов	93
4.5. Использование грамматик для выбора внутренних элементов	95
4.6. Замыкания	96
4.7. Работа с синтаксическим деревом	98
4.8. Аннотации	100
4.9. Расширение литералов	101
4.10. Снижение синтаксического шума	101
4.11. Динамический отклик	102
4.12. Проверка типов	103
Глава 5. Реализация внешнего DSL	105
5.1. Стратегия синтаксического анализа	105
5.2. Стратегия получения вывода	108
5.3. Концепции синтаксического анализа	110
5.3.1. Лексический анализ	110
5.3.2. Грамматики и языки	111
5.3.3. Регулярные, контекстно-свободные и контекстно-зависимые грамматики	112
5.3.4. Нисходящий и восходящий синтаксический анализ	114
5.4. Смешивание с другим языком	115
5.5. XML DSL	117
Глава 6. Выбор между внутренними и внешними DSL	119
6.1. Обучение	119
6.2. Стоимость построения	120
6.3. Осведомленность программистов	121
6.4. Общение с экспертами предметной области	121
6.5. Смешивание в базовом языке	121
6.6. Границы строгой выразительности	122
6.7. Настройка времени выполнения	123
6.8. Скатывание в обобщенность	123
6.9. Соединение DSL	124
6.10. Подведение итогов	124
Глава 7. Альтернативные вычислительные модели	127
7.1. Несколько альтернативных моделей	130
7.1.1. Таблицы решений	130
7.1.2. Система правил вывода	130
7.1.3. Конечный автомат	132
7.1.4. Сеть зависимостей	132
7.1.5. Выбор модели	133

Глава 8. Генерация кода	135
8.1. Выбор объекта генерации	136
8.2. Как генерировать код	138
8.3. Смешивание сгенерированного и рукописного кодов	139
8.4. Генерация удобочитаемого кода	140
8.5. Предварительный анализ генерации кода	141
8.6. Источники дополнительной информации	141
Глава 9. Языковые инструментальные средства	143
9.1. Элементы языковых инструментальных средств	143
9.2. Языки определения схем и метамодели	144
9.3. Редактирование исходного текста и проекционное редактирование	149
9.3.1. Множественные представления	151
9.4. Иллюстративное программирование	151
9.5. Тур по инструментам	153
9.6. Языковые инструментальные средства и CASE-инструменты	154
9.7. Следует ли использовать языковые инструментальные средства	155
Часть II. Общие вопросы	157
Глава 10. Зоопарк DSL	159
10.1. Graphviz	159
10.2. JMock	160
10.3. CSS	162
10.4. Hibernate Query Language (HQL)	163
10.5. XAML	164
10.6. FIT	166
10.7. Make и другие	167
Глава 11. Семантическая модель	171
11.1. Как это работает	171
11.2. Когда это использовать	173
11.3. Вводный пример (Java)	174
Глава 12. Таблица символов	177
12.1. Как это работает	177
12.1.1. Статически типизированные символы	179
12.2. Когда это использовать	180
12.3. Дополнительная литература	180
12.4. Сеть зависимостей во внешнем DSL (Java и ANTLR)	180
12.5. Использование символьных ключей во внутреннем DSL (Ruby)	182
12.6. Использование перечислений для статически типизированных символов (Java)	183
Глава 13. Переменная контекста	187
13.1. Как это работает	187
13.2. Когда это использовать	188
13.3. Чтение INI-файла (C#)	188

Глава 14. Постройтель конструкции	191
14.1. Как это работает	191
14.2. Когда это использовать	192
14.3. Построение простых полетных данных (C#)	192
Глава 15. Макрос	195
15.1. Как это работает	195
15.1.1. Текстовые макросы	196
15.1.2. Синтаксические макросы	199
15.2. Когда это использовать	202
Глава 16. Уведомление	205
16.1. Как это работает	205
16.2. Когда это использовать	206
16.3. Очень простое уведомление (C#)	206
16.4. Уведомление анализа (Java)	207
Часть III. Вопросы создания внешних DSL	211
Глава 17. Трансляция, управляемая разделителями	213
17.1. Как это работает	213
17.2. Когда это использовать	216
17.3. Карты постоянных клиентов (C#)	216
17.3.1. Семантическая модель	217
17.3.2. Синтаксический анализатор	219
17.4. Синтаксический анализ неавтономных инструкций контроллера мисс Грант (Java)	221
Глава 18. Синтаксически управляемая трансляция	229
18.1. Как это работает	230
18.1.1. Лексический анализатор	231
18.1.2. Синтаксический анализатор	233
18.1.3. Генерация вывода	235
18.1.4. Семантические предикаты	236
18.2. Когда это использовать	236
18.3. Дополнительная литература	236
Глава 19. Форма Бэкуса–Наура	237
19.1. Как это работает	237
19.1.1. Символы множественности (операторы Клини)	239
19.1.2. Другие полезные операторы	240
19.1.3. Грамматики, разбирающие выражения	240
19.1.4. Преобразование РБНФ в БНФ	241
19.1.5. Действия	243
19.2. Когда это использовать	245

Глава 20. Лексический анализатор на основе таблицы регулярных выражений	247
20.1. Как это работает	248
20.2. Когда это использовать	249
20.3. Лексический анализатор контроллера мисс Грант (Java)	249
Глава 21. Синтаксический анализатор на основе рекурсивного спуска	253
21.1. Как это работает	254
21.2. Когда это использовать	257
21.3. Дополнительная литература	257
21.4. Рекурсивный спуск и контроллер мисс Грант (Java)	257
Глава 22. Комбинатор синтаксических анализаторов	263
22.1. Как это работает	264
22.1.1. Выполнение действий	267
22.1.2. Функциональный стиль комбинаторов	268
22.2. Когда это использовать	268
22.3. Комбинаторы синтаксических анализаторов и контроллер мисс Грант (Java)	269
Глава 23. Генератор синтаксических анализаторов	277
23.1. Как это работает	277
23.1.1. Встроенные действия	278
23.2. Когда это использовать	280
23.3. Hello World (Java и ANTLR)	280
23.3.1. Написание базовой грамматики	281
23.3.2. Построение синтаксического анализатора	282
23.3.3. Добавление кода действий в грамматику	284
23.3.4. Применение шаблона Generation Gap	286
Глава 24. Построение дерева	289
24.1. Как это работает	289
24.2. Когда это использовать	291
24.3. Использование синтаксиса построения дерева ANTLR (Java и ANTLR)	292
24.3.1. Токенизация	293
24.3.2. Синтаксический анализ	294
24.3.3. Наполнение семантической модели	296
24.4. Построение дерева с использованием кода действий (Java и ANTLR)	299
Глава 25. Встроенная трансляция	305
25.1. Как это работает	305
25.2. Когда это использовать	306
25.3. Контроллер мисс Грант (Java и ANTLR)	306
Глава 26. Встроенная интерпретация	311
26.1. Как это работает	311
26.2. Когда это использовать	311
26.3. Калькулятор (ANTLR и Java)	312

Глава 27. Внешний код	315
27.1. Как это работает	315
27.2. Когда это использовать	317
27.3. Встраивание динамического кода (ANTLR, Java и Javascript)	317
27.3.1. Семантическая модель	318
27.3.2. Синтаксический анализатор	320
Глава 28. Альтернативная токенизация	325
28.1. Как это работает	325
28.1.1. Кавычки	326
28.1.2. Лексическое состояние	328
28.1.3. Изменение типа токена	330
28.1.4. Игнорирование типов токенов	331
28.2. Когда это использовать	332
Глава 29. Вложенные операторные выражения	333
29.1. Как это работает	333
29.1.1. Применение восходящих синтаксических анализаторов	334
29.1.2. Нисходящие синтаксические анализаторы	335
29.2. Когда это использовать	337
Глава 30. Символ новой строки в качестве разделителя	339
30.1. Как это работает	339
30.2. Когда это использовать	341
Глава 31. Прочие вопросы	343
31.1. Синтаксические отступы	343
31.2. Модулярная грамматика	345
Часть IV. Вопросы создания внутренних DSL	347
Глава 32. Построитель выражений	349
32.1. Как это работает	350
32.2. Когда это использовать	350
32.3. Свободный календарь с построителем и без него (Java)	351
32.4. Использование для календаря нескольких построителей (Java)	353
Глава 33. Последовательность функций	357
33.1. Как это работает	357
33.2. Когда это использовать	358
33.3. Простая конфигурация компьютера (Java)	358
Глава 34. Вложенные функции	361
34.1. Как это работает	361
34.2. Когда это использовать	363
34.3. Простой пример конфигурации компьютера (Java)	363
34.4. Обработка различных аргументов с помощью токенов (C#)	365

34.5. Использование токенов подтипов для поддержки интегрированной среды разработки (Java)	366
34.6. Использование инициализаторов объектов (C#)	368
34.7. Повторяющиеся события (C#)	369
34.7.1. Семантическая модель	369
34.7.2. DSL	372
Глава 35. Соединение методов в цепочки	375
35.1. Как это работает	375
35.1.1. Построители или значения	377
35.1.2. Проблема окончания	377
35.1.3. Иерархическая структура	378
35.1.4. Последовательные интерфейсы	378
35.2. Когда это использовать	379
35.3. Простой пример конфигурации компьютера (Java)	379
35.4. Соединение методов в цепочки со свойствами (C#)	383
35.5. Последовательные интерфейсы (C#)	383
Глава 36. Перенос области видимости в объект	387
36.1. Как это работает	388
36.2. Когда это использовать	388
36.3. Коды безопасности (C#)	389
36.3.1. Семантическая модель	389
36.3.2. DSL	391
36.4. Использование вычисления экземпляра (Ruby)	393
36.5. Использование инициализатора экземпляра (Java)	395
Глава 37. Замыкание	397
37.1. Как это работает	397
37.2. Когда это использовать	401
Глава 38. Вложенные замыкания	403
38.1. Как это работает	403
38.2. Когда это использовать	405
38.3. Заворачивание последовательности функций во вложенное замыкание (Ruby)	405
38.4. Простой пример (C#)	407
38.5. Применение соединения методов в цепочки (Ruby)	408
38.6. Последовательность функций с явными аргументами замыканий (Ruby)	410
38.7. Применение вычисления экземпляра (Ruby)	411
Глава 39. Список литералов	415
39.1. Как это работает	415
39.2. Когда это использовать	415
Глава 40. Ассоциативные массивы литералов	417
40.1. Как это работает	417
40.2. Когда это использовать	418

40.3. Настройка конфигурации компьютера с помощью списков и отображений (Ruby)	418
40.4. Имитация Lisp (Ruby)	419
Глава 41. Динамический отклик	423
41.1. Как это работает	424
41.2. Когда это использовать	424
41.3. Расчет бонусов с помощью анализа имен методов (Ruby)	426
41.3.1. Модель	426
41.3.2. Построитель	428
41.4. Расчет бонусов с помощью цепочек вызовов (Ruby)	429
41.4.1. Модель	430
41.4.2. Построитель	430
41.5. Уменьшение шума в коде контроллера тайника (JRuby)	433
Глава 42. Аннотации	439
42.1. Как это работает	440
42.1.1. Определение аннотации	440
42.1.2. Обработка аннотаций	441
42.2. Когда это использовать	442
42.3. Пользовательский синтаксис с обработкой времени выполнения (Java)	443
42.4. Использование метода класса (Ruby)	445
42.5. Динамическая генерация кода (Ruby)	446
Глава 43. Работа с синтаксическим деревом	449
43.1. Как это работает	449
43.2. Когда это использовать	450
43.3. Генерация запросов IMAP из условий C# (C#)	451
43.3.1. Семантическая модель	452
43.3.2. Построение из исходного текста C#	454
43.3.3. Отступление	458
Глава 44. Класс таблицы символов	461
44.1. Как это работает	462
44.2. Когда это использовать	462
44.3. Статически типизированный класс таблицы символов (Java)	463
Глава 45. Шлифовка текста	469
45.1. Как это работает	469
45.2. Когда это использовать	470
45.3. Шлифовка правил диктанта (Ruby)	470
Глава 46. Расширение литералов	473
46.1. Как это работает	473
46.2. Когда это использовать	474
46.3. Ингредиенты рецепта (C#)	474

Часть V. Альтернативные вычислительные модели	477
Глава 47. Адаптивная модель	479
47.1. Как это работает	480
47.1.1. Внедрение императивного кода в адаптивную модель	481
47.1.2. Инструментарий	483
47.2. Когда это использовать	483
Глава 48. Таблицы принятия решений	485
48.1. Как это работает	485
48.2. Когда это использовать	487
48.3. Вычисление оплаты заказа (C#)	487
48.3.1. Модель	487
48.3.2. Синтаксический анализатор	491
Глава 49. Сеть зависимостей	495
49.1. Как это работает	496
49.2. Когда это использовать	498
49.3. Анализ зелий (C#)	498
49.3.1. Семантическая модель	499
49.3.2. Синтаксический анализатор	500
Глава 50. Система правил вывода	503
50.1. Как это работает	504
50.1.1. Цепочки выводов	505
50.1.2. Противоречивые выводы	505
50.1.3. Шаблоны в структуре правила	506
50.2. Когда это использовать	507
50.3. Проверка для членства в клубе (C#)	507
50.3.1. Модель	507
50.3.2. Синтаксический анализатор	508
50.3.3. Развитие DSL	509
50.4. Правила избрания: расширение членства в клубе (C#)	511
50.4.1. Модель	512
50.4.2. Синтаксический анализатор	514
Глава 51. Конечный автомат	517
51.1. Как это работает	517
51.2. Когда это использовать	519
51.3. Контроллер таймера (Java)	519
Часть VI. Генерация кода	521
Глава 52. Генерация с помощью преобразователя	523
52.1. Как это работает	523
52.2. Когда это использовать	524
52.3. Контроллер таймера (Java генерирует C)	525

Глава 53. Шаблонная генерация	529
53.1. Как это работает	529
53.2. Когда это использовать	531
53.3. Генерация конечного автомата тайника с помощью вложенных условных конструкций (Velocity и Java, генерирующие C)	531
Глава 54. Встроенный помощник	537
54.1. Как это работает	538
54.2. Когда это использовать	538
54.3. Состояния тайника (Java и ANTLR)	539
54.4. Должен ли помощник генерировать HTML (Java и Velocity)	541
Глава 55. Генерация, осведомленная о модели	545
55.1. Как это работает	546
55.2. Когда это использовать	546
55.3. Конечный автомат тайника (C)	546
55.4. Динамическая загрузка конечного автомата (C)	553
Глава 56. Генерация, игнорирующая модель	557
56.1. Как это работает	557
56.2. Когда это использовать	558
56.3. Конечный автомат тайника как вложенные условные конструкции (C)	558
Глава 57. Отделение генерируемого кода с помощью наследования	561
57.1. Как это работает	562
57.2. Когда это использовать	563
57.3. Генерация классов из схемы данных (Java и немного Ruby)	563
Список литературы	569
Предметный указатель	570

*Посвящается Синди
— Мартин*

Предисловие

Предметно-ориентированные языки (Domain-Specific Languages — DSL) были частью компьютерного мира еще до того, как я научился программировать. Спросите ветеранов Unix или Lisp, и они будут счастливы утомить вас до судорог рассказами о том, как предметно-ориентированные языки помогали им в их работе и какие трюки с их помощью они ухитрялись выполнять. Тем не менее этим языкам не суждено было стать заметной частью упомянутого компьютерного мира. Большинство людей знают о предметно-ориентированных языках только по чьим-то рассказам, и часто эти рассказы ограничиваются описанием только одной из сторон имеющихся методов.

Я написал эту книгу в надежде изменить ситуацию и предоставить вам как можно больше информации о методах предметно-ориентированных языков, чтобы вы могли принять осознанное решение об их использовании в своей работе и о том, какие именно методы предметно-ориентированных языков применять.

Предметно-ориентированные языки популярны по нескольким причинам, но я остановлюсь только на двух из них: повышение производительности труда разработчиков и улучшение связи с экспертами в предметной области. Правильно выбранный язык может сделать сложный блок кода существенно проще для понимания, что повышает производительность работающих с ним. Он также упрощает общение разработчика программного обеспечения со специалистами в предметной области, по сути предоставляя текст, который одновременно действует и как выполняемое программное обеспечение, и как описание проблемы, которое узкие специалисты в данной области знаний могут прочесть, чтобы понять, как их идеи представлены в программной системе. Трудно переоценить возможность говорить со специалистами на одном языке — эта возможность разрушает самый высокий из барьеров на пути общения между программистами и их клиентами и устраняет массу узких мест в разработке специализированного программного обеспечения.

Однако не хотелось бы и преувеличивать значение предметно-ориентированных языков. Я часто говорю, что всякий раз, рассматривая преимущества применения предметно-ориентированного языка для решения той или иной проблемы, вы должны выполнить еще одно рассмотрение, подставив на этот раз вместо слов “предметно-ориентированный язык” слово “библиотека”. Многое из того, что вы получаете с помощью предметно-ориентированного языка, можно получить, создав соответствующую программную структуру. Большинство предметно-ориентированных языков на самом деле представляют собой просто косметический фасад над библиотеками или программной структурой. В результате выгоды от предметно-ориентированных языков часто оказываются меньшими, чем думают неискушенные в этих вопросах люди; однако в любом

случае для принятия решения нужно четко понимать, какие именно положительные и отрицательные стороны имеет каждое из решений. Знание хорошо себя зарекомендовавших технологий существенно снижает стоимость построения предметно-ориентированного языка (и я надеюсь, что моя книга поможет вам получить эти знания). Фасад может быть декоративной деталью, но часто он так же полезен, как и само здание.

Почему сейчас ?

Предметно-ориентированные языки известны издавна, но в последние годы они породили значительный всплеск интереса. И именно в это время я решил потратить пару лет на эту книгу. Я не уверен, что смогу пояснить, чем вызван такой интерес к предметно-ориентированным языкам, но я поделюсь своей личной точкой зрения.

На рубеже нового тысячелетия возникло ощущение превосходящей разумные рамки стандартизации в языках программирования, по крайней мере в моем мире корпоративного программного обеспечения. Несколько лет Java рассматривался как Единственный Язык Будущего, и даже когда Microsoft выпустила свой C#, он был очень похож на Java. В новых разработках доминирующую позицию занимали компилируемые статические объектно-ориентированные языки программирования с С-образным синтаксисом (даже Visual Basic не избежал попыток приведения его к этому общему знаменателю).

Но вскоре стало ясно, что не все в порядке в королевстве Java/C#. В наличии имелась логика, не хотевшая укладываться в ложе этих языков программирования, что привело к возникновению конфигурационных файлов в формате XML. Программисты шутили, что в результате они пишут больше строк XML, чем на Java/C#. Отчасти это было связано с желанием изменить поведение программы во время выполнения, а отчасти — с желанием выразить аспекты поведения более дружественным и простым для пользователя способом. XML, несмотря на его “зашумленный” синтаксис, позволяет определять собственный словарь и предоставляет строгую иерархическую структуру.

Но “зашумленность” XML все же оказалась слишком большой. Люди жаловались на то, что от угловых скобок у них рябит в глазах, и возникло желание получить преимущества XML-файлов конфигурации без затрат, связанных с применением XML.

Наш рассказ постепенно достиг взрывоподобного явления Ruby On Rails. Независимо от области применения этот язык оказывал огромное влияние на работу программистов с библиотеками и структурными схемами. Большая часть методов работы сообщества Ruby представляет собой более гибкий подход, при котором взаимодействие с библиотекой должно было походить на программирование на специализированном языке. Этот способ мышления восходит к одному из старейших языков программирования — Lisp. При этом подходе и способе мышления можно увидеть цветение даже на каменистой почве Java/C#: в обоих языках все более популярными становятся интерфейсы, вероятно, из-за влияния таких продуктов, как JMock и Hamcrest.

Узнав обо всем этом, я почувствовал, что у меня имеется немалый пробел в знаниях. Я видел людей, использующих XML там, где более удобным и ничуть не более трудным было бы применение иного синтаксиса. Я видел людей, пытавшихся загнать Ruby в тесные рамки сложных выражений, в которых гораздо легче было бы применить пользовательский синтаксис. Я видел программистов, играющих с синтаксическими анализаторами там, где можно было бы обойтись существенно меньшим количеством работы с гибкими интерфейсами в их обычных языках программирования.

Мне кажется, что причина всех этих перегибов — в недостатке знаний. Квалифицированные программисты недостаточно знакомы с технологиями предметно-ориентирован-

ных языков, чтобы принять обоснованное решение о том, какие из них использовать. Этот пробел в знаниях я бы и хотел восполнить.

Значение предметно-ориентированных языков

Более подробно на эту тему мы поговорим в главе 2, “Использование предметно-ориентированных языков”, а пока что я вижу две основные причины, по которым вы должны быть заинтересованы в предметно-ориентированных языках (а значит, и в методах, описанных в этой книге).

Первая причина заключается в повышении производительности труда программиста. Рассмотрим фрагмент кода

```
input =~ /\d{3}-\d{3}-\d{4}/
```

Вы можете распознать в нем регулярное выражение, и, возможно, вы знаете, что оно означает. Регулярные выражения часто критикуют за излишнюю загадочность, но представьте, как бы выглядел обычный код сопоставления шаблону без регулярных выражений. Насколько легко было бы понять и модифицировать такой код в сравнении с регулярным выражением?

Предметно-ориентированные языки позволяют сделать некоторые частные задачи программирования более легкими для понимания, а значит, соответствующие программы можно будет быстрее писать, легче изменять, и они будут менее подвержены ошибкам.

Вторая причина заинтересованности в предметно-ориентированных языках выходит за рамки программирования. Поскольку такие языки меньше и их легче понимать, они позволяют специалистам в предметной области, не являющимся программистами, понимать код, управляющий важными аспектами их профессии. Анализ реального кода со специалистами в предметной области позволит значительно обогатить и сделать более полезным общение между программистами и их клиентами.

Обсуждая подобные темы, люди часто говорят, что предметно-ориентированные языки позволяют полностью избавиться от программистов. Я очень скептически отношусь к этому аргументу; в конце концов, то же самое в свое время говорили и о COBOL. Хотя, конечно, есть языки (такие, как CSS), написанные людьми, которые не называют себя программистами. Однако для предметно-ориентированных языков чтение написанного кода имеет большее значение, чем написание. Если эксперты в предметной области читают и в основном понимают написанный программистами код, то им будет гораздо легче общаться с программистом, который этот код создавал.

Вторая причина использования предметно-ориентированных языков не так проста, но награда стоит затраченных усилий. Общение между программистами и их клиентами — самое узкое место в разработке программного обеспечения, так что все, что может облегчить такое общение, стоит затраченных усилий.

Не бойтесь объема этой книги

Толщина этой книги может немного испугать. Я сам всегда настороженно отношусь к толстым книгам, потому что у всех нас не так уж много времени, которое мы можем посвятить изучению чего-то нового (и которое гораздо важнее при выборе книги, чем ее цена). Поэтому я воспользовался форматом, который предпочитаю в таких случаях — двойной книги.

Двойная книга — это две книги под одной обложкой. Первая — обычная повествовательная книга, предназначенная для чтения от корки до корки. Моя цель в этой книге — предоставить краткий обзор темы, достаточный для понимания, но не для практической работы. Объем этой части — не более пары сотен страниц, что вполне можно позволить себе прочесть полностью.

Вторая (большая по объему) книга представляет собой справочный материал, который предназначен не для подробного чтения (хотя некоторые люди так и делают), а для того, чтобы при необходимости обратиться к ней, как к справочнику. Одни читатели полностью прорабатывают первую книгу, чтобы получить общее представление о теме, а затем обращаются ко второй части через предметный указатель — в поисках только того материала, который их интересует. Другие читают справочный раздел так же, как и описательный. При разделении книги на две я хотел помочь вам в выборе материала, который можно пропустить и в который следует погрузиться как следует. Решение за вами.

Я также попытался сделать справочную часть в достаточной степени самостоятельной, так что если вы захотите узнать, например, о построении деревьев, то можете прочесть только соответствующую часть книги и получить вполне приличное представление о том, что следует делать, даже если вы подзабыли материал первой части. Таким образом, как только вы прочтете первую часть книги, она станет справочником, что очень удобно при практической работе, когда вам нужно найти информацию о конкретных деталях.

Основная причина, по которой книга получилась такой большой, — я просто не придумал, как сделать ее покороче. Одна из главных моих целей в этой книге заключается в предоставлении читателю обзора большого количества различных методов предметно-ориентированных языков. Есть книги о генерации кода, о метапрограммировании в Ruby и об использовании генераторов анализаторов. Я же стремился охватить все эти методы, чтобы вы могли лучше понять их сходства и различия. Все они играют определенную роль в более широком ландшафте, и моя цель — так организовать путешествие по этому ландшафту, чтобы вы ознакомились не только с отдельными пейзажами, но и с каждой из его частей достаточно подробно, чтобы иметь возможность тут же начать работу с описанными методами.

О чем вы узнаете из этой книги

Я задумал эту книгу как руководство по различным видам предметно-ориентированных языков и подходам к их построению. Часто, начиная экспериментировать с предметно-ориентированными языками, люди используют какую-то одну технологию. Смысл этой книги — показать вам широкий спектр методов, чтобы вы могли оценить, какие из них лучше всего подходят в ваших обстоятельствах. Я представил подробную информацию и примеры реализации многих из этих методов. Естественно, я не могу показать вам все, что можно сделать, но для принятия первоначальных решений этого должно быть достаточно.

Из первых глав вы получите представление о том, что такое предметно-ориентированные языки, когда они могут пригодиться и какова их роль в сравнении с библиотеками. Вы узнаете, с чего начать при построении внешних и внутренних предметно-ориентированных языков. В главах о внешних предметно-ориентированных языках рассказывается о роли синтаксического анализатора, полезности генератора анализаторов и о способах применения анализаторов для синтаксического анализа внешнего предметно-ориентированного языка. В главах, посвященных внутренним предметно-ориентированным языкам, показано, как использовать языковые конструкции в стиле таких язы-

ков. Хотя это и не подскажет вам, как лучше всего использовать ваш конкретный язык, но поможет понять, как методы одного языка соотносятся с методами других языков.

В главах, посвященных генерации кода, изложены стратегии генерации кода. Имеется также глава с очень кратким обзором нового поколения инструментов (в большей части книги я ориентируюсь на методы, которые использовались на протяжении десятилетий).

Для кого предназначена эта книга

Основная целевая аудитория этой книги — профессиональные разработчики программного обеспечения, которые рассматривают возможность построения предметно-ориентированного языка. Я представляю такого читателя как имеющего по крайней мере пару лет опыта программирования и хорошо знакомого с основными идеями проектирования программного обеспечения.

Те, кто серьезно занимаются проектированием языков, вероятно, в этой книге не найдут для себя много нового. Я, однако, надеюсь, что вы сочтете полезным использованный мною подход к изложению представленной в книге информации. Хотя в этой области проделана огромная работа, особенно в научной сфере, в мире профессионального программирования в данном вопросе непочатый край работы.

Первые несколько глав повествовательной части книги будут полезны всем, кого интересуют предметно-ориентированные языки и их применение. В этой повествовательной части представлен обзор используемых в данной области технологий.

Это не книга о Java или C#

Как в большинстве книг, которые я пишу, изложенные идеи в значительной степени зависят от языка программирования. Один из моих главных приоритетов — раскрыть общие принципы и модели, которые можно использовать с любым языком программирования, с которым вы будете иметь дело. А значит, идеи в книге должны быть ценными для вас, если вы используете любой современный объектно-ориентированный язык программирования.

Одним из потенциальных пробелов книги являются функциональные языки программирования. Хотя я думаю, что большая часть книги останется применимой и в этом случае, я не имею достаточного опыта работы с функциональными языками, чтобы понять, насколько их парадигма программирования изменит приведенные здесь советы. Книга несколько ограничена в случае процедурных языков (т.е. не объектно-ориентированных языков, таких как C), поскольку некоторые из описанных мною методов базируются на объектно-ориентированном подходе.

Хотя я описываю общие принципы, для того чтобы сделать это правильно, необходимы примеры применения конкретного языка программирования. При выборе языка для примеров основным критерием служила распространность языка; в результате почти все примеры в этой книге написаны на Java или C#. Оба они широко используются и у обоих — С-образный синтаксис, управление памятью и библиотеки, позволяющие устранить многие неприятности. Я не утверждаю, что это наилучшие языки для написания предметно-ориентированных языков (в частности, потому, что лично я так не думаю), но они являются неплохими языками для представления описываемых мною общих концепций. Я пытался использовать оба языка в равной мере, склоняя чашу весов в пользу одного из них только тогда, когда он позволял упростить решение поставленной задачи. Я также старался избегать элементов языка, которые требуют слишком хорошего знания

синтаксиса, хотя это и труднодостижимый компромисс, так как умелое применение внутренних предметно-ориентированных языков часто предусматривает использование синтаксических особенностей базовых языков.

Существует несколько идей, которые требуют обязательного применения динамического языка и поэтому не могут быть показаны на Java или C#. В таких случаях я обращался к Ruby, поскольку это динамический язык, с которым я знаком лучше всего. Он также неплохо подходит для написания предметно-ориентированных языков. И вновь, несмотря на мои личные симпатии к этому языку, не стоит делать вывод, что рассмотренные методы неприменимы в других ситуациях.

Я должен отметить, что имеется много других языков программирования, пригодных для предметно-ориентированных языков, включая ряд языков, специально разработанных для упрощения создания внутренних предметно-ориентированных языков. Я не упоминаю их здесь, потому что не настолько много работал с ними, чтобы с уверенностью их вам рекомендовать. Однако не нужно рассматривать это заявление как мое отрицательное мнение о них.

В частности, одна из трудностей написания книги о предметно-ориентированных языках, не привязанной к конкретному языку программирования, состоит в том, что полезность многих методов очень сильно зависит от особенностей конкретного языка. Всегда следует помнить, что ваша языковая среда может серьезно смеяться акценты по сравнению с обобщениями, которые я вынужден делать в этой книге.

Чего не хватает в этой книге

Один из наиболее разочаровывающих моментов при написании книги — когда я понимаю, что пора остановиться. Мне понадобилось несколько лет, чтобы написать ее, и я верю, что эти годы потрачены не зря. Но я осознаю, что в книге имеется и много пробелов. Я хотел бы заполнить их, но на это потребуется слишком много времени. Мне представляется, что лучше иметь неполную, но опубликованную книгу, чем годами ждать выпуска полной книги, если таковая вообще возможна. Поэтому я упомяну главные пробелы, которые я вижу, но у меня нет времени на их заполнение.

Я уже упоминал об одном из них — о роли функциональных языков. Существует большой опыт создания предметно-ориентированных языков на современных функциональных языках на базе ML и/или Haskell, но в моей книге эта работа, по сути, проигнорирована. Это интересный вопрос — насколько знакомство с функциональными языками может повлиять на структуру изложенного материала?

Возможно, наиболее разочаровывающим пробелом для меня является отсутствие достойного обсуждения вопросов диагностики и обработки ошибок. Я помню, как в университете нас учили, что по-настоящему важной частью компиляторов является диагностика, так что я отлично понимаю, насколько важную тему я не раскрыл.

Моя любимая часть этой книги — раздел об альтернативных вычислительных моделях. Я мог бы написать гораздо больше, но времени неумолимо. В конце концов я решил, что, хотя я написал меньше, чем мог бы, этого должно быть достаточно, чтобы вдохновить вас на самостоятельное изучение других книг.

Справочник

Повествовательная часть книги имеет обычную структуру, но я думаю, что нужно немного подробнее рассказать о структуре справочного раздела. Я разбил его на ряд тем,

сгруппированных в главы, чтобы однотипные темы находились в одном месте. Моя цель — чтобы каждая тема была самостоятельным материалом (если вы прочли повествовательную часть книги, то сможете погрузиться в интересующую вас тему без необходимости обращаться к другому материалу). При наличии исключений я упоминаю об этом в начале соответствующей темы.

Большинство тем описаны как шаблоны. Цель шаблона — решение некоторой часто возникающей задачи. Так, если проблема формулируется как “Как структурировать синтаксический анализатор?”, для ее решения можно воспользоваться двумя моделями — Delimiter-Directed Translation (213) и Syntax-Directed Translation (229).

О шаблонах для разработки программного обеспечения за последние пару десятилетий написано очень много, и разные авторы имеют разные точки зрения на них. Я считаю шаблоны полезными хотя бы потому, что они обеспечивают способ структурирования справочного раздела, как в данной книге. В повествовательной части говорится, что если необходимо выполнить анализ текста, то наиболее вероятными кандидатами являются две указанные модели; шаблоны же дают подробную информацию для выбора одной из них и начала ее реализации.

Хотя большая часть справочного раздела написана с использованием шаблонной структуры, я не применял ее абсолютно везде. Этот метод подходит не для всех разделов справочника. В ряде тем, таких как Nested Operator Expression (333), решение в действительности оказывается центральным моментом темы, и тема не укладывается в структуру, используемую мною для шаблонов. Поэтому в подобных ситуациях я прибегаю к другому способу описания. Есть и иные случаи, которые трудно назвать шаблонами, такие как Macros (195) или BNF (237), но при этом использование шаблонной структуры оказалось хорошим способом описания. В целом я руководствовался шаблонной структурой, в частности отделение “как это работает” от “когда это использовать”, похоже, вполне укладывается в описываемую мною концепцию.

Шаблонная структура

Большинство авторов при описании шаблонов используют некоторый стандартный шаблон. Я столь же “шаблонен” и потому прибегаю к шаблонам, хотя они и не столь “шаблонны”, как у других авторов. Мой шаблон впервые использован в [10].

Пожалуй, самым важным элементом моего шаблона является **название**. Одна из главных причин, по которым я использую шаблоны подобно элементам предметного указателя, — это помогает создать строгий словарь для обсуждения темы. Конечно, нет никакой гарантии, что этот словарь будет широко применяться, но по крайней мере он заставляет быть последовательным при написании книги и при этом служит для других отправной точкой, если, конечно, они пожелают его использовать.

Следующие два элемента — это **намерение** и **набросок**. Они кратко характеризуют шаблон в целом. Они также являются напоминанием о модели, так что, если вы точно знаете, что есть подходящий шаблон, но не помните его названия, эти элементы могут помочь вашей памяти. Намерение представляет собой одно-два предложения, а набросок — нечто более наглядное. Иногда я использую в качестве наброска диаграмму, иногда — фрагмент кода, т.е. то, что, как мне кажется, способно быстро передать сущность рассматриваемой модели. При использовании диаграммы я иногда использую UML, но при этом с радостью воспользуюсь чем угодно, если решу, что так смысл будет передан более наглядно.

Далее наступает очередь **резюме**, как правило, сосредоточивающегося на мотивационном примере. Это пара абзацев, назначение которых все то же — помочь людям получить обзор шаблона, прежде чем погрузиться в детали.

Два основных раздела описания шаблона — это “Как это работает” и “Когда это использовать”. Порядок этих разделов достаточно произволен: если вы пытаетесь решить, следует ли использовать шаблон, можете прочесть только раздел “Когда это использовать”. Однако весьма часто этот раздел не имеет смысла без знания, как работает данный шаблон.

Последнее разделы — с примерами. Хотя я делаю все, чтобы объяснить, как работает шаблон, в разделе “Как это работает”, чтобы поставить точку, часто следует приводить конкретные примеры. Беда в том, что примеры кода опасны, потому что показывают только одно приложение шаблона, и можно подумать, что шаблон — это и есть приведенное приложение, а не общая концепция. Вы можете использовать один и тот же шаблон сто раз, причем всякий раз немного иначе, но у меня весьма ограниченное количество как бумаги, отпущенное на книгу, так и энергии для написания примеров. Таким образом, всегда помните, что картина гораздо шире, чем конкретика приведенного примера.

Все примеры преднамеренно очень просты и сфокусированы только на рассматриваемом шаблоне. Я использую простые и независимые примеры, потому что они соответствуют моей цели — сделать каждую главу справочника самостоятельной. Естественно, при практическом применении шаблона возникнет масса других вопросов, но простой пример, как мне кажется, дает по крайней мере шанс понять основные моменты. Более мощные примеры будут реалистичнее, но заставят вас иметь дело с массой вопросов, не относящихся к изучаемому шаблону. Моя цель — показать вам фрагменты головоломки, а уж собрать их вместе для ваших конкретных целей — это ваша проблема.

Это также означает, что еще одним приоритетом являлась понятность кода. Я не учитывал вопросы производительности, обработки ошибок и другие моменты, которые отвлекали бы от сущности шаблона.

Я старался также избегать кода, которому, по моему мнению, трудно следовать, даже если это идиомы использованного мною языка.

В описании ряда шаблонов будут отсутствовать те или иные разделы; в некоторых случаях это может быть раздел примеров, например когда пример к другому шаблону лучше (в таких ситуациях я указываю эти более подходящие примеры).

Благодарности

Как обычно, когда я пишу книгу, в ее издании участвует много людей. И хотя на обложке указано мое имя, есть много людей, без которых книга была бы значительно менее качественной.

Моя первая благодарность — моей коллеге Ребекке Парсонс (Rebecca Parsons). Одна из моих проблем при написании этой книги заключалась в недостатке серьезного академического образования. Ребекка с ее опытом в теории языков программированияоказала мне огромную помощь. Кроме того, она — один из ведущих технических специалистов-практиков ThoughtWorks, так что она сочетает отличную академическую подготовку с большим практическим опытом. Ей хотелось (и, безусловно, она достаточно квалифицирована для этого) играть более важную роль в издании этой книги, но компания ThoughtWorks сочла ее слишком ценным работником. Я рад, что Ребекка все же смогла выкроить время для наших многочасовых бесед.

Когда дело доходит до рецензентов, автор всегда надеется (и отчасти боится) на рецензента, который прочтет книгу и найдет в ней все проблемные места. Мне посчастливилось познакомиться с Майклом Хангром (Michael Hunger), который замечательно сыг-

рал эту роль. С первых дней появления книги на моем сайте он начал бомбардировать меня сообщениями об ошибках и советами по их устранению, и поверьте, это было то, что нужно... Майкл также подталкивал меня к описанию методов с использованием статической типизации, в частности по отношению к статически типизированным таблицам символов (*Symbol Table* (177)). Он сделал множество дополнительных предложений, которых хватит еще на пару книг, и я надеюсь когда-нибудь их написать.

За несколько последних лет мы с коллегами Ребеккой Парсонс (Rebecca Parsons), Нилом Фордом (Neal Ford) и Ола Бини (Ola Bini) написали несколько учебных пособий по данному материалу. В процессе работы над ними было сформулировано множество идей, которые вошли в данную книгу.

Компания ThoughtWorks щедро дала мне много свободного времени, чтобы написать эту книгу. Я рад, что нашел компанию, которой удалось сделать так, чтобы мне хотелось оставаться активным членом ее команды.

У меня была сильная группа официальных экспертов, которые от корки до корки изучили эту книгу, нашли, как я надеюсь, все ошибки и предложили массу улучшений:

Дэвид Бок (David Bock)
Жилад Брака (Gilad Bracha)
Айно Корри (Aino Corry)
Свен Эффтинж (Sven Efftinge)
Эрик Эванс (Eric Evans)
Джей Филдс (Jay Fields)
Стив Фриман (Steve Freeman)
Брайан Готц (Brian Goetz)
Стив Хайес (Steve Hayes)
Клиффорд Хиз (Clifford Heath)
Майкс Хангр (Michael Hunger)

Дэвид Инг (David Ing)
Джереми Миллер (Jeremy Miller)
Рави Мохан (Ravi Mohan)
Тиранс Парр (Terance Parr)
Нат Прайс (Nat Pryce)
Крис Селлс (Chris Sells)
Натаниэль Шутта (Nathaniel Schutta)
Крег Тавернер (Craig Taverner)
Дэйв Томас (Dave Thomas)
Гленн Вандербург (Glenn Vanderburg)

Небольшое, но важное спасибо Дэвиду Ингу (David Ing) за название “Зоопарк DSL”.

Одна из приятных сторон должности редактора серии — возможность подобрать действительно хорошую команду авторов, которая помогает решать все вопросы и оттачивать идеи. Я особенно хочу поблагодарить Эллиотта Расти Харольда (Elliotte Rusty Harold) за его подробнейшие комментарии и анализ.

В качестве генераторов идей выступали многие мои коллеги из ThoughtWorks. Я хочу поблагодарить всех, кто разрешал мне копаться в своих проектах в течение последних нескольких лет. В них я нашел гораздо больше идей, чем могу описать, и я получал большое удовольствие от этого поиска.

Два человека сделали весьма ценные замечания по Интернету, на которые мне удалось отреагировать, прежде чем книга была передана в печать: Павел Бернхаузер (Pavel Bernhauser, “Mocky”) и Роман Яковенко (Roman Yakovenko, “tdyer”).

Спасибо всем сотрудникам издательства, которое опубликовало мою книгу. Особенно хочется отметить работу выпускающего редактора Грега Донча (Greg Doench) и главного редактора Джона Фуллера (John Fuller).

Дмитрий Кирсанов (Dmitry Kirsanov) превратил мой небрежный английский в язык, достойный страниц книги, а Алина Кирсанова (Alina Kirsanova) подготовила макет книги и ее предметный указатель.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com

WWW: <http://www.williamspublishing.com>

Наши почтовые адреса:

в России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

в Украине: 03150, Киев, а/я 152

Часть I

Описание

Глава 1

Вводный пример

Начиная писать, я обычно кратко объясняю, о чем, собственно, я пишу. В данном случае мне нужно объяснить читателям, что такое предметно-ориентированный язык (Domain-Specific Language — DSL). Я хотел бы сделать это на конкретном примере, за которым последуют более абстрактные определения. Итак, я начинаю с примера демонстрации видов DSL. В следующей главе я постараюсь обобщить определение в нечто более практическое.

1.1. Готическая безопасность

В моей памяти остались яркие детские воспоминания о приключенческих телефильмах. Часто эти фильмы снимались в старых готических замках, где героям нужно было искать тайники с сокровищами. Чтобы найти их, героям приходилось выполнять странные действия, например потянуть определенный канделябр на лестнице, а затем два раза нажать на незаметный выступ на стене.

Давайте представим, что некоторая компания принимает решение создать систему безопасности на основе этой идеи. Должна быть настроена беспроводная сеть и установлены небольшие устройства, которые посылают четырехсимвольные сообщения, когда происходят те или иные интересные события. Например, датчик, прикрепленный к выдвижному ящику, будет отправлять при его открывании сообщение D2OP. Имеются также небольшие управляющие устройства, которые отвечают на четырехсимвольные командные сообщения, например такое устройство может открыть дверь, когда получает сообщение D1UL.

В центре всего этого — некий контроллер программного обеспечения, который прослушивает сообщения о событиях, определяет, что следует делать, и посылает командные сообщения. Компания во время кризиса по дешевке закупила много неких периферийных устройств с поддержкой Java и использует их в качестве контроллеров. Когда клиент закупает готическую систему безопасности, здание оснащается большим количеством устройств и контроллеров с управляющей программой на языке программирования Java.

В данном примере я сосредоточусь на этой управляющей программе. У каждого клиента — индивидуальные потребности, но если рассмотреть выборку достаточных размеров, то можно обнаружить общие закономерности. Мисс Грант закрывает дверь спальни, открывает ящик, и включается свет для доступа к тайнику. Мисс Шоу поворачивает кран и открывает один из двух шкафчиков, включив соответствующую лампочку. У мисс Смит имеется тайник внутри запертого шкафа в ее офисе. Чтобы открыть шкаф, она должна снять со стены картину, три раза включить настольную лампу и открыть верхний ящик своего картотечного шкафа. Если она забывает выключить настольную лампу прежде, чем откроет тайник, включается сигнал тревоги.

Хотя этот пример кажется бредом, главное в нем то, что мы имеем семейство систем, которые при общности большинства компонентов и поведения имеют некоторые важные различия. В данном случае общее то, что контроллер посыпает и принимает сообщения одинаково от всех клиентов, но последовательность событий и команд у них отличается. Мы хотим устроить все так, чтобы компания могла установить новую систему с минимальными усилиями, поэтому программирование последовательности действий в контроллере должно быть как можно более простым.

Знакомясь со всеми этими примерами, мы видим, что имеет смысл рассматривать контроллер как конечный автомат. Каждый датчик посылает событие, которое может изменить состояние контроллера. Контроллер, перейдя в новое состояние, может послать по сети управляющее сообщение.

Сейчас я должен признаться, что изначально я писал все наоборот. Конечный автомат представляет собой хороший пример для DSL, и я начал именно с него. Готический же замок я выбрал потому, что этот пример конечного автомата оказался не таким скучным, как другие.

1.1.1. Контроллер мисс Грант

Хотя у моей мифической компании тысячи довольных клиентов, мы сосредоточим свое внимание только на моей любимой мисс Грант. В ее спальне есть тайник, который обычно закрыт на замок и спрятан. Чтобы его открыть, нужно закрыть дверь, открыть второй ящик комода и включить лампочку у кровати — в любом порядке. Как только это будет сделано, тайник разблокируется.

Я могу представить себе эту последовательность в виде диаграммы состояний, показанной на рис. 1.1.

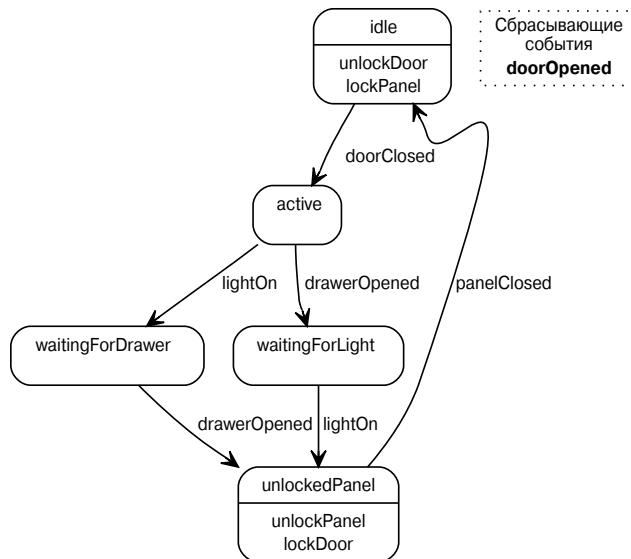


Рис. 1.1. Диаграмма состояний контроллера мисс Грант

Если вы еще не сталкивались с конечными автоматами, то знайте, что они являются распространенным способом описания поведения — не универсальным, но хорошо подходящим для подобных ситуаций. Основная идея заключается в том, что контроллер мо-

жет находиться в различных состояниях. Когда вы находитесь в некотором состоянии, определенные события будут переводить вас в другие состояния, в которые имеются переходы из текущего состояния. Таким образом, последовательность событий ведет вас от состояния к состоянию. В этой модели действия (передача команд) происходят при входе в состояние (другие виды конечных автоматов могут выполнять действия в других местах).

Этот контроллер представляет собой простой конечный автомат, хотя и имеется некоторая хитрость. Контроллеры различных клиентов имеют разные состояния покоя, в которых находятся большую часть времени. Некоторые события могут приводить к возврату системы в это состояние покоя, даже если она находится в середине и далее более интересной последовательности переходов между состояниями, таким образом, эффективно выполняя сброс модели. В случае мисс Грант таким “сбрасывающим событием” является открытие двери.

Введение сбрасывающего события означает, что описанный здесь конечный автомат отличается от классической модели конечного автомата. Есть несколько довольно хорошо известных вариаций конечных автоматов; данная модель начинается с применения одного из них, но сбрасывающее событие добавляет функциональную возможность, уникальную для данного контекста.

В частности, следует отметить, что сбрасывающее событие не является абсолютно необходимым для контроллера мисс Грант. В качестве альтернативы можно было бы просто добавить к каждому состоянию переход по событию `doorOpened` в состояние покоя. Понятие сбрасывающего события является полезным, поскольку существенно упрощает диаграмму.

1.2. Модель конечного автомата

После того как команда решила, что конечный автомат представляет собой хорошую абстракцию для определения работы контроллера, следующий шаг заключается в обеспечении абстракции в самой программе. Если мы хотим думать о поведении контроллера в терминах событий, состояний и переходов, то нужно сделать так, чтобы этот словарный запас присутствовал и в программном коде. По сути, это принцип проектирования на основе предметной области (*Domain-Driven Design*) *единого языка* (*Ubiquitous Language*) [7], т.е. мы строим общий язык для использования как специалистами из данной предметной области (которые описывают, как должна работать создаваемая система безопасности), так и программистами.

При работе в Java естественный способ сделать это — с помощью *модели предметной области* [10] конечного автомата.

Контроллер общается с устройствами путем получения сообщений о событиях и отправки управляющих сообщений. Все они представляют собой четырехбуквенные коды, пересылаемые по коммуникационным каналам. Я хочу обращаться к ним в коде контроллера через символические имена, поэтому я создаю классы событий и команд с кодами и именами. Они представлены в виде отдельных классов (с суперклассом), поскольку играют разные роли в коде контроллера (рис. 1.2).

```
class AbstractEvent...
    private String name, code;

    public AbstractEvent(String name, String code) {
        this.name = name;
        this.code = code;
    }
    public String getCode() { return code; }
```

```

public String getName() { return name; }

public class Command extends AbstractEvent

public class Event extends AbstractEvent

```

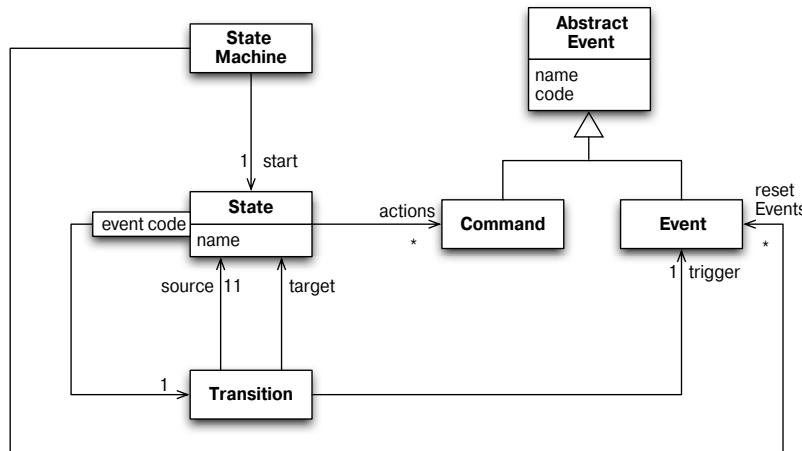


Рис. 1.2. Диаграмма классов структуры конечного автомата

Класс состояния отслеживает отправляемые команды и переходы.

```

class State...
private String name;
private List<Command> actions = new ArrayList<Command>();
private Map<String, Transition> transitions =
    new HashMap<String, Transition>();

class State...
public void addTransition(Event event,
                           State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(),
                    new Transition(this, event,
                                   targetState));
}

class Transition...
private final State source, target;
private final Event trigger;

public Transition(State source, Event trigger,
                  State target) {
    this.source = source;
    this.target = target;
    this.trigger = trigger;
}
public State getSource() {return source;}
public State getTarget() {return target;}
public Event getTrigger() {return trigger;}
public String getCode() {return trigger.getCode();}

```

Конечный автомат находится в своем начальном состоянии.

```
class StateMachine...
    private State start;

    public StateMachine(State start) {
        this.start = start;
    }
```

Все другие состояния конечного автомата достижимы из данного.

```
class StateMachine...
    public Collection<State> getStates() {
        List<State> result = new ArrayList<State>();
        collectStates(result, start);
        return result;
    }

    private void collectStates(Collection<State> result,
                               State s) {
        if (result.contains(s)) return;
        result.add(s);
        for (State next : s.getAllTargets())
            collectStates(result, next);
    }

    class State...
        Collection<State> getAllTargets() {
            List<State> result = new ArrayList<State>();
            for (Transition t : transitions.values())
                result.add(t.getTarget());
            return result;
        }
    }
```

Для обработки сбрасывающих событий я поддерживаю в конечном автомате их список.

```
class StateMachine...
    private List<Event> resetEvents = new ArrayList<Event>();

    public void addResetEvents(Event... events) {
        for (Event e : events) resetEvents.add(e);
    }
```

Не обязательно иметь отдельную структуру для сбрасывающих событий наподобие приведенной выше. Их можно обрабатывать путем простого объявления дополнительных переходов конечного автомата.

```
class StateMachine...
    private void addResetEvent_byAddingTransitions(Event e) {
        for (State s : getStates())
            if (!s.hasTransition(e.getCode()))
                s.addTransition(e, start);
    }
```

Я предпочитаю явные сбрасывающие события, так как они лучше выражают мои намерения. Хотя это и немножко усложняет конечный автомат, становится яснее, как именно он должен функционировать, а также назначение определения каждого конкретного автомата.

Закончив со структурами, перейдем к поведению. Как оказывается, это очень просто. Контроллер имеет метод обработчика, который принимает код события, получаемый от устройства.

```
class Controller...
    private State currentState;
    private StateMachine machine;
```

```

public CommandChannel getCommandChannel() {
    return commandsChannel;
}

private CommandChannel commandsChannel;

public void handle(String eventCode) {
    if (currentState.hasTransition(eventCode))
        transitionTo(currentState.targetState(eventCode));
    else if (machine.isResetEvent(eventCode))
        transitionTo(machine.getStart());
    // Игнорирование неизвестных событий
}

private void transitionTo(State target) {
    currentState = target;
    currentState.executeActions(commandsChannel);
}

class State...
public boolean hasTransition(String eventCode) {
    return transitions.containsKey(eventCode);
}
public State targetState(String eventCode) {
    return transitions.get(eventCode).getTarget();
}
public void executeActions(CommandChannel
                           commandsChannel) {
    for (Command c : actions)
        commandsChannel.send(c.getCode());
}

class StateMachine...
public boolean isResetEvent(String eventCode) {
    return resetEventCodes().contains(eventCode);
}

private List<String> resetEventCodes() {
    List<String> result = new ArrayList<String>();
    for (Event e : resetEvents) result.add(e.getCode());
    return result;
}

```

Здесь игнорируются все события, не зарегистрированные в данном состоянии. Для любого распознанного события происходят переход в целевое состояние и выполнение команд, определенных в этом целевом состоянии.

1.3. Программирование контроллера мисс Грант

Теперь, реализовав модель конечного автомата, я могу приступить к программированию контроллера мисс Грант следующим образом.

```

Event doorClosed = new Event("doorClosed", "D1CL");
Event drawerOpened = new Event("drawerOpened", "D2OP");
Event lightOn = new Event("lightOn", "L1ON");
Event doorOpened = new Event("doorOpened", "D1OP");
Event panelClosed = new Event("panelClosed", "PNCL");

Command unlockPanelCmd = new Command("unlockPanel", "PNUL");
Command lockPanelCmd = new Command("lockPanel", "PNLK");

```

```

Command lockDoorCmd = new Command("lockDoor", "D1LK");
Command unlockDoorCmd = new Command("unlockDoor", "D1UL");

State idle = new State("idle");
State activeState = new State("active");
State waitingForLightState = new State("waitingForLight");
State waitingForDrawerState = new State("waitingForDrawer");
State unlockedPanelState = new State("unlockedPanel");

StateMachine machine = new StateMachine(idle);

idle.addTransition(doorClosed, activeState);
idle.addAction(unlockDoorCmd);
idle.addAction(lockPanelCmd);

activeState.addTransition(drawerOpened,
                         waitingForLightState);
activeState.addTransition(lightOn, waitingForDrawerState);
waitingForLightState.addTransition(lightOn,
                                   unlockedPanelState);
waitingForDrawerState.addTransition(drawerOpened,
                                    unlockedPanelState);

unlockedPanelState.addAction(unlockPanelCmd);
unlockedPanelState.addAction(lockDoorCmd);
unlockedPanelState.addTransition(panelClosed, idle);

machine.addResetEvents(doorOpened);

```

Я рассматриваю этот последний фрагмент кода как совершенно отличный по своему характеру от предыдущих частей. Приведенный ранее код описывал построение модели конечного автомата; этот же последний фрагмент кода посвящен настройке этой модели для одного конкретного контроллера. Вы должны часто встречаться с таким разделением. С одной стороны, имеется код библиотек, программных структур или реализации компонентов, а с другой — код конфигурации или сборки компонента. По сути, это отделение общего кода от варьируемого. Мы структурируем общий код в набор компонентов, которые затем настраиваем для различных целей (рис. 1.3).



Рис. 1.3. Библиотека, используемая в нескольких конфигурациях

Вот другой способ представления этого конфигурационного кода.

```

<stateMachine start="idle">
    <event name="doorClosed" code="D1CL"/>
    <event name="drawerOpened" code="D2OP"/>
    <event name="lightOn" code="L1ON"/>
    <event name="doorOpened" code="D1OP"/>
    <event name="panelClosed" code="PNCL"/>

    <command name="unlockPanel" code="PNUL"/>
    <command name="lockPanel" code="PNLK"/>
    <command name="lockDoor" code="D1LK"/>
    <command name="unlockDoor" code="D1UL"/>

    <state name="idle">
        <transition event = "doorClosed"
                    target="active"/>
        <action command="unlockDoor"/>
        <action command="lockPanel"/>
    </state>

    <state name="active">
        <transition event = "drawerOpened"
                    target="waitingForLight"/>
        <transition event = "lightOn"
                    target="waitingForDrawer"/>
    </state>

    <state name="waitForLight">
        <transition event = "lightOn"
                    target="unlockedPanel"/>
    </state>

    <state name="waitForDrawer">
        <transition event = "drawerOpened"
                    target="unlockedPanel"/>
    </state>

    <state name="unlockedPanel">
        <action command="unlockPanel"/>
        <action command="lockDoor"/>
        <transition event = "panelClosed"
                    target="idle"/>
    </state>

    <resetEvent name="doorOpened"/>
</stateMachine>
```

Этот стиль представления должен быть понятен большинству читателей — это обычный XML-файл. У такого способа есть несколько преимуществ. Одним из очевидных его достоинств является то, что не нужно компилировать отдельную Java-программу для каждого устанавливаемого на “боевое дежурство” контроллера — мы можем просто скомпилировать компоненты конечного автомата плюс соответствующий синтаксический анализатор в общий JAR-файл и передать XML-файл, который конечный автомат должен считывать при запуске. Любые изменения в поведении контроллера могут быть сделаны без распространения нового JAR-файла. Конечно, за это придется платить тем, что многие ошибки в синтаксисе конфигурации смогут быть обнаружены только во время выполнения, хотя некоторые XML-схемы могут немного в этом помочь. Я также большой поклонник всестороннего тестирования, которое отлавливает большинство ошибок

времени компиляции, а также другие ошибки, недоступные проверке типов. При таком тестировании я гораздо меньше беспокоюсь о переносе обнаружения ошибок со времени компиляции на время выполнения.

Второе преимущество состоит в выразительности самого файла. Больше не нужно беспокоиться о деталях подключения с применением переменных. Вместо этого использован декларативный подход, который во многих отношениях гораздо понятнее. Ограничения, накладываемые на содержимое такого файла, зачастую очень полезны, так как уменьшают влияние склонности людей к ошибкам и опечаткам.

Вы, наверное, часто слышали, как о таком подходе говорят как о декларативном программировании. Обычно используемая модель — императивная, когда мы управляем компьютером, указывая ему последовательность действий, которые он должен выполнить. “Декларативный” — термин достаточно туманный, но в целом относится к подходам, которые отходят от императивной модели. Здесь мы делаем шаг в указанном направлении: отходим от перетасовки переменных и представляем действия и переходы поэлементами XML.

Эти преимущества поясняют, почему так много программ на языках программирования Java и C # используют конфигурационные файлы в формате XML. Порой создается впечатление, что приходится писать больше XML-кода, чем кода на вашем основном языке программирования.

Вот еще одна версия кода конфигурации.

```
events
  doorClosed    D1CL
  drawerOpened  D2OP
  lightOn       L1ON
  doorOpened    D1OP
  panelClosed   PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel  PNUL
  lockPanel    PNLK
  lockDoor     D1LK
  unlockDoor   D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn      => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end
```

```
state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end
```

Это тоже код, хотя и использующий незнакомый вам синтаксис. На самом деле это некоторый пользовательский синтаксис, который я создал для данного конкретного примера. Мне кажется, что такой синтаксис проще и писать, и, прежде всего, читать, чем синтаксис XML. Он краток и не замусорен массой скобок и кавычек, чем страдает XML. Я думаю, что вы бы разработали в этой ситуации свой синтаксис, отличный от моего, но главное здесь в том, что вы можете создать такой символ, который предпочтителен для вас и вашей команды. Можете загрузить его в во время выполнения (как и XML), но вы не обязаны делать это (как не обязаны и в случае XML), если хотите выполнить загрузку во время компиляции.

Этот язык представляет собой предметно-ориентированный язык, который имеет множество общих характеристик DSL. Во-первых, он создан только для очень узкой цели и не может сделать что-нибудь иное, кроме настройки конкретного вида конечного автомата. В результате этот DSL очень прост — в нем нет поддержки управляющих структур или чего-то еще в этом роде. Он даже не полный по Тьюрингу. Вы не могли бы написать все приложение на этом языке; все, что вы можете сделать, — описать один небольшой аспект приложения. В результате DSL должен быть скомбинирован с другими языками, чтобы иметь возможность сделать что-нибудь реальное. Но простота DSL означает, что его легко редактировать и обрабатывать.

Эта простота делает язык более понятным для тех, кто пишет программы для контроллера, но при этом может сделать поведение контроллера видимым не только разработчикам. Люди, которые настраивают системы, могут посмотреть на такой код и понять, как должна работать система, даже если не понимают кода Java, работающего в контроллере. Даже если они только читают код DSL, этого может быть достаточно, чтобы обнаружить ошибки и эффективно общаться с разработчиками на языке программирования Java. Хотя при создании предметно-ориентированного языка, который действует как средство общения с экспертами в предметной области и бизнес-аналитиками, возникает немало практических трудностей, восполнение наиболее сложного коммуникационного пробела в разработке программного обеспечения стоит того, чтобы попытаться их преодолеть.

Теперь взглянем еще раз на XML-представление. Является ли оно предметно-ориентированным языком? Я считаю, что является. Он скрыт в синтаксисе XML, но это по-прежнему предметно-ориентированный язык. В связи с этим примером возникает вопрос о дизайне: что лучше — пользовательский синтаксис DSL или синтаксис XML? Синтаксис XML может оказаться проще для анализа, поскольку в этой области имеется множество наработок (впрочем, мне потребовалось примерно одно и то же количество времени как для написания анализатора для пользовательского синтаксиса, так и для написания анализатора для XML). Я считаю, что пользовательский синтаксис гораздо легче для чтения и понимания, по крайней мере в данном конкретном случае. Следует сказать, что большинство XML-файлов конфигурации по сути представляют собой DSL.

Теперь рассмотрим приведенный код. Похож ли он на DSL, разработанный для нашей задачи?

```
event :doorClosed,      "D1CL"
event :drawerOpened,   "D2OP"
event :lightOn,         "L1ON"
event :doorOpened,     "D1OP"
event :panelClosed,    "PNCL"
```

```
command :unlockPanel, "PNUL"
command :lockPanel,   "PNLK"
command :lockDoor,    "D1LK"
command :unlockDoor,  "D1UL"

resetEvents :doorOpened

state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end

state :active do
  transitions :drawerOpened => :waitingForLight,
               :lightOn => :waitingForDrawer
end

state :waitingForLight do
  transitions :lightOn => :unlockedPanel
end

state :waitingForDrawer do
  transitions :drawerOpened => :unlockedPanel
end

state :unlockedPanel do
  actions :unlockPanel, :lockDoor
  transitions :panelClosed => :idle
end
```

Этот код несколько многословнее, чем приведенный ранее пример кода пользовательского языка, но он, тем не менее, вполне понятен. Читатели с аналогичными моим предпочтениями языков программирования, вероятно, узнали код на языке программирования Ruby. Ruby предоставляет массу синтаксических вариантов, которые делают код более читаемым, так что я могу сделать его очень похожим на рассмотренный пользовательский язык.

Разработчики на языке программирования Ruby могут счесть этот фрагмент исходным текстом DSL. Я использую подмножество возможностей Ruby и реализую те же идеи, что и в случае XML и пользовательского синтаксиса. По существу, я вложил DSL в Ruby, используя в качестве собственного синтаксиса подмножество Ruby. В определенной мере это просто вопрос отношения к коду. Я предпочел смотреть на код Ruby сквозь “очки” DSL. Эта точка зрения имеет давние традиции — программисты на Lisp часто прибегают к созданию DSL внутри Lisp.

Это подводит меня к необходимости отметить существование двух видов DSL, которые я называю внешними и внутренними предметно-ориентированными языками. **Внешний DSL** представляет собой предметно-ориентированный язык, не зависящий от основного языка программирования, с которым он работает. Этот язык может применять пользовательский синтаксис или следовать синтаксису иного представления, такого как XML. **Внутренний DSL** представляет собой предметно-ориентированный язык, представленный с помощью синтаксиса языка программирования общего назначения. Это — стилизованное использование языка программирования для предметно-ориентированных целей.

Вы можете услышать еще один термин — **встроенный DSL**, используемый в качестве синонима для внутреннего DSL. Хотя это достаточно широко используемый термин, я избегаю его употребления, потому что термин “встроенный язык” может применяться к языкам сценариев, встроенным в приложения, таким как VBA в Excel или Scheme в Gimp.

Теперь еще раз подумайте об исходном коде конфигурации на языке программирования Java. DSL ли это? Я считаю, что нет. Этот код скорее крепко привязан к API, в то время как приведенный выше фрагмент на языке Ruby скорее вызывает ощущение декларативного языка. Означает ли это, что внутренний DSL в Java невозможен? А что вы скажете о таком исходном тексте?

```
public class BasicStateMachine extends StateMachineBuilder {

    Events doorClosed, drawerOpened, lightOn, panelClosed;
    Commands unlockPanel, lockPanel, lockDoor, unlockDoor;
    States idle, active, waitingForLight,
              waitingForDrawer, unlockedPanel;
    ResetEvents doorOpened;

    protected void defineStateMachine() {
        doorClosed. code("D1CL");
        drawerOpened. code("D2OP");
        lightOn. code("L1ON");
        panelClosed. code("PNCL");

        doorOpened. code("D1OP");

        unlockPanel. code("PNUL");
        lockPanel. code("PNLK");
        lockDoor. code("D1LK");
        unlockDoor. code("D1UL");

        idle
            .actions(unlockDoor, lockPanel)
            .transition(doorClosed).to(active)
            ;
        active
            .transition(drawerOpened).to(waitingForLight)
            .transition(lightOn). to(waitingForDrawer)
            ;
        waitingForLight
            .transition(lightOn).to(unlockedPanel)
            ;
        waitingForDrawer
            .transition(drawerOpened).to(unlockedPanel)
            ;
        unlockedPanel
            .actions(unlockPanel, lockDoor)
            .transition(panelClosed).to(idle)
            ;
    }
}
```

Это достаточно странное форматирование исходного текста, которое использует некоторые необычные соглашения, но это действительно код Java. Я бы назвал его предметно-ориентированным языком; хотя он не столь строг и логичен, как Ruby DSL, тем не менее он вполне справляется со своей задачей.

Чем же внутренний DSL отличается от обычного API? Это сложный вопрос, на который я постараюсь ответить позже, в разделе “Свободные API и API командных запросов”, с. 85.

Еще один термин, с которым вы можете столкнуться при рассмотрении внутренних DSL, — **свободный интерфейс** (fluent interface). Он подчеркивает тот факт, что внутренний DSL на самом деле представляет собой всего лишь особый вид API, разработанный с этим неуловимым качеством — свободой. Учитывая это различие, наверное, стоило бы иметь отдельный термин и для не свободных интерфейсов, так что я буду использовать термин **API командных запросов** (command-query API).

1.4. Языки и семантическая модель

В начале этого примера я говорил о построении модели для конечного автомата. Наличие такой модели и ее взаимоотношения с DSL — жизненно важные вопросы. В нашем примере роль DSL заключается в настройке модели конечного автомата. Так что, когда я анализирую версию с пользовательским синтаксисом и встречаю строки

```
events
  doorClosed D1CL
```

я должен создать объект события (`new Event ("doorClosed", "D1CL")`) и сохранить его (в таблице символов, **Symbol Table** (177)) так, чтобы, когда я встречу текст `doorClosed => active`, я мог бы включить его в переход (с использованием `addTransition`). Модель по сути представляет собой двигатель, обеспечивающий поведение конечного автомата. Можно сказать, что основная мощь проекта проистекает от наличия модели. Все, что делает DSL, — всего лишь обеспечивает удобочитаемый способ заполнения этой модели.

С точки зрения предметно-ориентированного языка я ссылаюсь на эту модель как на семантическую (**Semantic Model** (171)). Когда программисты обсуждают какой-нибудь язык программирования, в их лексиконе часто встречаются такие термины, как “синтаксис” и “семантика”. Синтаксис описывает корректные выражения, которые могут встречаться в программе, т.е. все, что охватывается грамматикой DSL с пользовательским синтаксисом. Семантика же программы — это ее смысл, т.е. то, что она делает при запуске. В данном случае это модель, которая и определяет семантику (рис. 1.4). Если вы привыкли к использованию *предметных моделей* [10], то пока что можете рассматривать семантическую модель как нечто очень близкое к упомянутой предметной модели.

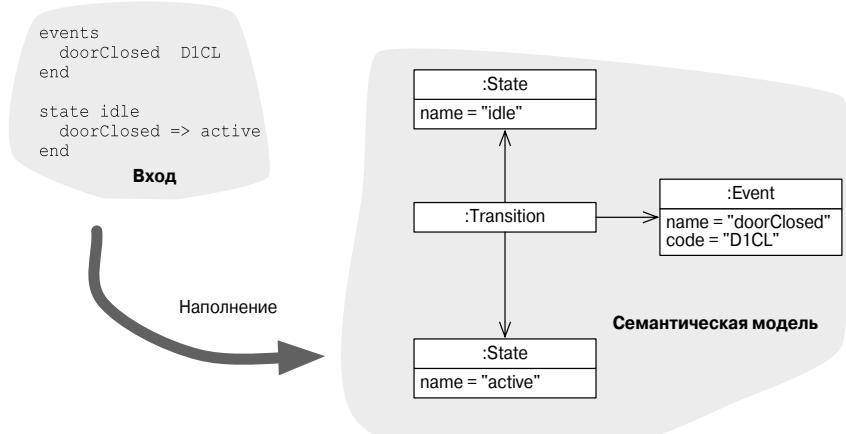


Рис. 1.4. Синтаксический анализ DSL заполняет семантическую модель (*Semantic Model* (171))

(Обратитесь к описанию Semantic Model (171), чтобы разобраться в отличиях между семантической и предметной моделями, а также между семантической моделью и абстрактным синтаксическим деревом.)

Одна из мыслей, которые я хочу до вас донести, — семантическая модель является жизненно важной частью хорошо спроектированного предметно-ориентированного языка. В реальных условиях вы обнаружите, что одни DSL используют семантическую модель, а другие — нет. Но мое категорическое мнение таково: вы почти всегда должны использовать семантическую модель. (Я считаю почти совершенно невозможным употреблять некоторые слова, такие как “всегда”, не предваряя их словом “почти”. Дело в том, что я почти никогда не встречал правил без исключений...)

Я выступаю за семантическую модель, поскольку она обеспечивает четкое отделение синтаксического анализа языка от результирующей семантики. В результате я могу рассматривать работу конечного автомата, совершенствовать его и отлаживать, ничуть не беспокоясь о языковых вопросах. Я могу запускать тесты на модели конечного автомата, заполняя его с помощью интерфейса командных запросов. Я могу независимо развивать модель конечного автомата и DSL, создавая функциональные возможности модели до их предоставления пользователям посредством предметно-ориентированного языка. Возможно, наиболее важным моментом является то, что я могу тестировать модель независимо от языка. Как вы видели ранее, все приведенные примеры DSL были построены на основе одной и той же семантической модели и создавали одну и ту же конфигурацию объектов в этой модели.

В нашем примере семантическая модель является объектной, однако семантическая модель может принимать и другие формы. Это может быть структура данных с поведением, сосредоточенным в отдельных функциях. Я все равно называю это семантической моделью, так как структура данных представляет конкретное значение сценария DSL в контексте этих функций.

С этой точки зрения DSL выступает просто в качестве механизма для выражения конфигурации модели. Многие преимущества данного подхода связаны не с предметно-ориентированным языком, а с моделью. Тот факт, что я могу легко настроить для клиента новый конечный автомат, является свойством модели, а не свойством предметно-ориентированного языка. Тот факт, что я могу внести изменения в контроллер без компиляции во время выполнения, — это возможность модели, а не предметно-ориентированного языка. Тот факт, что я могу повторно использовать код на нескольких контроллерах, — это свойство модели, а не предметно-ориентированного языка. Таким образом, предметно-ориентированный язык — всего лишь тонкий декоративный слой на массивном здании модели.

Модель дает много преимуществ и без всяких предметно-ориентированных языков. В результате мы постоянно используем модели. Мы применяем библиотеки и программные платформы, чтобы избежать излишней работы. При разработке собственного программного обеспечения мы создаем свои модели и абстракции, которые позволяют нам программировать быстрее. Хорошие модели, опубликованные в виде библиотек или платформ или просто обслуживающие наш собственный код, могут прекрасно работать без каких-либо предметно-ориентированных языков в поле зрения.

Однако те же предметно-ориентированные языки могут расширить возможности модели. Правильный DSL облегчает понимание того, что делает конкретный конечный автомат. Некоторые DSL позволяют настраивать модели во время выполнения программы. Таким образом, DSL являются полезным дополнением к некоторым моделям.

Преимущества DSL особо важны для конечного автомата, представляющего собой тип модели, содержание которой эффективно действует в качестве программы системы. Для того чтобы изменить поведение конечного автомата, необходимо изменить объекты в его модели и их взаимосвязи. Этот стиль модели часто называют адаптивной моделью

(Adaptive Model (478)). В результате получается система, в которой различия между кодом и данными стираются, так как для того, чтобы понять поведение конечного автомата, недостаточно просто просмотреть код; необходимо также разобраться, как экземпляры объектов связаны между собой. Конечно, в определенной степени это верно всегда, так как любая программа даст различные результаты при разных входных данных; однако в данном случае это особенно важно, потому что наличие объектов состояния изменяет поведение системы в значительно большей степени.

Адаптивные модели могут быть очень мощными, но они зачастую сложны в использовании, поскольку люди не могут видеть код, определяющий конкретное поведение. Ценность DSL определяется тем, что он обеспечивает явный способ представления этого кода в форме, которая воспринимается как программирование конечного автомата.

Аспектом конечного автомата, делающим его так хорошо подходящим для адаптивной модели, является то, что он представляет собой альтернативную модель вычислений. Наши обычные языки программирования предоставляют стандартный способ программирования автомата, хорошо работающий во многих ситуациях. Но иногда нам нужен иной подход, такой как State Machine (517), Production Rule System (503) или Dependency Network (495). Использование адаптивной модели является хорошим способом обеспечить альтернативную модель вычислений, а DSL является хорошим способом проще ее программировать. Далее я опишу несколько альтернативных вычислительных моделей (см. главу 7, “Альтернативные вычислительные модели”, с. 127), чтобы дать вам почувствовать, на что они похожи и как их можно реализовать. Часто можно слышать, как такое применение DSL называется декларативным программированием.

При рассмотрении этого примера я использовал процесс, в котором сначала строится модель, а затем на нее насылаивается DSL, чтобы помочь работать с ней. Я использую этот способ потому, что, по моему мнению, так проще понять, как DSL вписываются в процесс разработки программного обеспечения. Хотя первоначальное построение модели достаточно распространено, такой подход не единственный. В другом сценарии вы общаетесь с экспертами в предметной области и выясняете, что они понимают, что такое конечный автомат. В этом случае вы можете работать с ними над созданием DSL, который они смогут понять. В этом случае DSL и модель строятся одновременно.

1.5. Использование генерации кода

До сих пор в своем рассмотрении я сначала использовал DSL для заполнения семантической модели (Semantic Model (171)), а затем выполнял семантическую модель, чтобы обеспечить требуемое поведение контроллера. Этот подход среди программистов известен как **интерпретация**. При интерпретации некоторого текста мы выполняем его синтаксический анализ и тут же генерируем результат, который хотим получить с помощью анализируемой программы. (Термин “интерпретация” у программистов может означать разные понятия, но я буду использовать его сугубо для обозначения описанной формы немедленного выполнения исходного текста.)

В мире языков программирования альтернативой интерпретации является компиляция. В случае **компиляции** выполняется синтаксический анализ некоторого исходного текста программы и генерируется промежуточный вывод, который затем обрабатывается отдельно, чтобы обеспечить требуемое поведение. В контексте DSL компиляция обычно именуется **генерацией кода**.

Довольно сложно ясно выразить это отличие на примере конечного автомата, так что давайте воспользуемся еще одним маленьким примером. Представьте себе, что у меня есть какие-то правила для получения страховки. Пусть одно из них имеет вид требования

к возрасту — между 21 и 40 годами. Это правило может быть выражено с помощью DSL и обрабатываться при проверке ряда кандидатов на получение страховки.

В случае интерпретации в процессе работы процессор анализирует правила и загружает семантическую модель, возможно, при запуске программ. При проверке кандидата для него запускается семантическая модель и получается тот или иной результат (рис. 1.5).

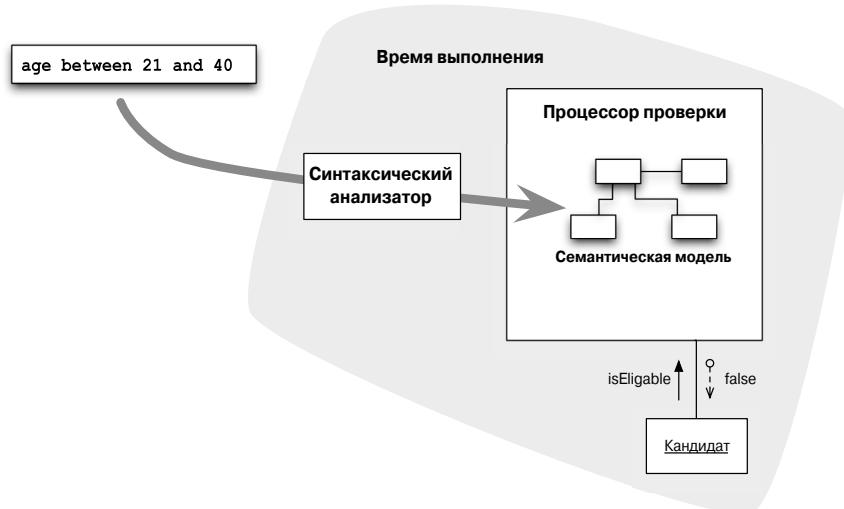


Рис. 1.5. Интерпретатор проводит синтаксический анализ исходного текста и выполняет действия в едином процессе

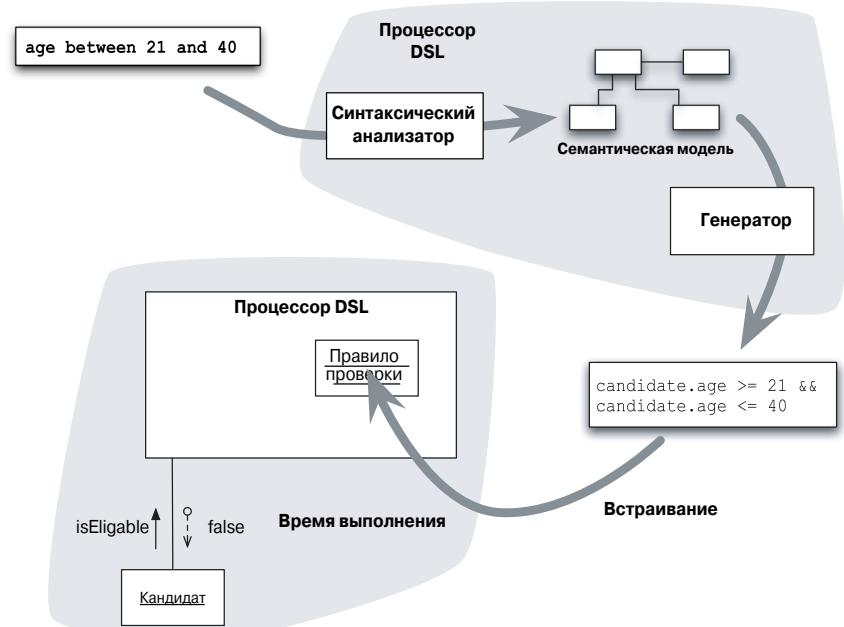


Рис. 1.6. Компилятор проводит синтаксический анализ исходного текста и генерирует некоторый промежуточный код, который затем передается другому процессу для выполнения

В случае компиляции синтаксический анализатор загружает семантическую модель как часть процесса сборки процессора проверки. Во время сборки процессор DSL производит некоторый код, который компилируется, упаковывается и встраивается в процессор, возможно, в виде совместно используемой библиотеки. Этот промежуточный код будет затем запускаться для оценки каждого конкретного кандидата (рис. 1.6).

Наш пример конечного автомата использовал интерпретацию: мы анализируем конфигурационный код и осуществляем наполнение семантической модели во время выполнения программы. Но мы могли бы вместо этого сгенерировать некоторый код, что позволило бы избежать встраивания кода анализатора и модели в устройство.

Генерация кода часто неудобна тем, что заставляет делать дополнительный шаг компиляции. Чтобы создать свою программу, сначала нужно скомпилировать платформу состояний и синтаксический анализатор, после — запустить этот анализатор для генерации исходного кода для контроллера мисс Грант, а затем — скомпилировать сгенерированный код. Это существенно усложняет процесс сборки.

Однако важным преимуществом генерации кода является то, что нет особых причин для генерации кода на том же языке программирования, который использован для синтаксического анализатора. В этом случае можно избежать второй компиляции путем генерации кода для динамического языка, такого как JavaScript или JRuby.

Генерация кода полезна также при использовании DSL с языковой платформой, у которой нет инструментов для поддержки DSL. Разработать систему безопасности для некоторых старых устройств, которые понимают лишь скомпилированный C, можно с помощью генератора кода, который использует наполненную семантическую модель в качестве входных данных и генерирует код на языке программирования C, который затем может быть скомпилирован для работы в старых устройствах. Я встречался с такими проектами, в которых генерируется код для MathCAD, SQL и COBOL.

Во многих работах по DSL внимание сосредоточено на генерации кода, вплоть до того, что он является основным предметом рассмотрения. В результате имеется много статей и книг, в которых превозносятся преимущества генерации кода. Однако, на мой взгляд, генерация кода — это просто механизм реализации, который в большинстве случаев не является действительно необходимым. Конечно, имеется множество ситуаций, когда следует прибегать к генерации кода, но еще больше ситуаций, когда она излишня.

Использование генерации кода распространено, когда синтаксический анализ входного текста и генерация кода выполняются без помощи семантической модели. Хотя это достаточно часто встречающийся способ работы генерирующих код предметно-ориентированных языков, я бы не рекомендовал его к применению, разве что в очень простых случаях. Семантическая модель позволяет разделить синтаксический анализ, семантику выполнения и генерацию кода. Такое разделение делает решение проблемы гораздо более простым. Оно также позволяет изменить свое решение, например вы можете заменить внутренний DSL внешним DSL без изменения процедур генерации кода. Точно так же можно легко сгенерировать несколько выходов без усложнения синтаксического анализатора. Вы можете также использовать с той же семантической моделью как модель интерпретации, так и генерацию кода.

В результате на протяжении большей части своей книги я буду предполагать наличие семантической модели, являющейся основным объектом приложения усилий DSL.

Обычно я сталкиваюсь с двумя стилями использования генерации кода. Один заключается в создании кода при “первом проходе”, который затем, как ожидается, будет использоваться в качестве шаблона, изменяемого вручную. Второй стиль обеспечивает “неприкасаемость” кода, т.е. он не будет модифицироваться вручную, кроме разве что некоторой трассировки в процессе отладки. Я почти всегда предпочитаю последний вариант, потому что он допускает свободную регенерацию кода. Это особенно верно в слу-

чае DSL, так как мы хотим использовать DSL для первичного представления логики, определяемой предметно-ориентированным языком. Это означает, что когда мы хотим изменить поведение, мы должны быть в состоянии легко изменить DSL. Следовательно, необходимо обеспечить отсутствие вмешательства в код вручную, хотя этот код может вызывать код, написанный вручную, и быть вызван таким кодом.

1.6. Использование языковых инструментальных средств

Два показанных к этому моменту стиля DSL — внутренний и внешний — являются традиционным способом классификации предметно-ориентированных языков. Возможно, они не столь распространены, как должны были бы быть, но они имеют долгую историю и достаточно широкое применение, а также известный и хорошо разработанный инструментарий, с которого можно начать работу над соответствующими DSL.

Однако имеется совершенно новая категория инструментов, которая может существенно изменить ситуацию: я называю их языковыми **инструментальными средствами** (*language workbenches*). Такое инструментальное средство представляет собой среду, разработанную для того, чтобы помочь программисту в создании DSL вместе со средствами их эффективного применения.

Одним из серьезных недостатков использования внешнего DSL является относительный ограниченный инструментарий. Большинство программистов не идут далее настройки подсветки синтаксиса в текстовом редакторе. И хотя, исходя из простоты DSL и малых размеров сценариев, можно утверждать, что этого достаточно, имеются аргументы и в пользу сложных инструментов, которыми обычно оснащены современные интегрированные среды разработки. Языковые инструментальные средства позволяют легко создавать не только синтаксический анализатор для предметно-ориентированного языка, но и среду для редактирования программ на этом языке.

Действительно интересным аспектом языковых инструментальных средств является то, что они позволяют создателю DSL выйти за рамки традиционного текстового редактирования кода. Наиболее очевидным примером является поддержка диаграммных языков, которые позволили бы описать конечный автомат непосредственно с помощью диаграммы переходов между состояниями.

Такой инструмент позволяет не только определять диаграммные языки, но и рассматривать DSL-сценарий с различных точек зрения. На рис. 1.7 показана не только диаграмма, но и списки состояний и событий, а также таблица для ввода кодов событий (которые могут быть удалены из диаграммы, чтобы не создавать излишней путаницы).

Такой тип многопанельной визуальной среды редактирования доступен в большом количестве инструментов, но требуются огромные усилия, чтобы построить что-то подобное для себя. Однако одно из достоинств языковых инструментальных средств в том и заключается, что они позволяют легко решать подобные задачи; в самом деле, я довольно быстро сумел создать нечто подобное показанному на рис. 1.7 уже при первом знакомстве с инструментом MetaEdit. Этот инструмент позволяет определить семантическую модель (*Semantic Model* (171)) для конечного автомата, графические и табличные редакторы на рис. 1.7 и написать на основе семантической модели генератор кода.

Однако несмотря на бесспорную привлекательность подобных инструментов, многие разработчики подозрительно относятся к таким “мышевозно-украшательским” инструментам. Есть определенные соображения, по которым имеет смысл текстовое представление источника. В результате это направление возглавляют другие инструменты, обеспечивающие такие возможности, как синтаксически ориентированное редактирование, автодополнение и т.п.

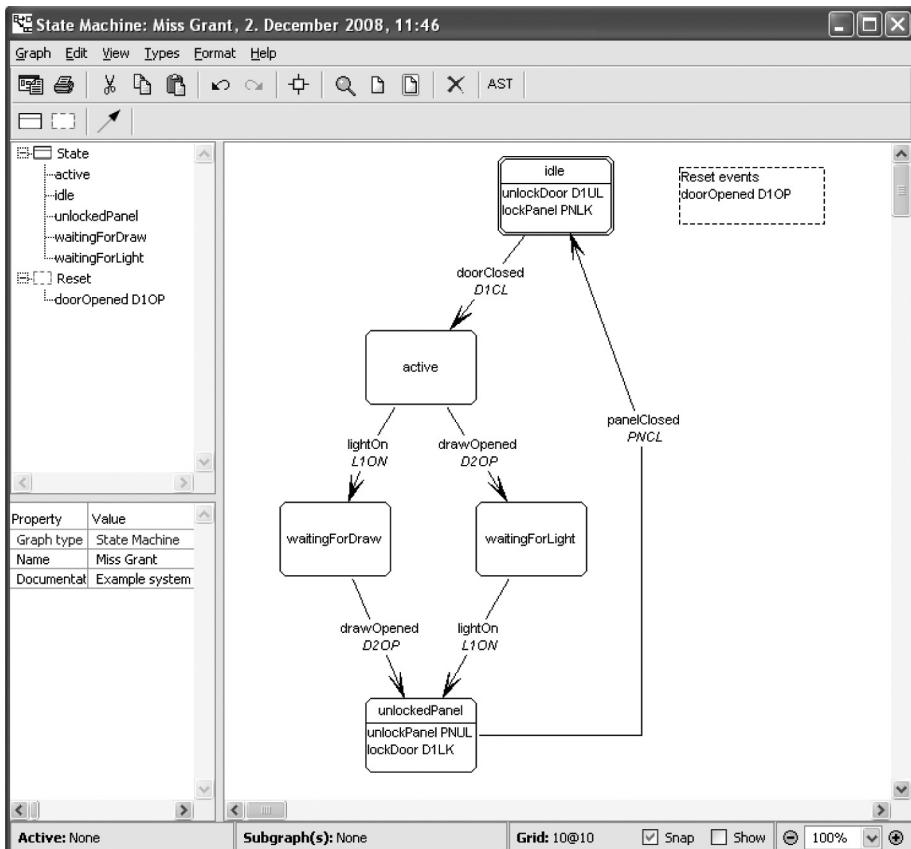


Рис. 1.7. Конечный автомат системы безопасности в языковом инструментальном средстве MetaEdit (источник: MetaCase)

Одним из преимуществ языковых инструментальных средств является то, что они позволяют программировать не программистам (я часто напоминаю, что именно это было первоначальной целью COBOL). Стоит также упомянуть об одной среде программирования, которая оказалась чрезвычайно успешной в плане предоставления инструментов программирования для не программистов, — об электронных таблицах.

Многие не рассматривают электронные таблицы как среду программирования, однако можно утверждать, что они являются наиболее успешной средой программирования из известных в настоящее время. В качестве среды программирования электронные таблицы имеют некоторые интересные особенности. Одной из них является тесная интеграция инструментов в среде программирования. Здесь нет понятия независимого от инструмента текстового представления, которое обрабатывается синтаксическим анализатором. Инструменты и язык тесно взаимосвязаны и спроектированы как одно целое.

Еще одним интересным элементом является то, что я называю **илюстративным программированием**. Взглянув на электронную таблицу, можно увидеть не формулы, которые выполняют вычисления, а числа, формирующие пример расчета. Эти цифры являются иллюстрацией того, что делает выполняемая программа. В большинстве языков программирования главной является программа, а результат ее работы виден только тогда,

когда выполняется ее тестовый запуск. В электронной же таблице главным является вывод, и программа видна только после щелчка на одной из ячеек.

Иллюстративное программирование как концепция не избаловано вниманием. Тем не менее именно оно делает электронные таблицы доступными для непрофессиональных программистов. Оно не лишено и недостатков. Так, отсутствие внимания к структуре программы приводит к большому количеству дублирований (“программирование путем копирования и вставки”) и плохо структурированным программам.

Языковые инструментальные средства поддерживают разработку новых видов программных платформ наподобие рассмотренной. Я думаю, что производимые ими DSL, пожалуй, ближе к электронным таблицам, чем к предметно-ориентированным языкам, какими мы их обычно представляем (и о которых я рассказываю в этой книге).

Я считаю, что языковые инструментальные средства имеют большой потенциал. Реализовав его, они смогут полностью изменить наши представления о разработке программного обеспечения. Однако это все еще дело будущего. Пока же языковые инструментальные средства только начинают свой путь, так что в этой области постоянно появляются новые подходы, а старые инструменты по-прежнему являются предметом глубокой эволюции. Вот почему я предпочитаю не углубляться в эту тему в данной книге. Я думаю, что за время жизни этой книги инструментальные средства успеют претерпеть существенные изменения. Тем не менее у меня есть глава и о них, так как я считаю, что нельзя упускать из виду это достаточно перспективное направление.

1.7. Визуализация

Одним из больших преимуществ языковых инструментальных средств является то, что они позволяют использовать более широкий спектр представлений DSL, в частности — графические представления. Однако даже в случае текстового DSL можно получить представление в виде диаграммы. Это было показано в начале данной главы. На рис. 1.1 вы могли заметить, что диаграмма не такая аккуратная, какими обычно являются мои диаграммы. Дело в том, что я ее не создал “с нуля”, а сгенерировал автоматически из семантической модели (*Semantic Model (171)*) контроллера мисс Грант. Мои классы конечного автомата могут не только выполнять, но и визуализироваться с помощью языка DOT.

Язык DOT является частью пакета Graphviz, который представляет собой инструмент с открытым исходным кодом, позволяющий описывать структуры математических графов (узлы и ребра), а затем автоматически их вычерчивать. Вы просто указываете узлы и ребра графа, какие фигуры использовать для его вывода, а также другие подсказки, и он определяет, как следует изобразить этот граф.

Такой инструмент, как Graphviz, чрезвычайно полезен для многих видов DSL, потому что дает еще одно представление. Подобная **визуализация** похожа на сам DSL тем, что позволяет человеку лучше понять модель. Визуализация отличается от источника тем, что она не редактируема. С другой стороны, она может сделать то, что не в состоянии сделать редактируемая форма, например вывести такую диаграмму.

Визуализации не обязаны иметь графический вид. Я часто использую простые текстовые визуализации, помогающие при отладке во время работы над синтаксическим анализатором. Я знаком с людьми, создающими визуализации в Excel, чтобы было легче общаться с экспертами предметной области. Когда тяжелая работа по созданию семантической модели закончена, добавить визуализацию очень просто. Нужно отметить, что визуализация создается на основе модели, а не DSL, так что ее можно выполнить, даже не используя для заполнения модели DSL.

Глава 2

Использование предметно-ориентированных языков

Теперь, ознакомившись с примерами в предыдущей главе, вы должны хорошо себе представлять, что такое DSL, хотя общее определение пока что не дано. (Еще несколько примеров приведено в главе 10, “Зоопарк DSL”, на с. 159.) Сейчас мы перейдем к этому определению и рассмотрим преимущества и недостатки предметно-ориентированных языков. Это нужно сделать пораньше, чтобы обеспечить определенный контекст для изучения вопросов их реализации в следующей главе.

2.1. Определение предметно-ориентированных языков

“Предметно-ориентированный язык” является полезным термином и концепцией, но имеет очень размытые границы. Об одних вещах можно с уверенностью говорить, что это DSL, а о других можно говорить и так, и эдак. Как и в большинстве случаев в области программного обеспечения, этот термин никогда не имел строгого определения. Но я думаю, что для этой книги такое определение имеет важное значение. Итак...

Предметно-ориентированный язык (сущ.) — это язык программирования с ограниченными выразительными возможностями, ориентированный на некую конкретную предметную область.

В этом определении есть четыре ключевых элемента.

- **Язык программирования.** DSL используется людьми для постановки задачи компьютеру. Как и структура любого современного языка программирования, его структура призвана облегчить понимание языка человеком, но при этом он должен выполняться компьютером.
- **Природа языка.** DSL является языком программирования и как таковой должен давать ощущение свободы выразительных возможностей, происходящее не только из отдельных выражений, но и из способа их соединения.
- **Ограниченные выразительные возможности.** Язык программирования общего назначения предоставляет множество возможностей: поддержку разнообразных данных, управления и абстрактных структур. Все это полезно, но делает язык сложным в освоении и использовании. DSL поддерживает минимум возможностей, необходимых для поддержки своей предметной области. Вы не можете построить всю программную систему с помощью DSL; вместо этого DSL используется для одного конкретного аспекта этой системы.

- **Ориентированность на предметную область.** Ограниченный язык полезен только в том случае, если имеет четкую направленность на небольшую предметную область. Ориентированность на предметную область — вот что придает ценность такому языку.

Обратите внимание на то, что ориентированность на предметную область оказалась последней в списке и является лишь следствием ограниченных выразительных возможностей. Многие используют буквальное определение DSL как язык для конкретной предметной области. Но буквальные определения часто неправильны: мы не называем монеты компакт-дисками, хотя это типичные диски, которые, конечно, более компактны, чем те, к которым мы применяем данный термин.

Я делю предметно-ориентированные языки на три основные категории: внешние DSL, внутренние DSL и языковые инструментальные средства.

- **Внешний DSL** представляет собой язык, отделенный от основного языка приложения, с которым он работает. Как правило, внешний DSL имеет пользовательский синтаксис, но при этом достаточно широко применяется синтаксис другого языка (частым выбором является XML). Сценарий во внешнем DSL, как правило, анализируется кодом принимающего приложения с помощью методов синтаксического анализа текста. Это стиль традиционных малых языков Unix. Примерами внешних DSL, с которыми вы, вероятно, сталкивались, являются регулярные выражения, SQL, Awk, а также XML-файлы конфигурации для систем наподобие Struts и Hibernate.
- **Внутренний DSL** — это специфический способ использования языка общего назначения. Сценарий во внутреннем DSL представляет собой корректный код языка общего назначения, который использует только подмножество возможностей этого языка в определенном стиле, чтобы работать с одним небольшим аспектом всей системы. Результат должен иметь вид исходного текста пользовательского, а не базового языка. Классическим примером этого стиля является Lisp; программисты часто говорят о программировании на Lisp как о создании и использовании DSL. Ruby также имеет разработанную высокую культуру DSL: многие библиотеки Ruby сделаны в стиле DSL. В частности, самая известная платформа Ruby — Rails — часто рассматривается как набор DSL.
- **Языковые инструментальные средства** представляют собой специализированные интегрированные среды разработки для определения и создания DSL. В частности, языковые инструментальные средства используются не только для определения структуры DSL, но и как пользовательские среды редактирования для людей, которые будут писать сценарии DSL. Получающиеся в результате сценарии тесно объединяют среду редактирования и язык.

С годами вокруг этих трех стилей сформировались определенные сообщества. Вы можете найти программистов, которые имеют большой опыт работы с внутренними DSL, но не знают, как строить внешние. На мой взгляд, это представляет собой проблему, поскольку в результате люди не могут выбрать наиболее подходящий инструмент для работы. Я помню разговор с членами команды, которая применяла очень сложные методы обработки внутреннего DSL для поддержки пользовательского синтаксиса, что, я уверен, было бы гораздо проще сделать с помощью внешнего DSL. Но поскольку они не знают, как строить внешние DSL, эта возможность для них закрыта. Наличие в данной книге как внутренних, так и внешних DSL очень важно, поскольку так вы получите всестороннюю информацию. (Более того, здесь есть даже информация об языковых инструментальных средствах, хотя в силу того, что эта область в настоящее время интенсивно развивается, изложена она более схематично.)

Другим взглядом на DSL является взгляд как на один из способов работы с абстракцией. При разработке программного обеспечения мы сначала строим абстракции, а затем работаем с ними, зачастую на нескольких уровнях. Наиболее распространенный способ построения абстракций — путем реализации библиотеки или платформы; наиболее распространенный способ работы с ними — посредством вызовов API. С этой точки зрения DSL является интерфейсом библиотеки, предоставляющим иной стиль манипуляций API. В этом контексте библиотека представляет собой семантическую модель (*Semantic Model* (171)) DSL. Следствием этого является то, что такие DSL, как правило, следуют за библиотеками. Я, кроме того, считаю, что семантическая модель является необходимым дополнением к хорошо построенному DSL.

Если послушать то, что говорят о DSL, можно решить, что построение DSL — тяжелый труд. На самом деле, как правило, тяжело построить модель; DSL представляет собой просто слой поверх нее. Конечно, чтобы создать хорошо работающий DSL, требуются усилия, но они, как правило, гораздо меньше, чем в случае создания базовой модели.

2.1.1. Границы DSL

Как я уже говорил, DSL представляют собой концепцию с размытыми границами. Хотя я не думаю, что кто-то будет спорить с тем фактом, что регулярные выражения представляют собой DSL, есть много иных достаточно спорных случаев. Поэтому я считаю нужным упомянуть некоторые из них здесь (это поможет получить более полное представление о том, что такое DSL).

Каждый стиль DSL имеет свои граничные условия, поэтому я буду рассматривать их по отдельности. При обсуждении не стоит забывать, что отличительными характеристиками DSL являются природа языка, предметная область и ограниченные выразительные возможности. Как оказывается, ориентированность на предметную область не является хорошим граничным условием — границы чаще связаны с ограниченными выразительными возможностями и природой языка.

Я начну с внутренних DSL. Здесь возникает вопрос о границе между внутренними DSL и обычным API командных запросов. Во многих отношениях внутренний DSL — это не более чем причудливый API (как когда-то было замечено в старой лаборатории Белла, “дизайн библиотеки есть дизайн языка”). На мой взгляд, главное отличие кроется в природе языка. Майк Робертс (Mike Roberts) предположил, что API командных запросов определяет словарь абстракции, в то время как внутренние DSL добавляют грамматику.

Распространенный способ документирования класса с API командных запросов состоит в перечислении всех его методов. При этом каждый метод должен иметь смысл сам по себе. У вас есть список “слов”, каждое из которых имеет некоторый самодостаточный смысл. Методы внутреннего DSL часто имеют смысл только в контексте большего выражения DSL. В приведенном ранее примере внутреннего DSL для Java у меня был метод `to`, указывающий целевое состояние перехода. Это плохое имя метода в случае API, но зато отлично вписывающееся в фразу типа `.transition(lightOn).to(unlockedPanel)`. В результате код внутреннего DSL имеет вид предложения, а не последовательности не связанных команд.

Ограниченные выразительные возможности для внутренних DSL, очевидно, не являются фундаментальным свойством языка, поскольку язык внутреннего DSL является языком общего назначения. В этом случае ограниченные выразительные возможности связаны со способом использования этого языка. При формировании выражения DSL вы ограничиваете сами себя небольшим подмножеством всех возможностей языка программирования общего назначения. Обычно в этом случае стремятся избегать условных

конструкций, циклов и переменных. Пирс Коли (Piers Cawley) называет такой язык “суржиком базового языка”.

В случае внешнего DSL имеется четкая граница между предметно-ориентированным языком и языком программирования общего назначения. Языки могут быть ориентированы на предметную область и при этом оставаться языками общего назначения. Хорошим примером в данном случае является R, язык и платформа для статистики; он имеет ярко выраженную направленность на работу со статистикой, но при этом имеет выразительные возможности универсального языка программирования. Таким образом, несмотря на его ориентированность на предметную область, я бы не назвал его предметно-ориентированным языком.

Более очевидным DSL являются регулярные выражения. Здесь ориентированность на предметную область (соответствие текста шаблону) сочетается с ограниченными возможностями, достаточными только для проверки соответствия текста. Одним из общих показателей DSL является то, что это не полный по Тьюрингу язык. Предметно-ориентированные языки обычно избегают обычных императивных структур управления (условия и циклы), не имеют переменных и не могут определять подпрограммы.

Многие могут не согласиться со мной и, используя буквальное определение DSL, будут утверждать, что такие языки, как R, должны рассматриваться как предметно-ориентированные. Поэтому я сделал сильный акцент на ограниченности выразительных возможностей как на важном отличии между DSL и языками общего назначения. Ограниченностю выразительных возможностей придает DSL иные характеристики как в плане их использования, так и в плане реализации. Это приводит к отличию представлений о DSL по сравнению с языками общего назначения.

Давайте рассмотрим XSLT. Предметной областью XSLT является преобразование XML-документов, но у него имеются все возможности, которых можно было бы ожидать от обычного языка программирования. Я думаю, что в этом случае способ применения языка важнее, чем сам язык. Если XSLT используется для преобразования XML, то это DSL. Однако если он используется для решения задачи о восьми ферзях, то это язык общего назначения. Конкретное применение языка может разместить его по обе стороны границы.

Еще одной границей в случае внешних DSL являются сериализованные структуры данных. Является ли список присваивания свойств (`color = blue`) в конфигурационном файле предметно-ориентированным языком? Я думаю, что здесь граничное условие — природа языка. Ряду присваиваний недостает гибкости, поэтому он не соответствует указанному критерию.

Аналогичное рассуждение применимо ко многим конфигурационным файлам. Многие современные среды обеспечивают возможность выполнять часть программирования через некоторые конфигурационные файлы, часто с использованием XML-синтаксиса. Во многих случаях эти конфигурационные файлы в формате XML являются DSL. Однако это не всегда справедливо. Иногда XML-файлы должны быть созданы с помощью других инструментов, так что формат XML применяется для сериализации и не предназначен для использования людьми. В этом случае, поскольку люди не будут его применять, я бы не стал классифицировать его как DSL. Конечно, полезно иметь формат хранения, который может прочесть человек, как минимум для целей отладки. Но вопрос не в том, является ли информация удобочитаемой для человека, а в том, будет ли такое представление основным способом взаимодействия человека с данным аспектом системы.

Одна из самых больших проблем, связанных с этими видами конфигурационных файлов, состоит в том, что, даже если они не предназначены для редактирования человеком, в конечном итоге на практике они являются основным механизмом редактирования. В такой ситуации XML становится предметно-ориентированным языком случайно.

При использовании языковых инструментальных средств граница проходит между ними и приложениями, которые позволяют пользователю проектировать собственные структуры данных и формы (наподобие Microsoft Access). В конце концов, можно взять модель состояний и представить ее в структуре реляционной базы данных (я видел идеи и похожие). После этого вы можете создавать формы для работы с моделью. Есть два вопроса: является ли Access языковым инструментальным средством и является ли результат вашей работы предметно-ориентированным языком?

Начну со второго вопроса. Поскольку мы строим конкретное приложение для конечного автомата, у нас есть и ориентированность на предметную область, и ограниченность выразительных возможностей. Критическим становится вопрос о природе языка. Если мы помещаем данные в формы и сохраняем их в таблице, то это обычно не реальный язык, каким мы его себе представляем. Таблица может быть выражением природы языка: и FIT (см. раздел “10.6. FIT” на с. 166), и Excel используют табличное представление и оба представляют собой языки (я считаю FIT предметно-ориентированным языком, а Excel — языком общего назначения). Но большинство приложений не пытаются достичь такой гибкости — они просто создают формы и окна, которые никак не подчеркивают внутренние взаимосвязи. Некоторые же приложения позволяют создавать макеты диаграмм для определения взаимосвязей элементов в стиле MetaEdit.

Что касается вопроса о том, является ли Access языковым инструментальным средством, то я бы хотел вернуться к назначению проекта. Access не должен был быть языковым инструментальным средством, хотя при желании вы можете использовать его таким образом. Посмотрите, как много людей используют Excel в качестве базы данных, хотя он предназначался совсем не для этого.

В более широком смысле является ли человеческий жаргон, например “феня”, предметно-ориентированным языком? Он имеет ограниченные выразительные возможности, ориентированность на предметную область, подобие грамматики, а также словарь. Он выходит за рамки моего определения, потому что я использую понятие “предметно-ориентированный язык” только для компьютерных языков. Если бы мы реализовали компьютерный язык для понимания выражений на фене, это был бы действительно DSL, но со словами обычного разговорного языка. Я использую термин **предметный язык** (domain language) для обозначения предметно-ориентированного естественного, человеческого языка, а термин **предметно-ориентированный язык** (Domain-Specific Language — DSL) — для языков компьютерных.

Итак, к чему привело это обсуждение границ DSL? Одно совершенно ясно — имеется очень мало резко очерченных границ. Разумные люди могут не согласиться с определением DSL. Проверка природы языка или ограниченности выразительных возможностей сама по себе является очень расплывчатой, поэтому следует ожидать столь же расплывчатых результатов тестирования. Кроме того, не все согласятся со сформулированными мною граничными условиями.

В этой дискуссии я сознательно обошел молчанием многие вопросы, но это не означает, что я не считаю их важными. Цель определения заключается в том, чтобы помочь общению разных людей — чтобы они говорили об одном и том же и употребляли одни и те же термины. Я считаю, что приведенное определение DSL помогает читателям более эффективно определять применимость и назначение описываемых в книге технологий.

2.1.2. Фрагментарные и автономные DSL

Конечный автомат системы безопасности, использованный мною в качестве примера, является автономным DSL. Под этим я подразумеваю, что вы можете взглянуть на блок сценария DSL (как правило, это один файл), и это весь DSL. Если вы знакомы

с DSL, но не базовым языком программирования, использованным для приложения, вам нужно понять, что делает этот код DSL, потому что базовый язык в этом коде либо полностью отсутствует (в случае внешнего DSL), либо подавлен внутренним DSL.

Другой способ появления кода DSL — в фрагментарной форме. В этом случае маленькие фрагменты DSL используются внутри кода на базовом языке. Вы можете рассматривать их как усовершенствования базового языка программирования с дополнительными возможностями. В этом случае вы не в состоянии следить за тем, что делает DSL, без понимания базового языка.

Для внешнего DSL хорошим примером фрагментарного предметно-ориентированного языка являются регулярные выражения. В вашей программе не найдется файла регулярных выражений, но небольшие фрагменты регулярных выражений будут встречаться в обычном коде приложения постоянно. Другим примером является SQL, который часто используется в виде инструкций SQL в контексте большей программы.

Аналогичные фрагментарные подходы используются и с внутренним DSL. Особенно плодотворная область применения внутреннего DSL — модульное тестирование. Популярной языковой возможностью фрагментарного внутреннего DSL являются аннотации (*Annotation* (439)), позволяющие добавлять метаданные к программным элементам базового кода. Это делает аннотации подходящими для фрагментарных DSL, но бесполезными для автономных.

Одни и те же DSL могут использоваться как в автономных, так и в фрагментарных контекстах; хорошим примером этого является SQL. Одни DSL предназначены для использования в фрагментарной форме, другие — в автономной форме, а третьи — и там, и здесь.

2.2. Зачем используется DSL

Теперь, я надеюсь, мы существенно продвинулись в понимании того, что же такое DSL. Следующий вопрос — почему нужно рассматривать возможность его применения?

Предметно-ориентированные языки являются инструментом с ограниченным назначением. Они не похожи на объектную ориентированность или на гибкую технологию разработки, которые внесли фундаментальные изменения в наше представление о разработке программного обеспечения. DSL являются всего лишь очень узкоспециализированным инструментом для очень специфичных ситуаций. В типичном проекте может использоваться полдесятка DSL в различных местах — и так в действительности и бывает.

В разделе “Языки и семантическая модель” (с. 41) я неоднократно напоминал, что DSL является тонким слоем над моделью, где модель может представлять собой библиотеку или каркас. Эта фраза должна напомнить вам, что всякий раз, когда вы думаете о преимуществах (или недостатках) DSL, важно отделять преимущества, предоставляемые моделью, от преимуществ DSL. Путать их — распространенная ошибка.

DSL имеют потенциал для реализации определенных преимуществ. Планируя использовать DSL, следует взвесить эти преимущества и решить, какие из них возможны в ваших обстоятельствах.

2.2.1. Повышение производительности разработки

Самое привлекательное в DSL то, что он предоставляет средства для более ясного выражения назначения части системы. Прочитав определение контроллера мисс Грант в форме DSL, вы наверняка согласитесь, что оно проще для понимания, чем код с API командных запросов.

Эта ясность — не просто эстетическая прихоть. Чем проще читается фрагмент кода, тем легче найти в нем ошибки и модифицировать систему. Таким образом, по той же причине, по которой мы стремимся использовать значимые имена переменных, документацию и понятные конструкции кода, нам следует поощрять и использование DSL.

Люди часто недооценивают влияние дефектов на производительность. Дефекты не только отвлекают от внешнего контроля качества программного обеспечения, но и замедляют работу разработчиков, заставляя тратить время на отладку и исправления. Ограниченные выразительные возможности DSL усложняют написание неправильного кода и облегчают поиск сделанных ошибок.

Сама модель уже обеспечивает значительное повышение производительности. Она позволяет избежать дублирования, собирая вместе общий код; но прежде всего она предоставляет абстракцию, которая позволяет быстрее и понятнее указать, что именно происходит в приложении. DSL усиливает эту возможность, предоставляя более выразительные средства, чем абстракция. DSL может помочь в использовании API, поскольку он переносит внимание на то, как именно должны быть скомбинированы различные методы API.

Интересным примером этому является использование DSL для “обертки” неудобных библиотек сторонних производителей. Обычные преимущества DSL, заключающиеся в более свободном интерфейсе, усиливаются, когда интерфейс командных запросов остается желать лучшего. Кроме того, DSL может значительно уменьшить объем материала, который должны изучать клиенты данного программного обеспечения.

2.2.2. Общение с экспертами в предметной области

Я считаю, что самая сложная часть программных проектов и самая распространенная причина неудач — общение с заказчиками и пользователями программного обеспечения. Предоставляя ясный и точный язык для общения с предметной областью, DSL может помочь улучшить это общение.

Это преимущество более тонкое, чем простой вопрос повышения производительности. Начнем с того, что многие DSL не подходят для общения, например те же регулярные выражения или зависимости при сборке не очень сюда вписываются. Только часть автономных DSL действительно применима к этому каналу общения.

Когда люди говорят о DSL в этом контексте, часто звучит мнение “Теперь мы избавимся от программистов и заставим специалистов самих указывать правила”. Я называю это “COBOL-заблуждением”, поскольку именно это ожидалось от COBOL. Это распространенный аргумент, но я не думаю, что он становится весомее от постоянного повторения.

Несмотря на COBOL-заблуждение, я думаю, что DSL все же могут улучшить общение. Это не значит, что эксперты в предметной области будут сами писать DSL, но они смогут прочесть его исходные тексты и, таким образом, понять, что делает система. Будучи в состоянии прочитать код DSL, эксперты в предметной области смогут найти допущенные ошибки. Они также смогут более эффективно общаться с программистами, которые пишут правила, возможно, смогут даже сделать какие-то грубые наброски, которые смогут быть переработаны в соответствующие правила DSL.

Я не говорю, что эксперты в предметной области никогда не должны сами писать исходные тексты на DSL. Я сталкивался со слишком многими случаями, когда команде удавалось успешно привлекать экспертов к написанию важных фрагментов поведения на DSL. Но я все же думаю, что наибольший выигрыш от использования DSL возникает тогда, когда эксперты начинают читать исходные тексты. Сосредоточение на чтении может оказаться первым шагом на пути к написанию на DSL, с тем преимуществом, что вы ничего не теряете, даже если не предпринимаете этот последующий шаг.

Мое внимание к DSL как к чему-то предназначенному для чтения предметными экспертами приводит к аргументу против использования DSL. Чтобы эксперты понимали содержимое семантической модели (*Semantic Model* (171)), можете просто визуализировать эту модель. Стоит задуматься, не является ли визуализация более эффективной, чем поддержка DSL. И как минимум полезно иметь визуализацию в дополнение к DSL.

Привлечение экспертов к DSL очень похоже на их участие в построении модели. Я считаю, что построение модели вместе с экспертами приносит большую пользу; создание *единого языка* (*ubiquitous language*) [7] углубляет связи между разработчиками программного обеспечения и экспертами в предметной области. DSL обеспечивает еще один способ их общения. В зависимости от обстоятельств вы можете привлекать экспертов как к участию в создании модели и DSL, так и только к работе с DSL.

Действительно, некоторые люди считают, что пытаться описывать предметную область, используя DSL, очень полезно, даже если DSL никогда не будет реализован. Это может быть полезно по крайней мере в качестве платформы для общения.

Экспертов в предметной области трудно привлечь к разработке DSL, но если этого удастся достичь, отдача будет весьма высокой. И даже если вам не удастся привлечь экспертов, вы все равно можете получить достаточный выигрыш в производительности труда разработчиков, оправдывающий затраченные на DSL усилия.

2.2.3. Изменения в контексте выполнения

Когда мы говорим о том, почему мы могли бы захотеть выразить свой конечный автомат в XML, одна из веских причин этого — определение может быть обработано во время выполнения, а не во время компиляции. Такого рода причины, когда нам требуется выполнение кода в другой среде, — достаточно распространенное основание для использования DSL. В случае конфигурационных файлов в формате XML обычной причиной является перенос логики со времени компиляции на время выполнения.

Есть и другие полезные изменения в контексте выполнения. В одном из проектов, с которыми я сталкивался, требовалось проходить по базам данных в поисках контрактов, соответствующих определенным условиям, и помечать их. Разработчики для указания этих условий писали специализированный DSL и использовали его для заполнения семантической модели (*Semantic Model* (171)) в Ruby. Считывание всех контрактов в память для выполнения логики запроса в Ruby было бы слишком медленным, но команда могла использовать представление семантической модели для генерации SQL для обработки запроса базой данных. Писать правила непосредственно на SQL было слишком трудно даже для разработчиков, не говоря уже о бизнесменах. Однако бизнесмены могли читать (а в данном случае и писать) соответствующие выражения в DSL.

Использование DSL наподобие описанного зачастую может преодолевать ограничения базового языка, позволяя выразить требования в удобном DSL-виде, а затем сгенерировать код для реальной среды выполнения.

Модель может облегчить такой перенос. Если у вас есть модель, можете либо выполнить ее непосредственно, либо сгенерировать код на ее основе. Модели могут быть заполнены как из интерфейса в стиле форм, так и из DSL. У DSL имеется несколько преимуществ по сравнению с формами. Так, DSL часто представляют сложную логику лучше форм. Кроме того, можно использовать одни и те же инструменты управления кодом (например, системы управления версиями) для управления правилами. Когда правила вводятся через форму и хранятся в базе данных, контролем версий часто пренебрегают.

Описанное далее относится к ложным преимуществам DSL. Иногда говорят, что хорошим свойством DSL является то, что он обеспечивает одно и то же поведение в различных языко-

вых средах. Можно написать бизнес-правила, которые генерируют код C# и Java, или описать проверки, которые будут работать на C# на сервере и на JavaScript на стороне клиента. Это ложное преимущество, поскольку вы можете добиться этого только с помощью модели, без DSL. DSL может облегчить понимание этих правил, но это уже отдельный вопрос.

2.2.4. Альтернативные вычислительные модели

Подавляющее большинство задач решается с помощью императивной модели вычислений. Это означает, что мы говорим компьютеру, что и в какой последовательности он должен делать, поток управления обрабатывается с помощью условных операторов и циклов, у нас есть переменные, т.е. масса вещей, которые мы считаем само собой разумеющимися. Императивная модель вычислений стала популярной, потому что ее сравнительно легко понять и применить к множеству задач. Однако это не всегда лучший выбор.

Хорошим примером являются конечные автоматы. Мы можем написать императивный код и условные операторы для обработки этого типа поведения — и этот код может быть весьма хорошо структурирован. Но зачастую более полезно рассматривать его как конечный автомат. Другим распространенным примером является определение, в какой последовательности следует строить программное обеспечение. Вы можете делать это с помощью императивной логики, но через некоторое время большинство людей признают, что легче воспользоваться сетью зависимостей (*Dependency Network* (495)), например для выполнения тестов должен быть скомпилирован последний вариант кода. В результате языки, предназначенные для описания процесса построения (например, Make и Ant), используют зависимости между задачами в качестве основного механизма структурирования.

Часто такие неимперативные подходы называются декларативным программированием. Смысл заключается в том, что при этом описывается, *что* должно быть сделано, в отличие от работы через императивные инструкции, которые описывают, *как* должна быть выполнена работа.

Вам не нужен DSL для того, чтобы использовать альтернативные модели вычислений. Основное поведение альтернативных вычислительных моделей определяется семантической моделью (*Semantic Model* (171)), как проиллюстрировано на примере конечного автомата. Однако DSL может иметь большое значение, так как облегчает работу с декларативными языками, наполняющими семантическую модель.

2.3. Проблемы с DSL

Выяснив, когда лучше использовать DSL, имеет смысл задуматься и о том, когда использовать DSL не следует (или по крайней мере о проблемах, связанных с их использованием).

По сути, единственная причина, по которой не стоит использовать DSL, — если вы не видите никаких преимуществ в применении DSL в своей ситуации, или по крайней мере вы не видите преимуществ, которые стоят затрат сил и времени на создание предметно-ориентированного языка.

Даже когда DSL применимы, они создают проблемы. Вообще говоря, я думаю, что их проблематичность в настоящее время завышается, как правило, потому, что люди недостаточно хорошо знакомы с построением DSL и с тем, как они вписываются в более широкую картину разработки программного обеспечения. Кроме того, многие указываемые программистами проблемы с DSL вытекают из той же путаницы между DSL и моделями, которая препятствует в достижении ряда преимуществ DSL.

Масса проблем с DSL специфична для конкретных стилей DSL, так что, чтобы разобраться в этих вопросах, необходимо иметь более глубокое понимание того, как эти DSL

реализованы. Поэтому оставим обсуждение этих проблем на более позднее время и рассмотрим только наиболее общие проблемы, связанные с материалом, который мы в настоящее время обсуждаем.

2.3.1. Языковая какофония

Наиболее широко распространено возражение против применения DSL, заключающееся в том, что я называю **проблемой языковой какофонии**: озабоченность тем, что языки трудно изучать, так что использовать многие языки будет гораздо сложнее, чем один. Необходимость знать несколько языков усложняет работу с системой и введение в проект новых людей.

Обычно, когда люди говорят об этой проблеме, у них имеется пара заблуждений. Прежде всего, они часто ошибочно сравнивают усилия по обучению DSL с усилиями по обучению языку общего назначения. DSL гораздо проще языка общего назначения, а следовательно, его гораздо легче выучить.

Многие критики понимают это, но по-прежнему возражают против DSL, полагая, что, даже если DSL относительно легко выучить, большое количество предметно-ориентированных языков в проекте затрудняет понимание происходящего в нем. Заблуждение здесь в том, что эти критики забывают, что проект всегда имеет такие сложные области, которые трудно изучить. Даже если у вас нет DSL, обычно есть много абстракций, которые вы должны понимать. Как правило, для упрощения работы с ними эти абстракции организованы в виде библиотек, так что, даже если вы и не должны изучать предметно-ориентированные языки, вам все равно придется выучить несколько библиотек.

Так что на самом деле вопрос ставится иначе — насколько труднее выучить DSL по сравнению с базовой моделью самой по себе? Я бы сказал, что дополнительные затраты на обучение DSL достаточно малы по сравнению с затратами на понимание модели. Действительно, так как весь смысл DSL — в упрощении понимания модели и управления ею, наличие DSL должна снизить стоимость обучения.

2.3.2. Стоимость построения

У DSL могут быть малые дополнительные затраты на лежащую в его основе библиотеку, но они есть. Требуется код, который сначала следует создать, а затем поддерживать. И, как и любой код, он требует внимания. Не каждая библиотека получает выгоду от наличия DSL-обертки вокруг нее. Если API командных запросов делает свою работу очень хорошо, то нет смысла в добавлении дополнительного API поверх него. Даже если DSL может помочь, иногда он требует слишком больших усилий на создание и поддержку.

Поддержка DSL является важным фактором. Даже с простым внутренним DSL могут возникнуть проблемы, если большая часть команды разработчиков считает его трудным для понимания. Внешние DSL добавляют массу дополнительных частей в процесс разработки, в том числе синтаксические анализаторы, которые часто просто пугают разработчиков.

Один из факторов, повышающих стоимость добавления DSL, — люди не привыкли их создавать, и им приходится изучать новые методы. Хотя затраты на обучение не следует игнорировать, нужно помнить, что они будут компенсированы тем, что в дальнейшем можно будет использовать DSL везде, где потребуется.

Помните также, что стоимость создания DSL связана со стоимостью построения модели. Любая сложная область нуждается в механизме управления этой сложностью, и если последняя достаточно велика, чтобы задуматься о применении DSL, то это почти наверняка говорит о выгоде построения модели. Предметно-ориентированные языки могут помочь в работе над моделью и снизить стоимость ее создания.

Это приводит к связанному вопросу, не приведет ли поощрение DSL к созданию множества плохих языков? Да, я думаю, что могут быть созданы плохие DSL так же, как существует множество библиотек с плохим API командных запросов. Вопрос в том, будет ли DSL еще хуже. Хороший предметно-ориентированный язык может послужить отличной оберткой для плохой библиотеки и упростить работу с ней (хотя я предпочел бы по возможности исправить саму библиотеку). Плохой DSL приведет лишь к пустой трате времени на его создание и поддержку, но то же самое можно сказать о любом плохом коде.

2.3.3. Язык гетто

Эта проблема отлична от проблемы языковой какофонии. В данном случае имеется в виду компания, которая построила ряд своих систем на базе внутреннего языка, не используемого в другом месте. Это усложняет поиск новых сотрудников и использование технологических новинок.

Я начну анализ этого аргумента вот с чего: если вы пишете целые системы на некотором языке, значит, он является языком программирования общего назначения, а не предметно-ориентированным языком (по крайней мере, в моем определении). Хотя вы можете использовать многие из методов DSL для построения языков общего назначения, я очень настоятельно рекомендую не делать этого. Создание и поддержание языка общего назначения — очень серьезное дело, которое потребует от вас большого труда и приговорит вас к “жизни в гетто”. Не делайте этого.

Я думаю, есть пара реальных вопросов, вытекающих из проблемы языка гетто. Первый из них заключается в том, что у предметно-ориентированного языка всегда есть опасность случайно превратиться в язык общего назначения. Вы берете DSL и постепенно добавляете к нему новые возможности. Сегодня это условные выражения, завтра — циклы, и в какой-то момент вы получаете полный по Тьюрингу язык программирования.

Единственная защита против этого — решительный отказ от такой эволюции. Убедитесь, что DSL сфокусированы на узких проблемах предметной области. Ставьте под вопрос любые новые возможности, которые представляются вам выходящими за эти узкие рамки. Если вам необходимо сделать больше, рассмотрите возможность использования нескольких языков, комбинируя их вместо того, чтобы позволить чрезмерно разрастись одному DSL.

Та же проблема может погубить и программную платформу. Хорошая библиотека характеризуется четким представлением о ее предназначении. Если ваша библиотека для работы с ценами на товары включает в себя реализацию протокола HTTP, вы страдаете от той же проблемы — неспособности разделить задания.

Второй вопрос заключается в том, что для построения своего здания не следует самому становиться кирпичным заводом. Это относится к библиотекам в той же мере, в какой и к предметно-ориентированным языкам. Мое общее правило при разработке программного обеспечения гласит: если что-то — не ваше дело, лучше не пишите это сами, а посмотрите, нельзя ли взять этот код откуда-то. В частности, с ростом количества систем с открытым исходным кодом часто имеет смысл выполнить расширение существующей системы, а не писать собственный код “с нуля”.

2.3.4. Ограниченнная абстракция

Достоинство DSL заключается в предоставлении абстракции, которую можно использовать при рассмотрении предметной области. Такая абстракция действительно очень ценна; она позволяет выразить поведение предметной области гораздо проще, чем при ее рассмотрении в терминах конструкций низкого уровня.

Однако любая абстракция, будь то DSL или модель, всегда несет в себе опасность установки ограничителей на ваше мышление. При использовании такой “зашоренной” абстракции вы тратите больше усилий на сопоставление реальности и абстракции. Вы сами увидите это, когда столкнетесь с чем-то, что не вписывается в абстракцию, и потратите много времени, пытаясь подогнать реалии под модель, вместо того чтобы поступить наоборот и изменить абстракцию так, чтобы она могла справиться с новым поведением.

Эта проблема характерна для множества абстракций, не только для DSL, но DSL может эту проблему усугубить. Поскольку DSL обеспечивает более удобный способ манипулирования абстракцией, программисты неохотно идут на внесение в него любых изменений. Ситуация еще больше усугубляется при использовании DSL экспертами предметной области, которые с еще большей неохотой относятся к изменениям привычной абстракции.

Как и в случае любой абстракции, на DSL следует смотреть как на нечто развивающееся, а не окончательно застывшее.

2.4. Более широкая обработка языка

Эта книга посвящена не только предметно-ориентированным языкам, но и методам обработки лингвистической информации. Эти темы пересекаются, так как 90% методов обработки лингвистической информации в усредненной команде разработчиков применяются для DSL. Однако эти методы могут не менее успешно использоваться и для решения ряда других задач, так что я не могу не обсудить некоторые из них.

Я столкнулся с отличным примером сказанному при посещении команды, работающей над проектом ThoughtWorks. Перед ними стояла задача связи с системой стороннего производителя путем пересылки сообщений, полезная нагрузка которых определялась тетрадями COBOL. Тетради COBOL представляют собой формат структуры данных для записей. Их было очень много, поэтому мой коллега Брайан Эгг (Brian Egge) решил построить анализатор для подмножества синтаксиса тетрадей COBOL, который генерировал Java-классы для взаимодействия с этими записями. Созданный анализатор мог легко обработать столько тетрадей, сколько требовалось; никакая иная часть кода ничего не знала о структурах данных COBOL, и любые изменения могли быть обработаны с помощью простой регенерации. Было бы большой натяжкой называть тетради COBOL предметно-ориентированным языком, но здесь были применены те же базовые технологии, которые применяются и для внешних DSL.

То, что я говорю о разных методах в контексте DSL, не означает, что их нельзя применять для решения других проблем. Научившись обрабатывать лингвистическую информацию, вы сможете использовать их самыми разными способами и в разных областях.

2.5. Жизненный цикл DSL

В этой вводной части я представил DSL, сначала описав платформу и ее API командных запросов, а затем добавив слой DSL над API для упрощения работы с последним. Я использовал этот подход, потому что он представляется мне более простым для понимания DSL, но это не единственный способ использования DSL на практике.

Распространенная альтернатива состоит в первоначальном определении DSL. В этом режиме вы начинаете с некоторых сценариев и пишете их так, как, по вашему представлению, должен выглядеть ваш DSL. Если язык является частью функциональности предметной области, то это лучше делать при участии эксперта в данной области — это хороший первый шаг в использовании DSL в качестве среды общения.

Одним людям нравится начинать работу с инструкций, которые должны быть синтаксически корректными. Это означает, что для внутреннего DSL они будут придерживаться синтаксиса базового языка. Для внешнего DSL они напишут инструкции, которые они точно смогут проанализировать. Другие люди в начале работы более неформальны и предпринимают второй проход для создания разумного синтаксиса.

Таким образом, разрабатывая (в нашем случае) конечный автомат, следует поработать вместе с людьми, которые понимают потребности клиентов. Вы можете прийти с множеством примеров поведения контроллера, основанных либо на пожеланиях клиентов в прошлом, либо на ваших представлениях о будущих желаниях. Для каждого из них вы можете написать их в некотором DSL-виде. По ходу работы с разными примерами вы будете изменять DSL, обеспечивая поддержку новых возможностей. К концу работы у вас будет представительная выборка ситуаций и псевдоDSL-описание каждой из них.

Если вы используете языковое инструментальное средство, то на данном этапе вам придется обойтись без него, ограничившись текстовым или графическим редактором или даже обычными ручкой и бумагой.

Получив представительное множество фрагментов псевдоDSL, вы можете начать реализацию. Реализация включает в себя разработку модели конечного автомата в базовом языке, API командных запросов для этой модели, конкретный синтаксис DSL и перевод между DSL и API командных запросов. Люди делают это по-разному. Одни предпочитают мелкие шаги на всех этапах: добавить немного в модель, изменить DSL для обработки добавления, проверить корректность сделанного. Другие предпочитают создавать и тестировать платформу целиком, а затем наращивать над ней слой DSL. Третьим удобнее начать с DSL, построить библиотеку, а затем “срастить” их в единое целое. Будучи “инкременталистом” по природе, я предпочитаю добавлять функциональность небольшими ломтиками, так что я отношусь к первым из трех описанных типов разработчиков.

Я мог бы начать с простейших из известных мне случаев. Я бы запрограммировал библиотеку, которая может поддерживать такой простой случай, с помощью методологии разработки, управляемой тестами. Затем я взял бы DSL и реализовывал его, пытаясь применить его к построенной платформе. Я хотел бы внести определенные изменения в DSL, которые упростили бы его построение, хотя я запустил бы эти изменения, но только после консультаций с экспертами в предметной области, чтобы убедиться в том, что мы работаем в одной среде общения. После того как я добился бы работы одного контроллера, я перешел бы к следующему. И так постепенно я развивал бы и тестировал платформу, а затем вносил бы изменения в DSL.

Это не означает, что не следует начинать с разработки модели. Более того, зачастую это как раз отличный выбор. Обычно он используется, когда вы сначала не думаете об использовании DSL или не уверены, что вы в нем нуждаетесь. Таким образом, вы сначала строите платформу, работаете с ней некоторое время, а затем решаете, что DSL будет полезным к ней дополнением. В этом случае у вас может иметься работающая и используемая многими клиентами модель конечного автомата. Затем вы понимаете, что так сложнее добавлять новых клиентов, и решаете попробовать DSL.

Вот несколько подходов, которые можно использовать для наращивания DSL над моделью. Метод “засевания” потихоньку строит DSL поверх модели, рассматривая ее в основном как черный ящик. Мы можем начать с рассмотрения имеющихся у нас контроллеров и набросать для каждого из них псевдоDSL. Затем мы будем реализовывать DSL сценарий за сценарием, в основном так же, как и в предыдущем случае. Обычно при этом никакие глубокие изменения в модель не вносятся, хотя я бы приветствовал добавление в модель методов для поддержки DSL.

При таком подходе к модели сначала добавляются гибкие методы, чтобы упростить настройку модели, а затем они постепенно превращаются в полноценный DSL. Этот

подход в большей степени ориентирован на внутренний DSL; вы можете рассматривать его как серьезный рефакторинг модели для получения внутреннего DSL. Привлекательным аспектом модели “засеваания” является ее “постепенность”, не требующая значительных затрат для создания DSL.

Конечно, вы не всегда знаете, есть ли у вас каркас. Вы можете создать несколько контроллеров и только потом понять, что у них много общих функциональных возможностей. В этом случае я реорганизовал бы систему, отделяя модель и код конфигурации. Такое разделение является жизненно важным шагом. Хотя при решении этой задачи я и мог бы иметь в виду DSL, я все же склонен, в первую очередь, выполнить указанное разделение, а уже затем наращивать сверху DSL.

Здесь я хотел бы обратить ваше внимание на еще один момент. Убедитесь, что все ваши сценарии DSL находятся под контролем той или иной системы управления версиями. Сценарий DSL становится частью вашего кода, а потому должен быть под контролем системы управления версиями, как и все остальное. Самое замечательное в текстовых DSL — они отлично работают с такими системами, что позволяет ясно отслеживать изменения в поведении вашей системы.

2.6. Что такое хороший дизайн DSL

Просматривая эту книгу, люди часто просят дать совет по созданию хорошего дизайна языка. В конце концов, дизайн языка — дело сложное, и хотелось бы избежать появления плохих языков. Я хотел бы дать хороший совет, но, признаюсь, у меня не хватает ясного представления об этом.

Главная цель DSL, как и любого текста, — ясность для читателя. Вы хотите, чтобы ваш типичный читатель, который может быть как программистом, так и экспертом в предметной области, был в состоянии понять, что означают инструкции DSL, причем как можно быстрее и точнее. Я не уверен, что могу много рассказать о том, как этого достичь, но я думаю, что важно не забывать об этой цели во время работы.

Я поклонник итеративного дизайна, и DSL не является исключением. Попробуйте воспользоваться знаниями о своей целевой аудитории. Будьте готовы представить несколько альтернатив и посмотреть на реакцию на них. Хороший язык создается путем проб и ошибок. Не бойтесь допустить неверный шаг: чем больше поворотов вы сделаете, тем больше среди них будет правильных и тем скорее вы найдете верный путь.

Не бойтесь использовать жargon предметной области в DSL и в своей семантической модели (*Semantic Model* (171)). Если пользователи DSL знакомы с жаргоном, они должны видеть его и в DSL. Жаргон может помочь налаживанию контактов с предметной областью, даже если он звучит для посторонних как бред.

Используйте распространенные соглашения из повседневной жизни. Если все вы используете Java или C#, примените для комментариев символы “//”, а для любой иерархической структуры — “{ ” и ” } ”.

Я думаю, что следует сделать еще одно предостережение: не пытайтесь сделать DSL столь же удобочитаемым, как и естественный язык. Такие попытки предпринимались неоднократно для языков общего назначения (наиболее очевидный пример — AppleScript). Беда в том, что такие попытки приводят к появлению языка, который затрудняет понимание семантики. Помните, что DSL является языком программирования, так что его использование должно восприниматься, как программирование с большими краткостью и точностью, которыми языки программирования отличаются от естественных языков. Попытка придать языку программирования вид естественного языка создает неверный контекст; при написании программы всегда нужно помнить, что вы находитесь в среде языка программирования.

Глава 3

Реализация предметно-ориентированных языков

Теперь, когда вы хорошо себе представляете, что такое DSL и в чем преимущества его использования, наступило время углубиться в методы его построения. Хотя многие из этих методов различны для внутренних и внешних DSL, у них много общего. В этой главе я сконцентрируюсь на общих для внутренних и внешних предметно-ориентированных языков вопросах и перейду к подробностям в следующей главе. Я также пока что проигнорирую языковые инструментальные средства (я вернусь к ним гораздо позже).

3.1. Архитектура обработки DSL

Возможно, одна из наиболее важных вещей, о которых следует поговорить, — это как работают реализации предметно-ориентированных языков (то, что можно назвать архитектурой системы DSL; рис. 3.1).

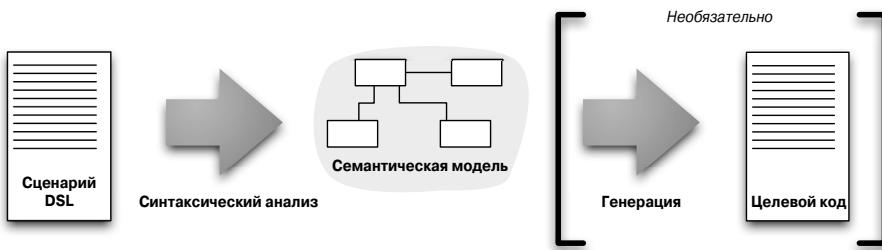


Рис. 3.1. Предпочитаемая автором архитектура обработки DSL

К настоящему времени вам, должно быть, ужасно надоело мое постоянное повторение, что DSL представляет собой тонкий слой над моделью. Говоря о модели в этом контексте, я именую ее шаблоном семантической модели (*Semantic Model* (171)). Ее суть в том, что все важное семантическое поведение охватывается моделью, а роль DSL — заполнить эту модель посредством этапа синтаксического анализа. Это означает, что семантическая модель играет центральную роль в представлении о DSL (почти во всей этой книге предполагается ее использование). Об альтернативах семантической модели я скажу в конце этого раздела, когда появится достаточный контекст для их обсуждения.

Будучи сторонником объектно-ориентированного программирования, я, естественно, предполагаю, что семантическая модель является объектной. Мне нравятся богатые объектные модели, в которых сочетаются данные и обработка. Но семантиче-

ская модель не обязана быть такой; она также может представлять собой только структуры данных. Хотя я всегда предпочитаю по возможности работать с объектами, использование семантической модели в форме данных лучше, чем отсутствие семантической модели вообще. Таким образом, хотя я буду считать в этой книге, что у нас имеются надлежащие поведенческие объекты, помните, что вы можете встретиться и только со структурами данных.

Во многих системах используется модель предметной области (**Domain Model** [10]) для охвата базового поведения программной системы. Часто DSL заполняет значительную часть модели предметной области. Я предпочитаю разделять понятия модели предметной области и семантической модели. Семантическая модель DSL обычно является подмножеством модели предметной области приложения, так как не все части модели предметной области наилучшим образом обрабатываются DSL. Кроме того, DSL могут быть использованы для выполнения иных задач, помимо заполнения модели предметной области (при ее наличии).

Семантическая модель — это обычная объектная модель, с которой можно работать точно так же, как и с любой иной имеющейся в вашем распоряжении объектной моделью. В случае примера с состояниями можно заполнить конечный автомат с помощью API командных запросов модели, а затем запустить его, чтобы получить требуемое поведение. В известном смысле модель независима от DSL, хотя на самом деле это близкие родственники.

(Если вы знакомы с технологиями компиляторов, у вас может возникнуть вопрос, не является ли семантическая модель тем же, что и абстрактное синтаксическое дерево. Краткий ответ — нет, это разные понятия. Я еще вернусь к этой теме в разделе “Работа синтаксического анализатора”, с. 67.)

Отделение семантической модели от DSL имеет несколько преимуществ. Основное из них то, что вы можете думать о семантике предметной области, не беспокоясь о синтаксисе DSL или синтаксическом анализаторе. Если вы вообще используете DSL, то, как правило, потому что вы работаете с чем-то довольно сложным, иначе вы бы его не использовали. А так как это что-то довольно сложное, то оно заслуживает собственной модели.

В частности, это позволяет тестировать семантическую модель путем создания объектов в модели и непосредственной работы с ними. Я могу создать много состояний и переходов и протестировать их, чтобы посмотреть, насколько корректно работают события и команды, не прибегая к синтаксическому анализу вообще. Если при этом возникли проблемы в работе конечного автомата, я могу выделить проблему в модели, не понимая, как работает синтаксический анализ.

Явная семантическая модель позволяет поддерживать несколько предметно-ориентированных языков для ее заполнения. Вы можете начать с простого внутреннего DSL, а затем добавить внешний DSL в качестве более простого для чтения альтернативного варианта. Поскольку у вас уже имеются написанные на внутреннем DSL сценарии и работающие с ним пользователи, вы можете сохранить существующий внутренний DSL и поддерживать два предметно-ориентированных языка. Поскольку у них одна семантическая модель, это не так трудно и, кроме того, помогает избежать дублирования между языками.

В продолжение темы укажу, что отдельная семантическая модель позволяет развивать модель и язык в отдельности. Если я хочу изменить модель, я могу исследовать этот вопрос без изменения DSL, добавив необходимые конструкции в предметно-ориентированный язык после того, как получуирующую модель. Я также могу экспериментировать с новым синтаксисом для DSL и просто убедиться, что он создает те же объекты в модели. Я могу сравнить два синтаксиса по заполнению семантической модели.

Во многом это разделение семантической модели и синтаксиса DSL отражает разделение модели предметной области и представления, которые мы видим в разработке корпоративного программного обеспечения. Зачастую я думаю о DSL как об еще одной форме пользовательского интерфейса.

Сравнение DSL с представлением накладывает определенные ограничения. Предметно-ориентированный язык и семантическая модель по-прежнему связаны, и при добавлении в DSL новых конструкций нужно обеспечить их поддержку в семантической модели, что зачастую означает одновременное изменение их обоих. Однако разделение означает, что я могу анализировать семантические вопросы отдельно от вопросов синтаксического анализа, что упрощает мою задачу.

Разница между внутренними и внешними DSL заключается в стадии синтаксического анализа — как в том, что именно анализируется, так и в том, как это делается. Оба стиля DSL будут производить семантическую модель одного вида, и, как я уже говорил ранее, нет никаких причин, чтобы не позволить одной семантической модели быть заполненной и внутренними, и внешними DSL. По сути, это именно то, что я сделал, когда программировал пример конечного автомата (когда у меня было несколько DSL, заполняющих одну семантическую модель).

В случае внешнего DSL имеется четкое разделение между сценариями DSL, синтаксическим анализатором и семантической моделью. Сценарии DSL пишутся на отдельном языке; синтаксический анализатор читает их и заполняет семантическую модель. В случае внутреннего DSL гораздо проще все это перепутать. Я сторонник явного слоя объектов (*Expression Builder* (349)), работа которых заключается в предоставлении необходимых свободных интерфейсов, действующих в качестве языка. Затем сценарии DSL выполняются путем применения методов построителя выражений, которые заполняют семантическую модель. Таким образом, в случае внутреннего DSL синтаксический анализ сценариев DSL выполняется комбинацией синтаксического анализатора базового языка и построителей выражений.

В этой связи возникает интересный момент — вам может показаться немного странным использование термина “синтаксический анализ” в контексте внутреннего DSL. Признаюсь, мне это тоже не совсем нравится. Но я пришел к выводу, что проводить параллели между внутренним и внешним DSL весьма полезно. При использовании традиционных методов синтаксического анализа вы получаете поток текста, преобразуете его в дерево синтаксического анализа и обрабатываете это дерево для получения полезного выхода. При синтаксическом анализе внутреннего DSL ваш вход представляет собой ряд вызовов функций. Вы также организуете их в иерархию (обычно неявно в стеке) с целью получения полезного выхода.

Еще одним фактором в пользу использования здесь термина “синтаксический анализ” является то, что в ряде случаев отсутствует непосредственная обработка текста. Во внутреннем DSL текст обрабатывает синтаксический анализатор базового языка, а процессор DSL обрабатывает дальнейшие языковые конструкции. Но то же самое происходит и с предметно-ориентированными языками, использующими XML: синтаксический анализатор XML преобразует текст в элементы XML, после чего с ними работает процессор DSL.

Сейчас стоит вернуться к различиям между внутренними и внешними DSL. Использованное ранее отличие — используется ли для написания сценария базовый язык программирования приложения — как правило, верно, но не на все 100%. Например, рассмотрим случай, когда приложение написано на Java, а сценарии DSL — на JRuby. В этом случае я бы классифицировал DSL как внутренний, по крайней мере по тому признаку, что вам нужно использовать приемы из части этой книги, посвященной внутренним DSL.

Истинное различие между ними состоит в том, что внутренние DSL пишутся на выполняемом языке, а синтаксический анализ осуществляется путем выполнения DSL в этом языке. Как в JRuby, так и в XML, DSL встраивается в синтаксис носителя, но при этом мы выполняем код JRuby, но только читаем структуры XML-данных. Большую часть времени внутренний DSL выполняется в основном языке приложения, так что это определение в общем случае оказывается более полезным.

Теперь, когда у нас имеется семантическая модель, необходимо заставить ее делать то, что нам нужно. В примере с конечным автоматом это задача управления системой безопасности. Есть два основных способа, как это сделать. Самый простой и обычно наилучший — просто выполнить семантическую модель. Семантическая модель представляется собой код и как таковая может быть запущена и выполнить все необходимые действия.

Другой вариант заключается в использовании генерации кода. Генерация кода означает, что мы создаем код, который отдельно компилируется и запускается. В некоторых кругах генерация кода рассматривается как неотъемлемая часть DSL. Я сталкивался с мнением, что при любой работе с DSL необходимо генерировать код. В редких случаях, когда я вижу кого-то, говорящего или пишущего о генераторах синтаксических анализаторов (Parser Generator (277)), они неизбежно говорят о генерации кода. Однако DSL не присуща необходимость генерации кода! В большинстве случаев лучше просто выполнить семантическую модель.

Всеким доводом в пользу генерации кода является ситуация, когда запуск модели и синтаксический анализ DSL должны выполняться в разных местах. Хорошим примером этого является выполнение кода в среде с ограниченным выбором языка, например в случае ограниченных аппаратных средств или в реляционной базе данных. Вы не хотите запускать синтаксический анализатор в небольшом устройстве или в SQL, поэтому реализуете синтаксический анализатор и семантическую модель на более подходящем языке и генерируете код C или SQL. Еще один случай — при наличии зависимостей синтаксического анализатора от библиотек, которых не должно быть в производственной среде. Это достаточно распространенная ситуация, если вы используете сложный инструментарий для своего DSL, в частности именно поэтому языковые инструментальные средства, как правило, прибегают к генерации кода.

В этих ситуациях все равно полезно иметь семантическую модель, которая может работать без генерации кода. Запустив семантическую модель, можно экспериментировать с выполнением DSL без необходимости разбираться в процессе генерации кода. Вы можете протестировать синтаксический анализ и семантику без генерации кода, что зачастую помогает выполнять тесты быстрее и для отдельных задач. Так вы можете выполнить проверки семантической модели и выявить ее ошибки до генерации кода.

Еще одним аргументом в пользу генерации кода, даже в среде, в которой можно ограничиться непосредственной интерпретацией семантической модели, является то, что многие разработчики считают логику богатых семантических моделей сложной для понимания. Создание кода на основе семантической модели упрощает понимание и может быть решающим моментом для команды из не очень опытных и способных разработчиков.

Но самое важное, о чем следует помнить, — это то, что генерация кода не является обязательным компонентом ландшафта DSL. Это одна из тех вещей, которые очень важны, когда в них есть нужда, но большую часть времени можно легко обходиться и без них. Генераторы кода похожи на снегоступы: зимой при большом количестве снега они необходимы, но никто не носит их в жаркий летний день.

Генерируя код, мы видим еще одно преимущество использования семантической модели: она отделяет генераторы кода от синтаксических анализаторов. Я могу написать генератор кода, ничего не зная о процессе синтаксического анализа, и независимо его про-

верить. Одно это стоит разработки семантической модели. Кроме того, в случае необходимости многоцелевой генерации кода модель упрощает ее поддержку.

3.2. Работа синтаксического анализатора

Итак, различия между внутренними и внешними DSL кроются в их синтаксическом анализе. И в самом деле: между ними много различий, но есть и очень много общего.

Одно из наиболее важных сходств состоит в том, что синтаксический анализ является строго иерархической операцией. Анализируя текст, мы организуем его части в виде деревовидной структуры. Рассмотрим простую структуру списка событий конечного автомата. При использовании внешнего синтаксиса это выглядит примерно так.

```
events
  doorClosed D1CL
  drawerOpened D2OP
end
```

Мы можем рассматривать эту составную конструкцию как список событий, каждое из которых имеет имя и код.

Рассмотрим аналог на внутреннем DSL в Ruby.

```
event :doorClosed "D1CL"
event :drawerOpened "D2OP"
```

Здесь нет явного понятия списка, но каждое событие по-прежнему представляет собой иерархию: событие содержит символьное имя и строку кода.

Всякий раз, когда вы смотрите на сценарий, подобный приведенному, вы можете представить его в виде иерархии. Такая иерархия называется **синтаксическим деревом** (или деревом разбора) (рис. 3.2). Любой сценарий может быть преобразован в множество потенциальных синтаксических деревьев — в зависимости от того, каким образом вы будете его разбирать. Синтаксическое дерево является гораздо более полезным представлением сценария, чем слова, поскольку с ним можно работать различными способами, по-разному обходя дерево.

При использовании семантической модели (*Semantic Model* (171)) мы получаем синтаксическое дерево и преобразуем его в семантическую модель. Если почитать материалы языковых сообществ, то в них можно заметить, что основной упор сделан на синтаксические деревья — либо они выполняются непосредственно, либо на их основе генерируется код. Фактически синтаксическое дерево можно использовать как семантическую модель. Впрочем, по возможности я бы этого не делал, потому что синтаксическое дерево очень тесно связано с синтаксисом сценария DSL и, таким образом, привязывает обработку DSL к его синтаксису.

Я говорю о синтаксическом дереве как если бы это была материальная структура данных в вашей системе наподобие модели XML DOM. Зачастую это вовсе не так. По большей части синтаксическое дерево формируется в стеке вызовов и обрабатывается при его обходе. В результате вы никогда не видите все дерево в целом, а видите только обрабатываемые в настоящий момент ветви (аналогично способу работы SAX с XML документом). Несмотря на это, как правило, полезно рассматривать такое призрачное синтаксическое дерево как скрывающееся в тени стека вызовов. Для внутренних DSL это дерево формируется аргументами в вызове функции (*Nested Function* (361)) и вложенными объектами (*Method Chaining* (375)). Иногда вы не видите строгой иерархии и должны ее имитировать (*Function Sequence* (357)) с иерархией, моделируемой с помощью переменных контекста (*Context Variable* (187)). Синтаксическое дерево может быть призрачным, но при

этом оно остается полезным инструментом. Использование внешнего DSL приводит к более явному синтаксическому дереву; более того, иногда вы действительно создаете структуру данных полномасштабного синтаксического дерева (*Tree Construction* (289)). Но даже внешние DSL обычно обрабатывают синтаксические деревья, формируя и отсекая ветви в стеке вызовов. (Выше я сослался на несколько моделей, которые еще не были описаны. Вы можете спокойно игнорировать их при первом чтении, но позже эти ссылки вам пригодятся.)

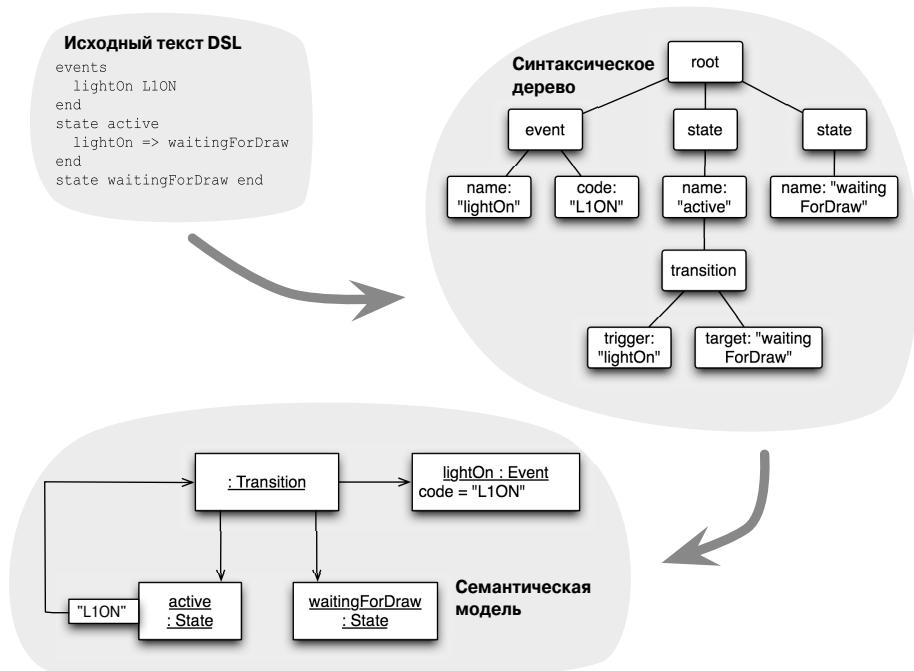


Рис. 3.2. Синтаксическое дерево и семантическая модель обычно являются различными представлениями сценария DSL

3.3. Грамматики, синтаксис и семантики

В работе с синтаксисом языка важным инструментом является грамматика. **Грамматика** представляет собой набор правил, которые описывают, как текстовый поток преобразуется в синтаксическое дерево. Большинство программистов сталкиваются с грамматиками в тот или иной момент своей деятельности, так как они часто используются для описания языков программирования, с которыми мы все работаем. Грамматика состоит из списка производных (правил вывода), в котором каждое правило имеет некий заголовок (нетерминал) и тело (инструкцию вывода). Таким образом, грамматика для инструкции сложения может выглядеть как `additionStatement := number '+' number`. Эта производная говорит о том, что если мы видим предложение языка `5+3`, синтаксический анализатор может распознать его как инструкцию сложения. Правила ссылаются одно на другое, так что у нас должно быть правило, которое говорит о том, как распознать корректное число. Из таких правил и составляется грамматика языка.

Важно понимать, что язык может иметь несколько определяющих его грамматик. Не существует *единственной* грамматики языка. Грамматика определяет структуру синтаксического дерева, которое генерируется для данного языка, и мы можем распознать несколько синтаксических деревьев для некоторого фрагмента исходного текста на этом языке. Грамматика просто определяет одну из форм синтаксического дерева; выбранные вами фактическая грамматика и синтаксическое дерево будут зависеть от многих факторов, включая особенности языка, с которым вы работаете, и то, как вы намерены обрабатывать синтаксическое дерево.

Грамматика также определяет только синтаксис языка, т.е. как он представляется в виде синтаксического дерева. Она ничего не говорит нам о его семантике, т.е. о том, что означает то или иное выражение. В зависимости от контекста 5+3 может означать и 8, и 53; синтаксис при этом один и тот же, а семантики различны. В случае семантической модели (*Semantic Model (171)*) определение семантики сводится к тому, как мы заполняем семантическую модель из синтаксического дерева и что мы делаем с этой семантической моделью. В частности, можно сказать, что если два выражения дают одну и ту же структуру в семантической модели, то они имеют одну и ту же семантику, даже если их синтаксисы различны.

Используя внешний DSL, в частности при синтаксически управляемой трансляции (*Syntax-Directed Translation (229)*), вы, вероятно, явно применяете грамматику для построения синтаксического анализатора. В случае внутреннего DSL явной грамматики нет, но все равно полезно думать о своем предметно-ориентированном языке в терминах грамматики. Эта грамматика помогает выбрать, какой из внутренних шаблонов DSL стоит использовать.

Рассматривая внутренние DSL, мы сталкиваемся с еще одной специфичной для них сложностью — проблема заключается в наличии двух анализирующих проходов и, таким образом, участии двух грамматик. Первый синтаксический анализ — самого базового языка программирования, который, очевидно, зависит от базовой грамматики. Этот анализ создает выполняемые инструкции базового языка. DSL-часть базового языка при выполнении будет создавать “призрачные” синтаксические деревья DSL в стеке вызовов. И только в ходе этого второго синтаксического анализа в игру вступает условная грамматика DSL.

3.4. Анализ данных

В процессе работы анализатор должен хранить различные данные, связанные с проводимым анализом. Эти данные могут представлять собой полное синтаксическое дерево, но в основном это некоторая другая информация. И даже если хранится полное синтаксическое дерево, есть и множество других данных, которые необходимо хранить для эффективной и корректной работы.

Анализ по своей природе представляет собой обход дерева, и всякий раз, когда вы обрабатываете часть сценария DSL, у вас имеются некоторые сведения о контексте, связанном с обрабатываемой вами ветвью синтаксического дерева. Однако зачастую вам нужна и информация, находящаяся за пределами этой ветви. Давайте вновь рассмотрим фрагмент из примера конечного автомата.

```
commands
    unlockDoor D1UL
end

state idle
    actions {unlockDoor}
end
```

Здесь мы видим распространенную ситуацию: команда определяется в одной части языка, а используется — в другой. Когда команда используется как часть действий состояния, мы находимся в ветви синтаксического дерева, отличной от той, где была определена эта команда. Если единственное представление синтаксического дерева находится в стеке вызовов, то определение команды к настоящему времени уже отсутствует. Таким образом, чтобы можно было разрешить ссылку в инструкции действия, следует хранить объект команды для последующего использования.

Для того чтобы сделать это, мы используем таблицу символов (Symbol Table (177)), которая является, по существу, словарем, ключ которого — идентификатор unlockDoor, а значение — объект, представляющий команду в нашем синтаксическом анализе. Обрабатывая текст unlockDoor D1UL, мы создаем объект для хранения соответствующих данных и вносим его в таблицу символов с ключом unlockDoor. Этот объект может быть объектом семантической модели для команды или промежуточным объектом, локальным для синтаксического дерева. Позже, обрабатывая действия {unlockDoor}, мы находим объект с помощью таблицы символов для выяснения взаимосвязей между состоянием и его действиями. Таблица символов, таким образом, является важнейшим инструментом для создания перекрестных ссылок. Если вы действительно создаете полное синтаксическое дерево во время синтаксического анализа, теоретически можно обойтись без таблицы символов, хотя обычно она все равно остается полезной конструкцией, упрощающей сборку программы воедино (рис. 3.3).

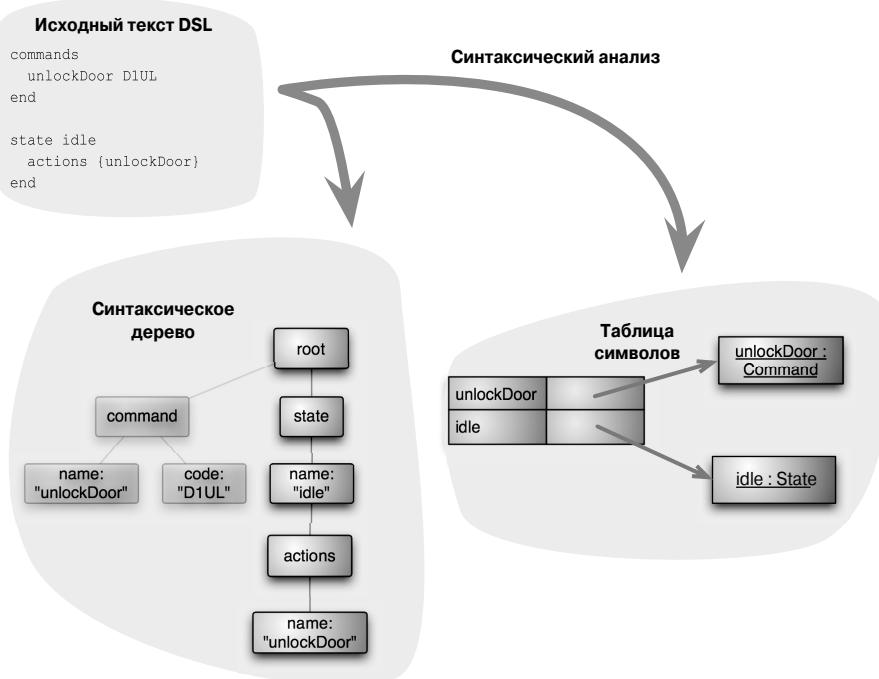


Рис. 3.3. Синтаксический анализ создает как синтаксическое дерево, так и таблицу символов

В конце этого раздела я более подробно остановлюсь на паре шаблонов. Я упоминаю их здесь, потому что они используются и во внутренних, и во внешних DSL, так что это

вполне подходящее место для такого упоминания, хотя в большей части этой главы рассматриваются вопросы на более высоком уровне.

В процессе синтаксического анализа необходимо сохранять его результаты. Иногда все результаты можно внести в таблицу символов, иногда большое количество информации можно хранить в стеке вызовов, а иногда синтаксическому анализатору требуются дополнительные структуры данных. Во всех этих случаях наиболее очевидные действия — создание объектов семантической модели (*Semantic Model* (171)). Зачастую, однако, придется создавать промежуточные объекты из-за невозможности создания объектов семантической модели до некоторого более позднего момента синтаксического анализа. Типичным примером такого промежуточного объекта является построитель конструкции (*Construction Builder* (191)), который представляет собой объект, собирающий все данные для объекта семантической модели. Это полезно, когда ваш объект семантической модели после создания имеет только данные для чтения, но в процессе синтаксического анализа для него накапливаются дополнительные данные. Постройтель имеет те же поля, что и объект семантической модели, но делает их доступными для чтения и записи, так что в них можно сохранять информацию. Как только будут готовы все необходимые данные, можно будет создать объект семантической модели. Использование построителей усложняет анализатор, но я предпочитаю этот путь изменению семантической модели для отказа от преимуществ свойств, доступных только для чтения.

В действительности иногда можно отложить создание объектов семантической модели до окончания обработки всего сценария DSL. В этом случае синтаксический анализ имеет различные фазы: во-первых, чтение сценария DSL и создание промежуточных данных синтаксического анализа; во-вторых, проход по этим промежуточным данным и заполнение семантической модели. Выбор, что именно предстоит сделать в ходе обработки текста, а что отложить, как правило, зависит от того, как должна быть заполнена семантическая модель.

Способ синтаксического анализа выражения часто зависит от контекста. Рассмотрим следующий текст.

```
state idle
  actions {unlockDoor}
end

state unlockedPanel
  actions {lockDoor}
end
```

Обрабатывая действия `{lockDoor}`, важно знать, что при этом мы находимся в контексте состояния `unlockedPanel`, а не пристаиваем. Часто этот контекст получается в процессе построения и обхода синтаксического дерева синтаксическим анализатором, но во многих случаях его трудно получить. Если мы не можем найти контекст путем изучения синтаксического дерева, то хорошим решением может быть хранение контекста, в данной ситуации — текущего состояния, в переменной. Я называю этот вид переменной контекста (*Context Variable* (187)). Такая переменная контекста, как и таблица символов, может хранить объект семантической модели или некоторый промежуточный объект.

Хотя переменная контекста представляет собой очень простой в использовании инструмент, обычно я все же предпочитаю избегать их применения, насколько это возможно. Следовать синтаксическому анализу кода легче, если читать его без необходимости мысленно манипулировать переменными контекста (так же, как большое количество изменяемых переменных усложняют отслеживание процедурного кода). Конечно, есть моменты, когда невозможно избежать использования переменных контекста, но я предпочитаю их избегать.

3.5. Макросы

Макросы (*Macros* (195)) являются инструментом, который может использоваться как с внутренними, так и с внешними DSL. В свое время они применялись довольно широко, но сегодня стали существенно менее распространенными. В большинстве контекстов я предлагаю обходиться без них, но иногда они все же полезны, так что следует выяснить, как они работают и когда их можно использовать.

Макросы бывают двух видов: текстовые и синтаксические. Проще всего понять текстовые макросы, которые позволяют заменить какой-нибудь текст некоторым другим текстом. Хороший пример, где они могут оказаться удобными, — указание цвета в CSS-файле. В CSS цвета указываются с помощью числовых кодов, таких как #FFB595 (кроме некоторых фиксированных именованных цветов). Такой код не слишком значим, но, что еще хуже, используя один и тот же цвет в разных местах, следует везде повторять его код. Это плохо, как и любой иной вид дублирования кода. Было бы лучше дать такому цвету имя, значимое в данном контексте, наподобие MEDIUM_SHADE, и в некотором одном месте определить, что MEDIUM_SHADE представляет собой #FFB595.

Хотя язык CSS, по крайней мере в настоящее время, не позволяет это сделать, для обработки таких ситуаций можно использовать макропроцессор. Для этого просто создайте файл, который представляет собой ваш CSS файл, но с применением MEDIUM_SHADE везде, где нужен соответствующий цвет. Затем макропроцессор выполнит простую замену текста MEDIUM_SHADE текстом #FFB595.

Это очень простая форма обработки макросов; чаще используются макросы, в которых можно использовать параметры. Классическим примером является препроцессор C, в котором можно определить, например, макрос для замены всех `sqrt(x)` на `x*x`.

Макросы предоставляют богатые возможности для создания DSL как в рамках базового языка (как препроцессор C), так и в виде отдельных файлов, преобразуемых в базовый язык. Недостатком является ряд проблем при применении макросов, которые делают их трудно используемыми на практике. В результате текстовые макросы в значительной степени потеряли привлекательность, и большинство специалистов вроде меня советуют их не использовать.

Синтаксические макросы также выполняют замены, но работают с синтаксически корректными элементами базового языка, преобразуя выражения из одного вида в другой. Самый известный своим интенсивным использованием синтаксических макросов — язык Lisp; хорошо также известен своими шаблонами C++. Использование синтаксических макросов для DSL является основной технологией написания внутренних DSL в Lisp, но дело в том, что вы можете применять синтаксические макросы только в том языке, который их поддерживает. В данной книге о них сказано немного, но это связано с тем, что синтаксические макросы поддерживаются лишь несколькими языками.

3.6. Тестирование DSL

За последние два десятилетия я существенно углубился в тему тестирования. Я большой поклонник разработки на основе тестирования (*test-driven development* [4]) и других подобных методов, которые выводят тестирование в авангард программирования. В результате я не могу говорить о DSL, не задумываясь об их тестировании.

В случае DSL тестирование можно разделить на три отдельные области: тестирование семантической модели (*Semantic Model* (171)), тестирование синтаксического анализатора и тестирование сценариев.

3.6.1. Тестирование семантической модели

Первая область тестирования DSL — тестирование семантической модели (*Semantic Model* (171)). Эти тесты должны гарантировать корректность поведения семантической модели, т.е. то, что при ее выполнении она дает верный вывод, зависящий от того, что помещается в модель. Это стандартная практика тестирования, такая же, как и при использовании любых других наборов объектов. Для проведения такого тестирования сам DSL не нужен — модель можно заполнить с помощью базового интерфейса самой модели. Это хорошо, так как позволяет тестировать модели независимо от DSL и синтаксических анализаторов.

Позвольте мне проиллюстрировать сказанное на примере контроллера системы безопасности. В данном случае это семантическая модель конечного автомата. Я могу протестировать семантическую модель путем ее наполнения с помощью кода с применением API командных запросов, как это делалось в главе 1, “Вводный пример”, на с. 34, где не требовалось наличие какого-либо DSL.

```
@Test
public void event_causes_transition() {
    State idle = new State("idle");
    StateMachine machine = new StateMachine(idle);
    Event cause = new Event("cause", "EV01");
    State target = new State("target");
    idle.addTransition(cause, target);
    Controller controller =
        new Controller(machine, new CommandChannel());
    controller.handle("EV01");
    assertEquals(target, controller.getCurrentState());
}
```

В приведенном выше коде показано, что я могу тестировать семантическую модель совершенно отдельно. Однако следует заметить, что реальный код теста в данном случае должен быть существенно большим, лучше организованным и разделенным.

Вот несколько способов достижения наилучшего разделения подобного кода. Во-первых, можно создать много маленьких конечных автоматов с минимальной функциональностью для тестирования различных возможностей семантической модели. Все, что нужно для того, чтобы убедиться, что события запускают переходы, — это простой конечный автомат с состоянием покоя и двумя исходящими переходами в другие состояния.

```
class TransitionTester...
State idle, a, b;
Event trigger_a, trigger_b, unknown;

protected StateMachine createMachine() {
    idle = new State("idle");
    StateMachine result = new StateMachine(idle);
    trigger_a = new Event("trigger_a", "TRGA");
    trigger_b = new Event("trigger_b", "TRGB");
    unknown = new Event("Unknown", "UNKN");
    a = new State("a");
    b = new State("b");
    idle.addTransition(trigger_a, a);
    idle.addTransition(trigger_b, b);
    return result;
}
```

Для того чтобы протестировать команды, достаточно и меньшего автомата с одним состоянием, достижимым из состояния покоя.

```

class CommandTester...
    Command commenceEarthquake =
        new Command("Commence Earthquake", "EQST");
    State idle = new State("idle");
    State second = new State("second");
    Event trigger = new Event("trigger", "TGGR");

protected StateMachine createMachine() {
    second.addAction(commenceEarthquake);
    idle.addTransition(trigger, second);
    return new StateMachine(idle);
}

```

Все эти разные тесты могут быть выполнены единообразно. Я могу поступить проще, обеспечив им общий суперкласс. Первое, что предоставляет этот класс, — возможность настройки общих устройств, в данном случае — инициализации контроллера и командного канала к прилагаемому конечному автомату.

```

class AbstractStateTesterLib...
protected
    CommandChannel commandChannel = new CommandChannel();
protected StateMachine machine;
protected Controller controller;

@Before
public void setup() {
    machine = createMachine();
    controller = new Controller(machine, commandChannel);
}

abstract protected StateMachine createMachine();

```

Теперь можно написать тесты для запуска событий контроллера и проверки состояний.

```

class TransitionTester...
@Test
public void event_causes_transition() {
    fire(trigger_a);
    assertEquals(a);
}

@Test
public void event_without_transition_is_ignored() {
    fire(unknown);
    assertEquals(idle);
}

class AbstractStateTesterLib...
//----- Вспомогательные методы -----
protected void fire(Event e) {
    controller.handle(e.getCode());
}
//----- Пользовательские проверки -----
protected void assertEquals(State s) {
    assertEquals(s, controller.getCurrentState());
}

```

Суперкласс предоставляет *вспомогательные тестовые методы* (Test Utility Methods [20]) и *пользовательские проверки* (Custom Assertions [20]) для упрощения чтения кода тестов.

Альтернативный подход к тестированию семантической модели заключается в заполнении большей модели, демонстрирующей многие возможности, и в выполнении над

ней ряда тестов. В нашем случае в качестве тестового устройства можно использовать контроллер мисс Грант.

```
class ModelTest...  
    private Event doorClosed, drawerOpened, lightOn,  
                doorOpened, panelClosed;  
    private State activeState, waitingForLightState,  
                unlockedPanelState, idle,  
                waitingForDrawerState;  
    private Command unlockPanelCmd, lockDoorCmd,  
                lockPanelCmd, unlockDoorCmd;  
    private CommandChannel channel = new CommandChannel();  
    private Controller con;  
    private StateMachine machine;  
  
@Before  
public void setup() {  
    doorClosed = new Event("doorClosed", "D1CL");  
    drawerOpened = new Event("drawerOpened", "D2OP");  
    lightOn = new Event("lightOn", "L1ON");  
    doorOpened = new Event("doorOpened", "D1OP");  
    panelClosed = new Event("panelClosed", "PNCL");  
    unlockPanelCmd = new Command("unlockPanel", "PNUL");  
    lockPanelCmd = new Command("lockPanel", "PNLK");  
    lockDoorCmd = new Command("lockDoor", "D1LK");  
    unlockDoorCmd = new Command("unlockDoor", "D1UL");  
  
    idle = new State("idle");  
    activeState = new State("active");  
    waitingForLightState = new State("waitingForLight");  
    waitingForDrawerState = new State("waitingForDrawer");  
    unlockedPanelState = new State("unlockedPanel");  
  
    machine = new StateMachine(idle);  
  
    idle.addAction(unlockDoorCmd);  
    idle.addAction(lockPanelCmd);  
  
    activeState.addTransition(drawerOpened,  
                             waitingForLightState);  
    activeState.addTransition(lightOn,  
                             waitingForDrawerState);  
  
    waitingForLightState.addTransition(lightOn,  
                                      unlockedPanelState);  
    waitingForDrawerState.addTransition(drawerOpened,  
                                      unlockedPanelState);  
  
    unlockedPanelState.addAction(unlockPanelCmd);  
    unlockedPanelState.addAction(lockDoorCmd);  
    unlockedPanelState.addTransition(panelClosed, idle);  
  
    machine.addResetEvents(doorOpened);  
    con = new Controller(machine, channel);  
    channel.clearHistory();  
}  
  
@Test  
public void event_causes_state_change() {  
    fire(doorClosed);
```

```

    assertCurrentState(activeState);
}

@Test
public void ignore_event_if_no_transition() {
    fire(drawerOpened);
    assertCurrentState(idle);
}

```

Здесь я вновь наполняю семантическую модель с помощью ее собственного интерфейса командных запросов. Поскольку в данном случае тестовые устройства более сложные, я могу упростить тестирующий код, применив DSL для создания устройств. Я могу сделать это, если у меня есть тесты для синтаксического анализатора.

3.6.2. Тестирование синтаксического анализатора

При использовании семантической модели (Semantic Model (171)) работа синтаксического анализатора состоит в ее наполнении. Поэтому наше тестирование синтаксического анализатора заключается в написании небольших фрагментов DSL и проверке того факта, что они создают корректные структуры в семантической модели.

```

@Test
public void loads_states_with_transition() {
    String code =
        "events trigger TGGR end " +
        "state idle " +
        "trigger => target " +
        "end " +
        "state target end ";
    StateMachine actual = StateMachineLoader.loadString(code);

    State idle = actual.getState("idle");
    State target = actual.getState("target");
    assertTrue(idle.hasTransition("TGGR"));
    assertEquals(idle.targetState("TGGR"), target);
}

```

Работать с семантической моделью так, как показано здесь, довольно неудобно, и это может привести к нарушению инкапсуляции объектов в семантической модели. Поэтому другой способ тестирования выхода синтаксического анализатора состоит в определении и использовании методов для сравнения семантических моделей.

```

@Test
public void loads_states_with_transition_using_compare() {
    String code =
        "events trigger TGGR end " +
        "state idle " +
        "trigger => target " +
        "end " +
        "state target end ";
    StateMachine actual = StateMachineLoader.loadString(code);

    State idle = new State("idle");
    State target = new State("target");
    Event trigger = new Event("trigger", "TGGR");
    idle.addTransition(trigger, target);
    StateMachine expected = new StateMachine(idle);

    assertEquals(expected, actual);
}

```

Проверка равенства сложных структур должна включать нечто большее, чем предполагает простой оператор равенства. Кроме того, при проверке недостаточно простого логического значения, поскольку нужно точно знать, в чем именно состоит различие между объектами. В результате в моем сравнении используется шаблон уведомления (Notification (205)).

```

class StateMachine...
public Notification probeEquivalence(StateMachine other) {
    Notification result = new Notification();
    probeEquivalence(other, result);
    return result;
}

private void probeEquivalence(StateMachine other,
                               Notification note) {
    for (State s : getStates()) {
        State otherState = other.getState(s.getName());
        if (null == otherState)
            note.error("missing state: %s", s.getName());
        else s.probeEquivalence(otherState, note);
    }
    for (State s : other.getStates())
        if (null == getState(s.getName()))
            note.error("extra state: %s", s.getName());
    for (Event e : getResetEvents()) {
        if (!other.getResetEvents().contains(e))
            note.error("missing reset event: %s", e.getName());
    }
    for (Event e : other.getResetEvents()) {
        if (!getResetEvents().contains(e))
            note.error("extra reset event: %s", e.getName());
    }
}

class State...
void probeEquivalence(State other, Notification note) {
    assert name.equals(other.name);
    probeEquivalentTransitions(other, note);
    probeEquivalentActions(other, note);
}

private void probeEquivalentActions(State other,
                                   Notification note) {
    if (!actions.equals(other.actions))
        note.error("%s has different actions %s vs %s",
                   name, actions, other.actions);
}

private
void probeEquivalentTransitions(State other,
                                 Notification note) {
    for (Transition t : transitions.values())
        t.probeEquivalent(
            other.transitions.get(t.getEventCode()), note);
    for (Transition t : other.transitions.values())
        if (!this.transitions.containsKey(t.getEventCode()))
            note.error("%s has extra transition with %s", name,
                       t.getTrigger());
}

```

В данном тесте выполняется обход объектов семантической модели с записью всех отличий в объект уведомления. Таким образом выявляются все различия, а не только первое. Затем выполняется проверка, имеются ли в уведомлении какие-то данные об ошибках.

```
class AntlrLoaderTest...
private void assertEqualsMachines(StateMachine left,
                                 StateMachine right) {
    assertNotificationOk(left.probeEquivalence(right));
    assertNotificationOk(right.probeEquivalence(left));
}

private void assertNotificationOk(Notification n) {
    assertTrue(n.report(), n.isOk());
}

class Notification...
public boolean isOk() {return errors.isEmpty();}
```

Можете считать меня пааноиком, проверяющим эквивалентность в обоих направлениях, но так поступаю не только я.

Проверка некорректных входных данных

Только что рассмотренные тесты положительны в том плане, что они гарантируют, что корректный входной текст DSL создает правильные структуры в семантической модели (*Semantic Model* (171)). Другая категория тестов — отрицательные тесты, которые определяют, что произойдет при неверном входном тексте DSL. Это относится ко всей области обработки ошибок и диагностики в целом. И хотя она выходит за рамки данной книги, я не могу не упомянуть о тестах некорректных входных данных.

Суть такого тестирования состоит в подаче на вход синтаксического анализатора неверной информации различных видов. При первом выполнении такого рода тестов просто интересна реакция синтаксического анализатора. Чаще всего получается непонятная, но критичная ошибка. В зависимости от того, какую диагностическую поддержку необходимо предоставить с DSL, этого может оказаться достаточно. Гораздо хуже, если в случае некорректного исходного текста он будет проанализирован без сообщений о каких-либо ошибках. Это нарушило бы принцип “быстрой ошибки”, заключающейся в том, что все ошибки должны выявляться на как можно более ранней стадии, а сообщения о них должны быть такими, чтобы не заметить их было бы невозможно. Если вы заполните модель недопустимым состоянием и у вас нет соответствующих проверок, то об этой проблеме вы узнаете намного позднее. Наличие расстояния между исходной ошибкой (загрузка неверной входной информации) и некорректной работой в дальнейшем затрудняет поиск ошибок, и чем больше это расстояние, тем сложнее поиск.

Мой пример конечного автомата включает минимальную обработку ошибок, которая, увы, свойственна всем примерам из данной книги. Один из примеров синтаксического анализатора, впрочем, будет проверен таким образом, просто чтобы посмотреть, что из этого выйдет.

```
@Test public void targetStateNotDeclaredNoAssert () {
    String code =
        "events trigger TGGR end " +
        "state idle " +
        "trigger => target " +
        "end ";
    StateMachine actual = StateMachineLoader.loadString(code);
}
```

Тест пройден — и это плохо. После этого, когда я пытаюсь сделать что-то с моделью (даже просто ее распечатать), я получаю исключение нулевого указателя. Этот пример достаточно прост (в конце концов, единственная его цель — дидактическая), но опечатка во входном тексте DSL может привести к существенному увеличению времени на отладку. Поскольку это мое личное время и я высоко его ценю, я хотел бы, чтобы хорошо работал принцип “быстрой ошибки”.

Поскольку проблема в том, что я создаю некорректную структуру в семантической модели, ответственность за соответствующую проверку лежит на семантической модели, в данном случае — на методе, который добавляет переход к состоянию. Я добавляю утверждение для выявления проблем такого рода.

```
class State...
    public void addTransition(Event event,
                               State targetState) {
        assert null != targetState;
        transitions.put(event.getCode(),
                        new Transition(this, event, targetState));
    }
```

Теперь я могу изменить тест, чтобы перехватить исключение. Помимо всего прочего, он документирует тип ошибки, вызываемой таким неверным входом.

```
@Test public void targetTypeNotDeclared () {
    String code =
        "events trigger TGGR end " +
        "state idle " +
        "trigger => target " +
        "end ";
    try {
        StateMachine actual =
            StateMachineLoader.loadString(code);
        fail();
    } catch (AssertionError expected) {}
```

В утверждении проверяется целевое состояние, а не событие, которое также может быть нулевым. Причина в том, что нулевое событие вызовет немедленное исключение нулевого указателя из-за наличия вызова `event.getCode()`. Это соответствует принципу “быстрой ошибки”. Проверить это можно с помощью другого теста.

```
@Test public void triggerNotDeclared () {
    String code =
        "events trigger TGGR end " +
        "state idle " +
        "wrongTrigger => target " +
        "end " +
        "state target end ";
    try {
        StateMachine actual =
            StateMachineLoader.loadString(code);
        fail();
    } catch (NullPointerException expected) {}
```

Исключение нулевого указателя соответствует принципу “быстрой ошибки”, но при этом оно не столь ясно, как невыполненное утверждение. Вообще говоря, я не применяю проверки на равенство `null` к аргументам своих методов, так как, по моему мнению, выгода от них не оправдывает необходимости читать дополнительный код. Исключением являются ситуации, в которых нулевые значения не приводят к немедленным сбоям в работе, как в рассмотренном выше случае нулевого целевого состояния.

3.6.3. Тестирование сценариев

Тестирование семантической модели (*Semantic Model* (171)) и синтаксического анализатора представляет собой модульное тестирование обобщенного кода. Однако сценарии DSL также являются кодом, так что мы должны рассмотреть и их тестирование. Я уже слышу аргументы типа “сценарии DSL слишком просты и очевидны, чтобы их тестировать”, но я, естественно, отношусь к ним отрицательно. Я представляю такое тестирование как механизм двойной проверки. В процессе написания кода и тестов мы определяем одно и то же поведение, используя совершенно разные механизмы, один из которых включает абстракции (код), а другой — примеры (тесты). Чтобы такое тестирование имело смысл, всегда следует выполнять двойную проверку.

Детали тестирования сценариев во многом зависят от того, что именно тестируется. Общий подход заключается в предоставлении тестовой среды, которая позволяет создавать исходные тексты, запускать сценарии DSL и сравнивать результаты. Подготовка такой среды, как правило, требует некоторых усилий, но то, что DSL легко читается, еще не означает, что программисты не будут делать ошибки. Если вы не предоставите тестовую среду (и, таким образом, у вас не будет механизма двойной проверки), вы значительно увеличите риск возникновения ошибок в сценариях DSL.

Тесты сценариев действуют и как тесты интеграции, поскольку любые ошибки в синтаксическом анализаторе или семантической модели должны приводить к их сбоям. Так что стоит выбрать несколько сценариев DSL для их использования для этой цели.

Часто альтернативные визуализации сценария оказывают помощь в тестировании и отладке DSL-сценариев. Различные текстовые и графические визуализации логики сценария выполнить относительно легко. Представление информации несколькими различными способами часто помогает людям находить ошибки. Понятие двойной проверки — главная причина написания самотестируемого кода.

Для нашего конечного автомата я начал с примеров, имеющих смысл для данного типа автомата. На мой взгляд, логичнее было бы запускать сценарии, каждый из которых представляет собой последовательность событий, посыпаемых автомatu. Затем следует проверить конечное состояние автомата и команды, которые были им посланы. Построение системы наподобие описанной удобочитаемым способом естественным образом приводит к другому DSL. Это не редкость — тестирование сценариев является обычным применением DSL, так как они хорошо отвечают потребности в ограниченном, декларативном языке.

```
events("doorClosed", "drawerOpened", "lightOn")
    .endsAt("unlockedPanel")
    .sends("unlockPanel", "lockDoor");
```

3.7. Обработка ошибок

Всякий раз, когда я пишу книгу, наступает момент, когда я понимаю, что следует ограничить рассматриваемый материал, иначе эту книгу никогда не увидит читатель. Хотя это и означает, что некоторая важная тема не рассматривается надлежащим образом, я все же считаю, что неполная книга полезнее, чем неизданная. Есть много тем, которые я бы хотел осветить в этой книге, и первой в списке является обработка ошибок.

Когда я изучал компиляторы в университете, преподаватель сказал, что самое простое при написании компилятора — это создание синтаксического анализатора и генератора кода, а самое трудное — обеспечение информативных сообщений об ошибках. Соответ-

ственno, диагностика ошибок оказалась за рамками изучаемого мною в университете курса, точно так же, как сейчас она находится за рамками этой книги.

Но эта “зарамочность” идет еще дальше: хорошая диагностика — большая редкость даже в успешных DSL. Многие очень полезные пакеты DSL выдают очень мало полезной информации об ошибках. Graphviz, один из моих любимых инструментов DSL, просто сообщает о синтаксической ошибке вблизи строки 4, и я чувствую себя счастливым от того, что меня хотя бы ориентировали по номеру строки. Я оказывался в ситуациях, когда приходилось по сути выполнять бинарный поиск (состоящий в комментировании половины строк рассматриваемого блока), чтобы найти место расположения проблемы.

Можно справедливо критиковать такие системы за их слабую диагностику ошибок, но уровень диагностики — это результат компромиссов. Время на улучшение обработки ошибок — это время, которое могло бы быть затрачено на добавление других возможностей. Практика свидетельствует, что в реальных DSL их пользователи, как правило, вполне мирятся со слабой диагностикой ошибок. В конце концов, сценарии DSL малы, так что грубые методы поиска ошибок в этом случае работают существенно лучше, чем с языками программирования общего назначения.

Я говорю это не для того, чтобы убедить вас не работать над диагностикой ошибок. В интенсивно используемых библиотеках хорошая диагностика может сэкономить много времени. Каждый компромисс уникален, и вы должны достигать его, исходя из собственных обстоятельств. Однако это соображение позволяет моей совести не так сильно мучить меня из-за того, что я не уделил данному вопросу должного внимания в этой книге.

Я не могу изложить в этой книге все, что мне хотелось бы написать по данной теме, но надеюсь, что приведенного материала будет достаточно для того, чтобы, если вы решите усовершенствовать свои разработки, вы уделили диагностике ошибок больше внимания.

(Один момент, который я должен упомянуть, — это самая грубая методика поиска ошибок, состоящая в комментировании больших блоков исходного текста. В случае внешнего DSL убедитесь, что вы поддерживаете в нем комментарии, — не только в силу очевидных причин, но и для того, чтобы помочь людям найти проблемы в своем коде. Работать с такими комментариями легче всего тогда, когда они продолжаются до конца строки. В зависимости от целевой аудитории я обычно использую либо “#” (стиль сценариев), либо “//” (стиль C). Это делается с помощью очень простого правила лексического анализатора.)

Если вы следете моему глобальному совету использовать семантическую модель (*Semantic Model* (171)), то учтите, что для размещения обработки ошибок есть два места: модель и синтаксический анализатор. Очевидным местом обработки синтаксических ошибок является синтаксический анализатор. Некоторые синтаксические ошибки будут обрабатываться вместо вас: синтаксические ошибки базового языка во внутреннем DSL или грамматические ошибки при использовании генератора синтаксических анализаторов (*Parser Generator* (277)) во внешнем DSL.

В ситуации с семантическими ошибками у вас есть выбор между обработкой их синтаксическим анализатором и моделью. В смысле семантики у обоих есть свои сильные стороны. Модель — действительно подходящее место для проверки правил семантически корректно сформированных структур. У вас есть вся необходимая информация, структурированная необходимым образом, так что вы можете написать ясный код для проверки наличия ошибок. Кроме того, если модель заполняется более чем из одного места (например, при использовании нескольких DSL или с помощью интерфейса командных запросов), то выполнять проверки следует именно здесь.

Размещение обработки ошибок только в семантической модели имеет один серьезный недостаток: отсутствует связь с источником проблемы в сценарии DSL, так что но-

мер строки неизвестен даже приблизительно. Это существенно затрудняет выяснение, что же пошло не так, но вовсе не является неразрешимой проблемой. Имеющийся опыт позволяет предположить, что даже сообщения об ошибке от модели достаточно, чтобы в самых разных ситуациях найти источник проблемы.

Существует несколько способов получения контекста сценария DSL, если в нем возникает необходимость. Наиболее очевидным является размещение правил обнаружения ошибок в синтаксическом анализаторе. Однако проблемой при использовании этой стратегии является существенное усложнение написания правил, так как работа ведется на уровне синтаксического дерева, а не семантической модели. Существенно вырастает и риск дублирования правил со всеми вытекающими из дублирования кода проблемами.

В качестве альтернативы можно внести синтаксическую информацию в семантическую модель. Можно добавить поле номера строки в объект семантического перехода, так что семантическая модель, обнаружив ошибки в переходе, может вывести номер строки из сценария. В этом случае проблема кроется в усложнении семантической модели, которая должна отслеживать синтаксическую информацию. Кроме того, отображение сценария на модель может не быть совершенно ясным и однозначным и привести к сообщениям об ошибках, которые еще больше запутают пользователя.

Третья стратегия (которая мне нравится больше всех) заключается в использовании для обнаружения ошибок семантической модели и запуске системы обнаружения ошибок в синтаксическом анализаторе. Таким образом, синтаксический анализатор будет анализировать фрагмент сценария DSL, заполнять семантическую модель и запрашивать проверку этой модели (если это не делается непосредственно при заполнении модели). Если модель находит ошибки, синтаксический анализатор может получить информацию о них и сопоставить с известным ему контекстом сценария DSL. Таким образом, происходит отделение синтаксических знаний (в синтаксическом анализаторе) от знаний семантических (в модели).

Полезным подходом является разделение обработки ошибок на инициацию, обнаружение и отчетность. Последняя стратегия переносит инициацию в синтаксический анализатор, обнаружение — в модель, а отчетность — в оба места. При этом модель поставляет семантику ошибки, а синтаксический анализатор добавляет к ней синтаксический контекст.

3.8. Миграция DSL

Одна из опасностей, о которой предупреждают сторонники DSL, заключается в том, что сначала вы проектируете DSL, а затем люди его используют. Как и любое другое программное обеспечение, успешные DSL будут развиваться. Это означает, что сцены, написанные в ранней версии DSL, могут не работать в более поздней версии.

Подобно многим другим свойствам DSL, хорошим и плохим, это очень похоже на то, что происходит с библиотеками. Если вы получили от кого-то библиотеку, написали использующий ее код, а библиотека затем была обновлена, ваш код может перестать работать. DSL в этом плане ничем не отличаются; определение DSL, по сути, представляет собой опубликованный интерфейс, так что вы имеете дело с теми же последствиями, что и при работе с библиотеками.

Я начал использовать термин **опубликованный интерфейс** в своей книге о рефакторинге [12]. Разница между опубликованным (published) и более распространенным открытым (public) интерфейсом состоит в том, что опубликованный интерфейс используется кодом, написанным иной, отдельной командой. Поэтому когда команда, которая определяет интерфейс, хочет его изменить, она не может просто переписать вызываю-

щий код. Изменение опубликованного DSL является проблемой как для внутренних, так и для внешних предметно-ориентированных языков. В случае неопубликованных DSL, пожалуй, легче поддаются изменению внутренние DSL, если рассматриваемый язык имеет автоматизированные средства рефакторинга.

Один из способов решения проблемы изменения DSL — предоставление инструментов, которые автоматически выполняют миграцию DSL с одной версии на другую. Они могут быть запущены либо во время обновления, либо автоматически при попытках запуска старых версий сценариев.

Есть два основных способа справиться с задачей миграции. Первый — стратегия **инкрементной миграции**. Это по сути то же понятие, что и используемое при эволюционном проектировании баз данных [9]. Для каждого изменения, вносимого в определение DSL, создается программа миграции, которая автоматически преобразует все старые сценарии в сценарии новой версии. Таким образом, выпуская новую версию DSL, вы одновременно обеспечиваете возможность перехода на новую версию любого кода, использующего DSL.

Важной частью инкрементной миграции является то, что вы делаете вносимые изменения минимальными. Представьте, что вы выполняете обновление с версии 1 до версии 2 и у вас имеется десять изменений, вносимых в определение вашего DSL. В этом случае не создавайте только один сценарий для миграции с версии 1 до 2; создайте вместо этого по крайней мере десять сценариев. Изменяйте определение DSL по одной возможности за раз и пишите сценарий миграции для каждого вносимого изменения. Вы можете решить, что стоит разбивать процесс еще сильнее и добавлять некоторые возможности более чем за один шаг (а значит, более чем одной миграцией). Может показаться, что такой способ приведет к большему объему работы, чем потребовал бы один сценарий, но дело в том, что сценарии миграции намного легче писать, если они маленькие, и достаточно просто соединить несколько таких последовательных миграций вместе. В результате вы сможете написать десять сценариев гораздо быстрее, чем один.

Другой подход заключается в миграции на основе моделей. Этую тактику можно использовать с семантической моделью (*Semantic Model* (171)). **Миграция на основе моделей** позволяет поддерживать несколько синтаксических анализаторов вашего языка, по одному для каждой версии. (Таким образом, вы делаете это только для версий 1 и 2, но не для промежуточных шагов.) Каждый синтаксический анализатор заполняет семантическую модель. При использовании семантической модели поведение синтаксического анализатора довольно простое, так что иметь их несколько не доставляет особых хлопот. Затем вы запускаете синтаксический анализатор, соответствующий версии сценария, с которой вы работаете. Так вы можете работать с несколькими версиями сценариев, не прибегая к их миграции. Для выполнения миграции вы пишете генератор на основе семантической модели, который генерирует представление сценария DSL. Таким образом, вы можете запустить синтаксический анализатор для сценария версии 1, заполнить семантическую модель и получить сценарий для версии 2 с помощью генератора.

Одна из проблем, возникающих в процессе применения данной миграции на основе модели, заключается в том, что в сценариях легко потерять то, что не имеет значения для семантики, но что авторы сценариев хотели бы сохранить. Очевидный пример — комментарии.

Если изменений в DSL достаточно много, может оказаться невозможным преобразование сценария версии 1 в сценарий версии 2 семантической модели. В этом случае, возможно, потребуется сохранить модель версии 1 (или промежуточную модель) и придать ей возможность генерировать сценарии версии 2.

У меня нет явно выраженного предпочтения того или иного из описанных путей.

Сценарии миграции могут быть запущены при необходимости самими программистами сценариев или автоматически системой DSL. При автоматическом выполнении

очень полезно иметь возможность записи версии DSL в сценарии, чтобы синтаксический анализатор мог легко его выяснить и выполнить миграцию. Действительно, некоторые авторы DSL утверждают, что все DSL обязательно должны содержать указание версии в сценарии, чтобы можно было легко обнаруживать устаревшие сценарии и обеспечивать поддержку миграции сценариев. Хотя инструкция указания версии несколько “засоряет” сценарий, ее очень трудно усовершенствовать.

Конечно, есть еще один вариант — отказаться от миграции, т.е. поддерживать синтаксический анализатор версии 1 и позволить ему заполнять модель версии 2. Вы должны помочь людям перейти на новую версию, и они должны будут это сделать, если хотят использовать дополнительные возможности. Но непосредственная поддержка старых сценариев, если таковая возможна, весьма полезна, так как позволяет пользователям мигрировать в собственном темпе.

Хотя такие методы весьма привлекательны, возникает вопрос об их стоимости на практике. Как я уже говорил, эта проблема точно такая же, как и в случае широко используемых библиотек, а здесь автоматизированные схемы миграции практически не применяются.

Глава 4

Реализация внутреннего DSL

Теперь, когда рассмотрены общие вопросы реализации DSL, пришло время ознакомиться с особенностями реализации конкретных типов DSL. Я решил начать с внутренних DSL, так как они являются наиболее доступной для написания формой DSL. Вам не нужно изучать их грамматику и синтаксический анализ, как при освоении внешних DSL, а в отличие от языковых инструментальных средств для работы вам не нужны никакие специальные инструменты. Используя внутренний DSL, вы работаете в своей обычной языковой среде. Поэтому не удивителен проявленный в последние годы интерес к внутренним DSL.

При использовании внутренних DSL вы существенно ограничены базовым языком. Так как любое используемое вами выражение должно быть корректным выражением базового языка, особенности применения внутреннего DSL тесно связаны с возможностями языка. Существенно стимулирует применение внутренних DSL сообщество программистов на языке программирования Ruby, который имеет много поддерживающих DSL особенностей. Однако многие методы Ruby могут быть использованы и в других языках, хотя обычно не столь элегантно. Старейшиной в области применения внутреннего DSL является Lisp, один из древнейших компьютерных языков, обладающий хотя и ограниченным, но очень подходящим для данного предназначения набором возможностей.

Еще один термин, имеющий отношение к внутренним DSL, — **свободный интерфейс** (fluent interface). Его впервые ввели Эрик Эванс (Eric Evans) и я для описания языкообразных API. Это синоним внутренних DSL, рассматриваемых с точки зрения API. Он указывает на ключевую разницу между API и DSL — природу языка. Как я уже указывал, между ними имеется некая переходная зона. Можно иметь разумные, но плохо обоснованные аргументы о том, является ли некоторая конкретная конструкция языка “языкообразной”. Преимущество таких аргументов в том, что они поощряют всестороннее обдумывание используемых методов и удобочитаемости DSL. Недостатком же их является то, что они могут способствовать постоянному использованию (под новыми именами) одних и тех же личных предпочтений.

4.1. Свободные API и API командных запросов

Для большинства программистов основным шаблоном свободного интерфейса является соединение, или связывание, методов в цепочку вызовов (Method Chaining (375)). Код с применением обычного API может выглядеть следующим образом.

```
Processor p = new Processor(2, 2500, Processor.Type.i386);  
Disk d1 = new Disk(150, Disk.UNKNOWN_SPEED, null);  
Disk d2 = new Disk(75, 7200, Disk.Interface.SATA);  
return new Computer(p, d1, d2);
```

При использовании связывания методов это же можно выразить иначе.

```
computer()
  .processor()
    .cores(2)
    .speed(2500)
    .i386()
  .disk()
    .size(150)
  .disk()
    .size(75)
    .speed(7200)
    .sata()
.end();
```

При связывании методов используется последовательность вызовов методов, в которой каждый вызов работает с результатом предыдущего вызова. Методы объединяются путем вызова одного поверх другого. В обычном объектно-ориентированном коде такие конструкции часто высмеиваются как “обломки поезда”: методы, разделенные точками, выглядят, как вагоны, а обломками их считают в связи с тем, что зачастую это признак кода, не позволяющего вносить изменения в интерфейсы классов в средине цепочки вызовов. Однако если поразмыслить без предубежденности, то связывание методов позволяет легко объединить несколько вызовов методов без множества переменных и получить код, вызывающий ощущение исходного текста некоторого отдельного языка.

Но связывание методов — это не только способ получить подобное ощущение. Вот еще один вариант записи той же функциональности с помощью последовательности инструкций вызова методов, которую я называю последовательностью функций (*Function Sequence* (357)).

```
computer();
  processor();
    cores(2);
    speed(2500);
    i386();
  disk();
    size(150);
  disk();
    size(75);
    speed(7200);
    sata();
```

Как можно видеть, если попробовать соответствующим образом организовать и разместить последовательность функций, она может читаться так же легко, как и цепочка методов. (В названии шаблона я употребляю слово “функция” вместо “метод”, так как его можно использовать не в объектно-ориентированном контексте с помощью вызовов функций, в то время как при связывании методов в цепочку требуется, чтобы это были объектно-ориентированные методы.) Главное в том, что свобода выражается не столько в стиле используемого синтаксиса, сколько в способе именования и разложения самих методов.

В первые дни объектно-ориентированного программирования на меня и многих других огромное влияние оказала книга Бертрана Мейера (*Bertrand Meyer*) *Object-Oriented Software Construction*. Одна из использованных им аналогий заключалась в том, что, говоря об объектах, он рассматривал их в качестве машин. С этой точки зрения объект представляет собой “черный ящик”, а его интерфейс — это просто набор дисплеев для просмотра наблюдаемого состояния объекта, и кнопок, которые можно нажимать для изменения объекта (рис. 4.1). Вам фактически предлагается меню из различных операций, которые можно выполнять над объектом. Этот стиль интерфейса является доминирующим способом представления взаимодействия с программными компонентами. Он доминирует на-

столько, что ему даже не дали имя, и мне пришлось для его описания самостоятельно придумать термин “интерфейс командных запросов”.

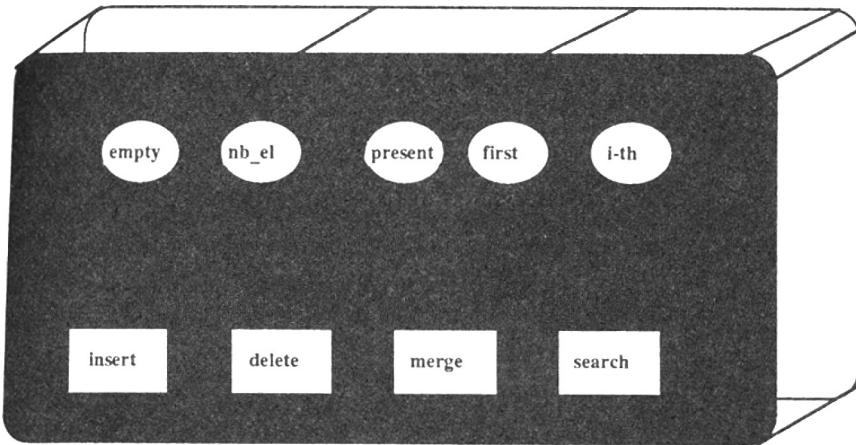


Рис. 4.1. Оригинальный рисунок из книги Мейера для иллюстрации машинной метафоры. Овалы представляют кнопки запросов со световыми индикаторами, которые указывают состояние машины при нажатии кнопки, но не изменяют его. Прямоугольные кнопки изменяют состояние, но не имеют индикаторов, чтобы указать, что получилось в результате

Суть свободных интерфейсов — в ином подходе к использованию компонентов. Вместо представления в виде черного ящика с кнопками мы осуществляляем лингвистический подход, составляя предложения, объединяющие объекты. Именно эта перемена в подходе и является основной разницей между внутренними DSL и простым вызовом API.

Как я уже говорил, это очень нечеткое различие. Трактовка API как языков также достаточно древняя и хорошо изученная аналогия, история которой началась в те времена, когда объекты еще не стали нормой. Существует множество ситуаций, когда трудно сказать, что именно находится перед вами — API командных запросов или свободный интерфейс. И тем не менее я считаю, что при всей его нечеткости это полезное различие.

Одним из следствий различия между этими двумя стилями интерфейса является различие в правилах создания хорошего интерфейса. Здесь оказываются кстати машинная метафора Мейера и приведенный выше рисунок, который иллюстрирует принцип разделения запросов и команд.

Разделение запросов и команд гласит, что методы объекта должны быть разделены на команды и запросы. Запрос представляет собой метод, который возвращает значение, но не меняет наблюдаемого состояния системы. Команда может изменять наблюдаемое состояние, но не должна возвращать значение. Это важный принцип, поскольку он помогает идентифицировать методы запросов. Так как запросы не имеют побочных эффектов, их можно вызывать многократно, а также можно изменять порядок их использования без изменения результатов вызовов. В случае команд нужно быть намного более осторожными, поскольку они обладают побочными эффектами.

Разделение команд и запросов — чрезвычайно важный принцип в программировании, и я настоятельно рекомендую его использовать. Одним из следствий использования связывания методов в цепочку во внутреннем DSL является нарушение этого принципа: каждый метод, изменяющий состояние, возвращает объект для продолжения цепочки вызовов. Я сорвал голос, призывая людей следовать принципу разделения команд и запросов, и буду убеждать их и впредь, но свободные интерфейсы следуют совсем другому набору правил.

Еще одно важное отличие между командными запросами и свободными интерфейсами — в именовании методов. Присваивая имена интерфейсу командных запросов, вы хотите, чтобы они имели смысл в автономном контексте. Часто, когда люди ищут метод для выполнения каких-то действий, они опускают глаза на список методов на странице веб-документа или в меню интегрированной среды разработки. Поэтому имена должны ясно указывать, что делают соответствующие методы, как если бы они представляли собой надписи на кнопках.

В случае свободных интерфейсов именование — это совсем иная задача. Здесь вы существенно меньше концентрируетесь на отдельных элементах языка, а больше — на предложениях, которые вы можете сформировать. В результате не редкость методы, имеющие которых не имеют смысла в открытом контексте, но вполне корректно читаются в контексте предложения DSL. При именовании DSL на первое место выступает предложение, а элементы именуются с учетом встраивания в этот контекст. DSL-имена создаются с учетом контекста конкретного предметно-ориентированного языка, в то время как имена командных запросов создаются для работы вне всякого контекста (или в любом контексте, что то же самое).

4.2. Необходимость в слое синтаксического анализа

Тот факт, что свободный интерфейс отличается от интерфейса командных запросов, может привести к осложнениям. Если смешать оба стиля интерфейса в одном и том же классе, это может запутать пользователя. Поэтому я сторонник отделения элементов DSL от обычных объектов командных запросов путем создания слоя построителей выражений (*Expression Builder* (349)) над обычными объектами. Построители выражений представляют собой объекты, единственной задачей которых является создание модели обычных объектов с использованием свободного интерфейса — трансляции свободных предложений в последовательность вызовов API командных запросов.

Одной из причин использования построителей выражений является различный характер интерфейсов, но основная причина — классический аргумент разделения понятий. Как только вы вводите какой-то язык, пусть даже внутренний, вы должны написать код, который понимает этот язык. Такой код зачастую будет нуждаться в отслеживании данных, которые имеют смысл только во время обработки языка, — данных синтаксического анализа. Понимание того, как работает внутренний DSL, требует определенного объема работы, который не нужен для наполнения базовой модели. Вам не нужно понимать ни DSL, ни как он работает, чтобы понять, как работает базовая модель, поэтому код обработки языка стоит располагать в отдельном слое.

Эта структура следует общей схеме обработки DSL. Базовая модель объектов интерфейса командных запросов — семантическая модель (*Semantic Model* (171)). Слой построителей выражений представляет собой часть синтаксического анализатора.

Определенные трудности вызывает употребление термина “синтаксический анализатор” для этого слоя построителей выражений. Обычно мы используем термин “анализатор” в контексте синтаксического анализа текста. В данном случае с текстом работает синтаксический анализатор базового языка. Но имеется много параллелей между тем, что мы делаем с помощью построителей выражений, и деятельностью анализатора. Ключевое отличие заключается в том, что, в то время как традиционный синтаксический анализатор преобразует поток токенов в синтаксическое дерево, входом для построителей выражений является поток вызовов функций. Параллели с другими синтаксическими анализаторами заключаются в том, что мы по-прежнему рассматриваем организацию вызовов функций в виде узлов дерева, используем структуры данных, аналогичные структурам данных син-

таксического анализа (такие, как таблицы символов (Symbol Table (177))), и по-прежнему заполняем семантическую модель.

Отделение семантической модели от построителей выражений привносит обычные преимущества семантической модели. Вы можете независимо тестировать построители выражений и семантическую модель. У вас может быть несколько синтаксических анализаторов, комбинация внутреннего и внешнего DSL или поддержка нескольких внутренних DSL несколькими построителями выражений. Вы можете независимо развивать построители выражений и семантическую модель. Это важно, так как DSL, как и любое другое программное обеспечение, не является жестко фиксированным. Вы должны быть в состоянии постоянно развивать программное обеспечение, и часто целесообразно изменить базовую платформу без внесения изменений в сценарии DSL (или наоборот).

Имеется аргумент против применения построителей выражений, но только тогда, когда объекты семантической модели объектов сами используют свободный интерфейс, а не интерфейс командных запросов. Есть несколько случаев, когда для модели имеет смысл использование свободного интерфейса: если это основной способ взаимодействия с ней. Однако в большинстве случаев я предпочитаю модель с интерфейсом командных запросов. Интерфейс командных запросов является более гибким в его использовании в различных контекстах. Свободному интерфейсу часто требуются временные данные синтаксического анализа. В частности, я возражаю против смешивания свободного интерфейса и интерфейса командных запросов у одних и тех же объектов — это оказывает-ся слишком запутанным решением.

Поэтому в оставшейся части книги я буду предполагать применение построителей выражений. Хотя я признаю, что вы можете не использовать построители выражений постоянно, я предполагаю, что в большинстве случаев вы все же будете иметь с ними дело, так что далее я буду излагать материал с учетом этого факта.

4.3. Использование функций

С самого начала применения вычислительной техники программисты стремятся упаковать общий код в некоторые повторно используемые фрагменты. Наиболее успешной конструкцией оказались функции (именуемые также подпрограммами, процедурами, а также — в объектно-ориентированном программировании — методами). API командных запросов, как правило, выражается в функциях, но и структуры DSL также часто построены на функциях. Разница между интерфейсом командных запросов и DSL в том, как комбинируются эти функции.

Имеется целый ряд шаблонов объединения функций для создания DSL. В начале этой главы я показал два из них. Давайте освежим их в памяти. Начнем со связывания методов в цепочки (Method Chaining (375)).

```
computer()
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size(150)
    .disk()
        .size(75)
        .speed(7200)
        .sata()
    .end();
```

Затем приведем метод последовательностей функций (Function Sequence (357)).

```
computer();
processor();
cores(2);
speed(2500);
i386();
disk();
size(150);
disk();
size(75);
speed(7200);
sata();
```

Это разные шаблоны для объединения функций, которые естественным образом приводят к вопросу “Какой же из них следует использовать?” Ответ включает много фактов. Первым из них является область видимости функций. Если вы используете связывание методов, то эти функции в DSL представляют собой методы, которые должны быть определены только для объектов, принимающих участие в цепочке, как правило для построителей выражений (Expression Builder (349)). С другой стороны, используя в последовательности “голые”, неквалифицированные функции, вы должны гарантировать их корректное разрешение. Наиболее очевидный способ состоит в использовании глобальных функций, но применение глобальных имен добавляет две новые проблемы: усложнение глобального пространства имен и применение глобальных переменных для данных синтаксического анализа.

В настоящее время хорошие программисты ужасно нервничают при одном только упоминании слова “глобальный”, поскольку глобальность затрудняет локализацию изменений. Глобальные функции будут видны в каждой части программы, но в идеальном случае хотелось бы, чтобы эти функции были доступны только там, где выполняется обработка DSL. Существуют различные языковые возможности, которые могут избавить от необходимости делать все глобальным. Так, пространства имен позволяют сделать функции выглядящими глобально только при импорте определенного пространства имен (в Java используется статический импорт).

Глобальные данные синтаксического анализа являются более серьезной проблемой. Какова бы ни была последовательность функций, вам нужно прибегать к переменным контекста (Context Variable (187)), чтобы знать, в каком месте анализируемого выражения вы находитесь. Рассмотрим вызовы `diskSize`. Необходимо знать, о размере какого из дисков идет речь, так что текущий диск отслеживается в переменной, значение которой обновляется при вызове функции `disk()`. Поскольку все функции являются глобальными, это состояние также будет носить глобальный характер. Есть разные способы работы с глобальностью — например, хранить все данные в синглтоне, — но отказаться от глобальных данных при использовании глобальных функций вы не в состоянии.

Связывание методов позволяет избежать большинства из них, потому что, хотя вам все еще нужна некоторая неквалифицированная функция для начала цепочки, как только вы ее начнете, все данные синтаксического анализа смогут храниться в объекте построителя выражений, определяющего цепочку методов.

Вы можете избежать всей этой глобальности при работе с последовательностями функций, применив шаблон Object Scoping (387). В большинстве случаев этот шаблон предполагает размещение сценария DSL в подклассе построителя выражений, так что вызовы неквалифицированных функций разрешаются с помощью методов суперкласса построителя выражений. Так можно справиться с обеими проблемами глобальности. Все функции в DSL определены только в классе построителя и, таким образом, локализова-

ны. Кроме того, поскольку это методы экземпляра, они обращаются непосредственно к данным экземпляра построителя для хранения данных синтаксического анализа. Преимущества описанного решения существенно превышают стоимость размещения сценария DSL в подклассе построителя, так что это мой выбор по умолчанию.

Еще одно преимущество указанного шаблона — в поддержке расширяемости. Если DSL позволяет легко использовать подкласс контекстного класса, пользователи DSL могут добавлять в язык собственные DSL-методы.

И последовательность функций, и связывание методов требуют использования переменных контекста для отслеживания синтаксического анализа. Третьим средством комбинирования функций являются вложенные функции (*Nested Function* (361)), которые часто позволяют избежать применения переменных контекста. Пример конфигурации компьютера с применением вложенных функций выглядит следующим образом.

```
computer(  
    processor(  
        cores(2),  
        speed(2500),  
        i386  
    ),  
    disk(  
        size(150)  
    ),  
    disk(  
        size(75),  
        speed(7200),  
        SATA  
    )  
) ;
```

Здесь комбинирование функций выполняется путем передачи вызовов функций в качестве аргументов для вызовов функций более высокого уровня. Вложенные функции обладают определенными мощными преимуществами иерархической структуры любого типа, которые часто встречаются в синтаксическом анализе. В нашем примере одним из непосредственных преимуществ является то, что иерархическая структура конфигурации отражена в языковых конструкциях — функция `disk` вложена в функцию `computer` так же, как вложены объекты получающейся в результате схемы. Вложенность функций, таким образом, отражает логическое синтаксическое дерево DSL. При применении последовательности функций и связывания методов я могу только намекнуть на синтаксическое дерево с помощью отступов в исходном тексте; вложенные функции позволяют мне отобразить дерево в языке (хотя форматирование исходного текста и в этом случае существенно отличается от того, как я форматирую обычный код).

Другим следствием является изменение порядка вычислений. При использовании вложенных функций аргументы функции вычисляются до нее самой. Это часто позволяет создавать объекты схемы без использования переменных контекста. В нашем случае функция `processor` вычисляется и может вернуть объект процессора до вычисления функции `computer`. Функция `computer` в этом случае может непосредственно создать объект компьютера с полностью сформированными параметрами.

Таким образом, вложенные функции очень хорошо работают при построении высокуюровневых структур. Однако это решение не является совершенным. Пунктуация с использованием скобок и запятых более заметна и понятна, но по сравнению с выделением с помощью отступа выглядит несколько “зашумленной”. (Здесь Lisp может дать фору другим языкам — его синтаксис очень хорошо приспособлен для работы с вложенными функциями.) Шаблон вложенных функций предполагает также использование неквали-

фицированных функций, поэтому ему свойственна та же проблема глобальности, что и последовательностям функций (которая так же решается с помощью шаблона переноса области видимости в объект).

Порядок вычисления также может привести к путанице, если вы мыслите в терминах последовательности команд, а не иерархической структуры. Простая последовательность вложенных функций приводит к вычислению в порядке, обратном порядку записи — третья (вторая (первая ())) . В то же время и последовательности функций, и связывание методов позволяют писать вызовы в том порядке, в котором они будут вычислены.

Вложенные функции также обычно проигрывают в том, что аргументы идентифицируются по позиции в вызове, а не по имени. Рассмотрим случай указания размера и скорости диска. Если все, что для этого нужно, — два целых числа, то можно обойтись вызовом `disk(75, 7200)` (к сожалению, он не помогает вспомнить, что означает каждый параметр). Проблему можно решить с помощью вложенных функций, возвращающих целое значение, и написать `disk(size(75), speed(7200))` . Этот код легче читать, но ничто не мешает мне написать `disk(speed(7200), size(75))` и получить диск, который, несомненно, меня удивит. Чтобы избежать этой проблемы, следует использовать более сложные промежуточные данные, заменив простое целое число объектом, но это достаточно раздражающее своей сложностью решение. Языки с ключевыми словами в аргументах помогают избежать проблемы, но, к сожалению, этот синтаксис встречается очень редко. Во многих отношениях связывание методов в цепочки является механизмом, который помогает добавлять ключевые слова в аргументы в языке, которому их недостает. (Немного позже я буду рассматривать шаблон Literal Map (417), который является еще одним средством преодоления отсутствия именованных параметров.)

Большинство программистов рассматривают интенсивное использование вложенных функций как необычное, но это просто отражение того, как мы применяем описанные шаблоны комбинирования функций в обычном (не DSL) программировании. Большую часть времени программисты используют последовательность функций с небольшими вкраплениями вложенных функций, а также (в объектно-ориентированном языке) связывание методов. Однако если вы программист на Lisp, то вложенные функции — это то, с чем вы постоянно сталкиваетесь. И хотя я описываю эти шаблоны в контексте написания DSL, на самом деле они представляют собой обобщенные шаблоны для комбинирования выражений.

До сих пор я описывал эти модели так, как если бы они были взаимоисключающими, но на самом деле в каждом конкретном DSL обычно используется сочетание этих (и других описанных далее) шаблонов. Каждый шаблон имеет свои сильные и слабые стороны, и различные назначения DSL приводят к разным потребностям. Вот один из возможных гибридов.

```
computer(
  processor()
    .cores(2)
    .speed(2500)
    .type(i386),
  disk()
    .size(150),
  disk()
    .size(75)
    .speed(7200)
    .iface(SATA)
);
computer(
  processor()
    .cores(4)
);
;
```

В этом сценарии DSL используются все три описанные выше шаблона. Он применяет последовательность функций для определения каждого компьютера, каждая функция `computer` использует вложенные функции для своих аргументов, каждый процессор и диск строятся с помощью связывания методов в цепочки.

Преимущество этого гибрида в том, что в каждом разделе примера используются сильные стороны каждого из шаблонов. Последовательность функций хорошо работает для определения элементов списка. Определения каждого компьютера четко отделены друг от друга. Их также легко реализовать, поскольку каждая инструкция просто добавляет полностью сформированный объект компьютера в результирующий список.

Вложенные функции для каждого компьютера исключают необходимость в переменной контекста для текущего компьютера, поскольку все аргументы будут вычислены до вызова самой функции компьютера. Если предположить, что компьютер включает процессор и переменное количество дисков, то для этого случая могут хорошо подойти списки аргументов. В общем случае применение шаблона `Nested Function` делает более безопасным использование глобальных функций, так как при этом оказывается проще организовать дело так, чтобы глобальные функции просто возвращали объекты и не изменяли состояние синтаксического анализа.

Если каждый процессор и диск имеет несколько необязательных аргументов, то здесь хорошо работает связывание методов. Так можно вызвать методы для установки тех значений, которые нужны для создания конкретного элемента.

Однако такая смесь также оказывается гремучей. В частности, смешивание шаблонов приводит к пунктуационной путанице: одни элементы разделяются запятыми, другие — точками, трети — точками с запятой. Как программист я могу понять такой текст, но при написании может оказаться трудно вспомнить, где и что применяется. Непрограммиста может испугать даже чтение такого исходного текста. Пунктуационные различия представляют собой артефакт реализации, а не значение самого DSL, так что в результате пользователям предоставляется излишняя для них информация о реализации, что всегда является достаточно подозрительной идеей.

Поэтому в нашей ситуации я, пожалуй, не стал бы использовать именно такой гибрид. Я скорее склонен применить для функции компьютера связывание методов вместо вложенных функций. Но для нескольких компьютеров я бы оставил последовательность функций, так как я думаю, что это обеспечит четкое и ясное для пользователей их отделение друг от друга.

Это обсуждение компромисса — миниатюрный пример решений, которые вы должны будете принимать при создании собственных DSL. Я могу только указать на некоторые плюсы и минусы разных шаблонов, но решать, что и в каких пропорциях применять, вы должны самостоятельно.

4.4. Коллекции литералов

Написание программы, будь то на языке общего назначения или на DSL, заключается в соединении элементов в одно целое. Обычно программы соединяют инструкции в последовательности и объединяют их с помощью функций. Еще один способ скомпоновать элементы заключается в применении шаблонов `Literal List` (415) и `Literal Map` (417).

Список литералов собирает список элементов одного и того же (или различных) типов, не фиксированного размера. На самом деле я уже использовал его ранее. Посмотри-те еще раз на версию кода конфигурации компьютера с использованием вложенных функций (`Nested Function` (361)).

```
computer(
  processor(
    cores(2),
    speed(2500),
    i386
  ),
  disk(
    size(150)
  ),
  disk(
    size(75),
    speed(7200),
    SATA
  )
);
```

Если свернуть вызовы функций нижнего уровня, то получится код наподобие следующего.

```
computer(
  processor(...),
  disk(...),
  disk(...));
;
```

Содержимое вызова функции `computer` представляет собой список элементов. В языках типа Java и C# вызов функций с переменным числом аргументов представляет собой обычный способ ввода списка литералов.

В других языках могут использоваться другие варианты. В Ruby, например, можно представить такой список с помощью встроенного синтаксиса Ruby для списков литералов.

```
computer [
  processor(...),
  disk(...),
  disk(...)]
;]
```

Если не считать квадратных скобок, здесь мало отличий от предыдущего фрагмента, но этот вид списка можно использовать и в иных контекстах, помимо вызовов функций.

В С-подобных языках есть синтаксис массива литералов `{1, 2, 3}`, который может использоваться как более гибкий список литералов, но его применение и содержимое обычно весьма ограничены. Другие языки, такие как Ruby, позволяют использовать списки литералов гораздо шире.

Языки сценариев допускают использование и второго типа коллекции литералов — или отображение (или ассоциативный массив, именуемый также хешем или словарем). С его помощью можно представить конфигурацию компьютера следующим образом (здесь вновь использован язык Ruby).

```
computer(processor(:cores => 2, :type => :i386),
         disk(:size => 150),
         disk(:size => 75,
              :speed => 7200,
              :interface => :sata))
```

Отображение литералов очень удобно в таких случаях, как приведенная здесь настройка процессора и дисков. Диск имеет несколько (необязательных) подэлементов, каждый из которых может быть установлен только один раз. Связывание методов (Method Chaining (375)) хорошо подходит для именования подэлементов, но вам нужно

добавить собственный код, чтобы обеспечить для каждого диска упоминание его скорости только один раз. Эта возможность встроена в отображение литералов и знакома людям, использующим соответствующий язык.

Еще лучшей конструкцией была бы функция с именованными параметрами. Smalltalk, например, справился бы с этим примерно так: `diskWithSize: 75 speed: 7200 interface: #sata`. Однако языков с именованными параметрами функций еще меньше, чем с синтаксисом отображения литералов. Но если вы используете такой язык, то применение именованных параметров является хорошим способом реализации отображения литералов.

Этот пример также вводит еще один синтаксический элемент — символьный тип данных. **Символ** (symbol) представляет собой тип данных, который, на первый взгляд, совпадает со строкой, предназначен, прежде всего, для поиска в отображениях, в частности в таблицах символов (*Symbol Table* (177)). Символы являются неизменяемыми и, как правило, реализуются так, что одно и то же значение символа соответствует одному и тому же объекту. Их лiteralная форма не допускает символов пробела, и они не поддерживают большинство операций над строками, так как их роль заключается не в хранении текста, а в выполнении поиска. Приведенные выше элементы, такие как `:cores`, являются символами — для указания символов Ruby использует двоеточия. В языках без этого типа данных можно вместо символов использовать строки, но если язык поддерживает символьный тип данных, то в описанных ситуациях следует применять именно его.

Сейчас наступил хороший момент, чтобы немного поговорить о том, почему Lisp столь привлекателен для реализации внутреннего DSL. Lisp имеет очень удобный синтаксис списка литералов: `(one two three)`. Тот же синтаксис используется и для вызова функции: `(max 5 14 2)`. В результате программа Lisp состоит из вложенных списков. Простые слова `(one two three)` являются символами, поэтому синтаксис языка представляет собой вложенные списки символов, что является прекрасной основой для внутренних DSL — при условии, что вас устроит DSL с этим же синтаксисом. Этот простой синтаксис одновременно является и сильной, и слабой стороной Lisp. Это сильная сторона, потому что это очень логичный и последовательный синтаксис. Слабость же его в том, что вы должны следовать этой достаточно необычной синтаксической форме. И пока вы к ней не привыкнете и она не станет близкой и понятной, вас будет раздражать все это множество глупых ненужных скобок.

4.5. Использование грамматик для выбора внутренних элементов

Как можно видеть, существует множество вариантов элементов внутренних DSL. Один из методов, которые можно использовать для их выбора, — рассмотрение логической грамматики вашего DSL. При рассмотрении могут иметь смысл типы грамматических правил, которые вы создаете при использовании синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)). Определенные типы выражений с их правилами, выраженнымными в форме Бэкуса–Наура (*BNF* (237)), предполагают применение внутренних структур DSL определенных видов.

Структура	BNF	Следует рассмотреть...
Обязательный список	<code>parent ::= first second third</code>	<code>Nested Function</code> (361)
Необязательный список	<code>parent ::= first maybeSecond? MaybeThird?</code>	<code>Method Chaining</code> (375), <code>Literal Map</code> (417)

Окончание таблицы

Структура	BNF	Следует рассмотреть...
Однородное мульти множество	parent ::= child*	Literal List (415) , Function Sequence (357)
Неоднородное мульти множество	parent ::= (this that theOther)*	Method Chaining (375)
Множество	Нет	Literal Map (417)

Если у вас есть предложение из обязательных элементов (`parent ::= first second`), то будут хорошо работать вложенные функции. Аргументы вложенной функции могут непосредственно соответствовать элементам правила. Если у вас имеется строгая типизация, то функциональная возможность **автозаполнения с учетом введенных символов** может предложить для каждой позиции корректные элементы.

Список с необязательными элементами (`parent ::= first maybeSecond? MaybeThird?`) при применении вложенных функций может привести к комбинаторному взрыву возможностей. Здесь обычно лучше работает связывание методов в цепочку, так как вызов метода указывает, какой элемент вы применяете. Самое сложное при использовании связывания методов в цепочку — обеспечение не более одного вызова метода для каждого элемента в правиле.

Предложение с несколькими объектами одного и того же подэлемента (`parent ::= child*`) хорошо работает со списком литералов. Если выражение определяет инструкции на высшем уровне вашего языка, то это одно из немногих мест, где я бы рассмотрел применение последовательности функций.

В случае нескольких объектов различных подэлементов (`parent ::= (this | that | theOther)*`) я бы вернулся к связыванию методов в цепочки, так как имя метода является хорошим указанием на конкретный элемент.

Множество подэлементов представляет собой общий случай, который не выражается в форме Бэкуса-Наура. Это ситуация, когда у вас имеется много дочерних элементов, но каждый из них может появиться не более одного раза. Его можно рассматривать и как список с обязательными элементами, в котором они могут находиться в любом порядке. Логичным выбором является отображение литералов; обычно возникающий при этом вопрос заключается в неспособности обеспечить правильные ключевые имена.

Правила грамматики в форме “как минимум один раз” (`parent ::= child+`) плохо подходят к конструкциям внутренних DSL. Лучше всего использовать общие формы с несколькими элементами и проверять наличие по крайней мере одного вызова во время синтаксического анализа.

4.6. Замыкания

Замыкания представляют собой возможность языка программирования, которая уже давно имеется в некоторых языках программирования (таких, как Lisp и Smalltalk), но только недавно начала проникать в широко распространенные языки. Для них используются разные названия — “лямбда”, “блоки”, “анонимные функции”. Вот краткое описание того, что они делают: позволяют взять некоторый встраиваемый код и упаковать его в объект, который можно передавать и вычислять там, где вам удобно. (Если вы еще никогда не сталкивались с этим понятием, обратитесь к Closure (397).)

В случае внутренних DSL мы используем замыкания в форме Nested Closure (403) в сценариях DSL. Вложенные замыкания обладают тремя свойствами, которые делают их

удобными для использования в DSL: встроенные вложения, отложенное вычисление и ограниченная область видимости переменных.

Как уже было отмечено, одной из важных возможностей вложенных функций (*Nested Function* (361)) является то, что они позволяют отражать иерархическую природу DSL способом, имеющим смысл в базовом языке программирования, а не с помощью отступов, как это приходится делать при применении последовательностей функций (*Function Sequence* (357)) и связывания методов в цепочки (*Method Chaining* (375)). Вложенные замыкания также обладают этим свойством, с тем дополнительным преимуществом, что вы можете поместить во вложение любой встроенный код (откуда и происходит термин **встраиваемое вложение** (*inline nesting*)). Большинство языков имеют ограничения на то, что вы можете передать в функцию в качестве аргументов, но вложенные замыкания позволяют разорвать эти ограничения. Таким способом вы можете вкладывать более сложные структуры, такие как последовательность функций, что невозможно при применении вложенных функций. Еще одно преимущество в том, что многие языки облегчают синтаксическое вложение нескольких строк во вложенное замыкание (в отличие от вложенной функции).

Отложенное вычисление — это, пожалуй, самая важная возможность, привносимая вложенными замыканиями. В случае применения вложенной функции аргументы включающей функции вычисляются до ее вызова. Иногда это полезно, а иногда, особенно при глубокой вложенности, это приводит к путанице. При использовании вложенного замыкания вы получаете полный контроль над временем вычисления замыканий. Вы можете изменить порядок вычислений, отказаться от некоторых вычислений вообще или хранить все замыкания для выполнения вычислений позже. Это особенно удобно, когда семантическая модель (*Semantic Model* (171)) такова, что требует строгого контроля над способом выполнения программы — форма модели, которую я называю адаптивной моделью (*Adaptive Model* (478)) и которую я опишу более подробно в главе 7, “Альтернативные вычислительные модели”, с. 127. В этих случаях DSL могут включать фрагменты базового кода и размещать эти блоки кода в семантической модели. Это позволяет более свободно смешивать DSL и базовый код.

Последним свойством является то, что вложенные замыкания позволяют ввести новые переменные, область видимости которых ограничена замыканием. Использование таких **переменных с ограниченной областью видимости** может облегчить понимание кода.

Сейчас самое время проиллюстрировать все сказанное на примере. Вот очередной пример сборки компьютера.

```
#ruby...
ComputerBuilder.build do |c|
  c.processor do |p|
    p.cores 2
    p.i386
    p.speed 2.2
  end
  c.disk do |d|
    d.size 150
  end
  c.disk do |d|
    d.size 75
    d.speed 7200
    d.sata
  end
end
```

(Здесь использован Ruby, так как Java не поддерживает замыкания, а синтаксис замыканий C# слишком многословен.)

Это хороший пример встроенной вложенности. Вызовы `processor` и `disk` содержат код с несколькими инструкциями Ruby. Это также демонстрирует ограниченность области видимости переменных для компьютера, процессора и дисков. Такие переменные, хотя и немного “зашумливают” код, делают его при этом проще для понимания и позволяют видеть объекты, с которыми выполняются действия. Это означает также, что код не нуждается в глобальных функциях или применении шаблона Object Scoping (387) таких функций, как `speed`, определенных для переменных с ограниченной областью видимости (и которые в данном случае являются построителями выражений (Expression Builder (349))).

В DSL наподобие приведенной конфигурации компьютера нет необходимости в отложенном вычислении. Это свойство замыкания более востребовано при вставке фрагментов базового кода в структуру модели.

Рассмотрим пример использования набора правил проверки. Обычно в объектно-ориентированной среде объект рассматривается как корректный или некорректный, и имеется определенный код для проверки этой корректности. Проверка часто может быть контекстной — вы проверяете возможность объекта выполнить некоторые действия. Если взять в качестве примера данные о человеке, то могут быть проверки, выясняющие, имеет ли он право на тот или иной вид страхового полиса. Я мог бы указать правила в виде DSL следующим образом.

```
// C#...
class ExampleValidation : ValidationEngineBuilder {
    protected override void build() {
        Validate("Годовой доход в наличии")
            .With(p => p.AnnualIncome != null);
        Validate("Положительный годовой доход")
            .With(p => p.AnnualIncome >= 0);
```

В этом примере функции `with` передается замыкание, которое, в свою очередь, получает в качестве аргумента объект-человек и содержит некоторый произвольный код C#. Этот код может быть сохранен в семантической модели и выполнен при работе модели, что обеспечивает большую гибкость в выборе проверок.

Вложенное замыкание — очень полезный шаблон DSL, но его часто неудобно использовать. Многие языки (например, Java) просто не поддерживают замыкания. Обойти отсутствие замыканий можно другими методами, например воспользовавшись указателями на функции в C, или командными объектами в объектно-ориентированном языке программирования. Эти методы имеют важное значение для поддержки адаптивных моделей в таких языках. Однако упомянутые механизмы требуют сложного и громоздкого синтаксиса, который может существенно “зашумливать” DSL.

Даже языки, которые поддерживают замыкания, часто делают это с помощью неудобного синтаксиса. В этом отношении C# с каждой новой версией становится все лучше, но все равно пока что его синтаксис замыканий не настолько ясен, как хотелось бы. Существенно чище и “стройнее” синтаксис замыканий в Smalltalk. Синтаксис Ruby почти столь жестроен, как и синтаксис Smalltalk, поэтому применение в нем вложенных замыканий достаточно распространено. Довольно странно, что Lisp, несмотря на первоклассную поддержку замыканий, имеет при этом неудобный синтаксис для работы с ними (основанный на применении макросов).

4.7. Работа с синтаксическим деревом

Поскольку я упомянул Lisp и его макросы, логично перейти к вопросам работы с синтаксическими деревьями (Parse Tree Manipulation (449)). Этот переход связан с макро-

сами Lisp, которые широко используются для того, чтобы облегчить работу с замыканиями; но, возможно, их величайшая сила в том, что они позволяют проделывать с кодом невероятные трюки.

Основная идея, лежащая в основе работы с синтаксическим деревом, заключается в том, чтобы получить выражение на базовом языке программирования и вместо его вычисления рассматривать соответствующее синтаксическое дерево как данные. Рассмотрим следующее выражение C#: `aPerson.Age > 18`. Если я возьму его с привязкой к переменной `aPerson` и вычислю его, то получу результат — логическое значение. Альтернатива вычислению, доступная в некоторых языках, — обработка этого выражения и получение его синтаксического дерева (рис. 4.2).

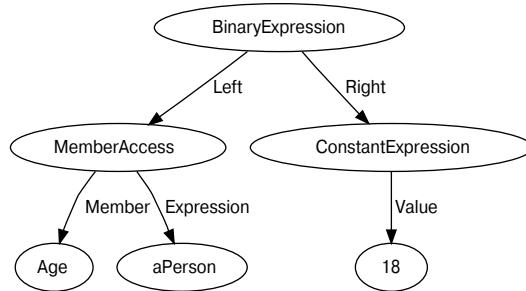


Рис. 4.2. Представление выражения `aPerson.Age > 18` в виде синтаксического дерева

При наличии такого синтаксического дерева с ним можно работать во время выполнения программы и делать всякие интересные вещи, например можно обойти синтаксическое дерево и создать запрос на другом языке, таком как SQL. Это по сути то, что делает язык .NET Linq. Он позволяет выражать SQL-запросы в C#, что предпочитают многие программисты.

Основное преимущество работы с синтаксическим деревом в том, что вы можете записывать выражения на базовом языке, которые затем могут быть преобразованы в различные выражения, наполняющие семантическую модель (*Semantic Model (171)*) **способами, которые не сводятся к простому хранению самих замыканий**.

В приведенном выше случае на языке программирования C# манипуляции синтаксическим деревом производятся с его представлением в виде объектной модели. В случае Lisp манипуляции выполняются с помощью преобразований исходного кода Lisp макросами. Lisp хорошо подходит для этого, поскольку структура его исходного кода очень близка к структуре синтаксического дерева. Работа с синтаксическими деревьями широко используется в Lisp для DSL — настолько широко, что программисты на Lisp часто сокрушаются по поводу отсутствия макросов в других языках. Что касается меня, то я считаю, что работа с объектной моделью синтаксического дерева в стиле C# эффективнее применения макросов Lisp, хотя это может быть связано с отсутствием у меня практики работы с макросами в Lisp.

Какой бы механизм вы ни использовали, возникает вопрос “Насколько важна работа с синтаксическим деревом как технология DSL?” Одним очень известным примером ее использования является Linq — технология Microsoft, которая позволяет выражать запросы на C# и преобразовывать их в различные языки запросов для множества целевых структур данных. Таким образом, один и тот же запрос на C# может быть преобразован в SQL-запрос для реляционных баз данных, в XPath — для структур XML или использоваться в C# — для структур данных C#, располагающихся в памяти. Это, по сути, меха-

низм, который позволяет коду приложения осуществлять трансляцию времени выполнения, генерируя произвольный код на основе выражений C#.

Манипуляции синтаксическим деревом — мощная, но довольно сложная технология, которая в прошлом практически не поддерживалась языками программирования, а в настоящее время привлекла гораздо больше внимания в связи с поддержкой в C# 3 и Ruby. Поскольку это относительно новая (по крайней мере за пределами мира Lisp) технология, трудно оценить, насколько она полезна на практике. Пока что мое впечатление таково, что это не слишком важная технология из тех, которые нужны крайне редко, но очень удобны тогда, когда в них возникает необходимость. Перевод с помощью Linq запросов к различным источникам данных дает прекрасный пример достоинств описанных методов; время покажет, появятся ли другие столь же успешные приложения.

4.8. Аннотации

Когда появился язык C#, многие программисты с усмешкой говорили, что это просто немного приукрашенный старый добрый Java. В определенной степени они были правы, хотя в любом случае не нужно издеваться над хорошо выполненной реализацией проверенных идей. Одним из примеров возможностей, которые не были простой копией, были **атрибуты**, функциональная возможность языка, которая позже была скопирована Java под названием “аннотации” (Annotation (439)). (Здесь я буду использовать термин из Java, поскольку термин “атрибут” в программировании слишком перегружен.)

Аннотация позволяет программисту приложить метаданные к программным конструкциям, таким как классы и методы. Она может быть прочитана во время компиляции или выполнения.

Например, предположим, что мы хотим объявить, что некоторые поля могут иметь значения только из ограниченного диапазона. С помощью аннотаций это можно сделать следующим образом.

```
class PatientVisit...
    @ValidRange(lower = 1, upper = 1000, units = Units.LB)
    private Quantity weight;
    @ValidRange(lower = 1, upper = 120, units = Units.IN)
    private Quantity height;
```

Очевидной альтернативой этому было бы размещение кода для проверки значений в модифицирующих методах. Однако аннотации имеют ряд преимуществ. Их проще читать, они упрощают проверку диапазона как при установке атрибута, так и на более позднем этапе проверки объекта и определяют правила проверки таким способом, который может быть легко считан при настройке соответствующего управляющего элемента графического интерфейса.

Некоторые языки обеспечивают возможность указания таких числовых диапазонов (насколько я помню, это сделано в Pascal). Вы можете рассматривать аннотации как способ расширения языка для поддержки новых ключевых слов и возможностей. Действительно, даже существующие ключевые слова могли бы быть сделаны лучше с аннотациями, например я считаю, что модификаторы доступа (`private`, `public` и т.д.) выиграли бы от этого.

Поскольку аннотации тесно связаны с базовым языком, они подходят для фрагментарных, но не автономных DSL. В частности, они создают ощущение высокой степени интегрированности расширений предметной области в базовый язык.

Сходство между аннотациями Java и атрибутами .NET вполне очевидно, но есть и другие языковые конструкции, которые выглядят иначе, делая по существу то же самое. Вот как выглядит способ указания верхнего предела размера строки в Ruby on Rails.

```
class Person
  validates_length_of :last_name, :maximum => 30
```

Этот синтаксис отличается тем, что вы указываете, к какому полю применяется проверка, путем предоставления имени этого поля (`:last_name`), а не путем размещения аннотации рядом с полем. Реализация также отличается тем, что это на самом деле метод класса, который выполняется при загрузке класса врабатывающую систему, а не особенность конкретного языка. Несмотря на указанные различия описанная методика все равно представляет собой добавление метаданных к элементам программы и используется подобно аннотациям. Так что, я думаю, ее стоит рассматривать как то же понятие, что и аннотация.

4.9. Расширение литералов

Одна из возможностей, которые вызвали недавний скачок интереса к DSL, является использование выражений DSL в Ruby on Rails. Типичным примером его DSL-выражений является фрагмент наподобие `5.days.ago`. Большая часть этого выражения представляет собой уже знакомое нам связывание методов в цепочку (*Method Chaining* (375)). Новизна в том, что цепочка начинается с целочисленного литерала. Хитрость здесь заключается в том, что целые числа представляются языком или стандартными библиотеками. Для того чтобы начать цепочку, подобную приведенной, нужно использовать расширение литералов (*Literal Extension* (473)). Для этого необходимо иметь возможность добавлять методы к классам внешних библиотек — возможность, которая может быть в базовом языке, но ее может и не быть. Java, например, ее не поддерживает, а C# (через методы расширения) и Ruby — поддерживают.

Одна из опасностей расширения заключается в том, что оно выполняет добавление глобально, в то время как оно должно применяться только в пределах часто ограниченного контекста использования DSL. Эта проблема Ruby усугубляется еще и тем, что не существует простого языкового механизма для выяснения, где было добавлено расширение. Язык C# решает этот вопрос, помещая методы расширения в пространство имен, которое нужно явно импортировать, чтобы иметь возможность воспользоваться этими методами.

К расширению литералов не нужно часто прибегать, но они могут быть очень удобны- ми. Эта технология позволяет полноценно настроить язык для своей предметной области.

4.10. Снижение синтаксического шума

Главное для внутренних DSL то, что они представляют собой просто выражения базового языка, написанные в удобочитаемом виде. Одним из последствий этого является то, что эти выражения приносят с собой синтаксические структуры базового языка. В каком-то смысле это хорошо, поскольку обеспечивает синтаксис, знакомый многим программистам, но кое-кто находит это раздражающим.

Один из способов уменьшить бремя этого синтаксиса — писать фрагменты DSL с использованием близкого, но не совпадающего с базовым, синтаксиса, а затем с помощью простой замены текста преобразовать написанное в базовый язык. Такая шлифовка текста (*Textual Polishing* (469)) может преобразовать фразу типа `3 hours ago` в `3.hours.ago`

или, более амбициозно, 3% if value at least \$30000 в percent(3).when.minimum(30000).

Хотя я часто встречал описание этой методики, должен признаться, что я не большой ее поклонник. Замены очень быстро становятся настолько запутанными, что легче перейти к использованию полностью внешнего DSL.

Еще один подход заключается в использовании подсветки синтаксиса. Большинство современных текстовых редакторов обеспечивают настраиваемые схемы окраски текста. Общаясь с экспертами в предметной области, можно использовать специальную схему, которая затеняет “шумы” синтаксиса, например делая их светло-серыми на белом фоне (можно пойти даже на полное их слияние с фоном).

4.11. Динамический отклик

Одним из свойств динамических языков, таких как Smalltalk или Ruby, является то, что они обрабатывают вызовы методов во время выполнения. В результате, если вы пишете `aPerson.name` и при этом метод `name` для `aPerson` не определен, код будет спокойно скомпилирован, и только во время выполнения будет обнаружена ошибка (в отличие от C# или Java, в которых такая ошибка будет обнаружена во время компиляции). Многие видят в этом проблемы, в отличие от некоторых поклонников динамических языков, которые считают это преимуществом.

Стандартный механизм, обычно используемый в этих языках для обработки таких ситуаций, — вызов специального метода. По умолчанию действие этого специального метода (именуемого `method_missing` в Ruby, `doesNotUnderstand` в Smalltalk) состоит в генерации ошибки, но программисты могут переопределить метод для выполнения других действий. Я называю это переопределение динамическим откликом (*Dynamic Reception* (423)), так как вы делаете динамический (времени выполнения) выбор относительно того, какой корректный метод следует получить. Динамический отклик может привести к ряду полезных идиом программирования, в частности при использовании прокси, когда вы хотите “завернуть” объект и что-то делать при вызовах его методов без необходимости знать, какие именно методы были вызваны.

В работе DSL динамический отклик обычно используется для переноса информации из аргументов метода в само его имя. Хорошим примером являются динамические искатели Rails Active Record. Если у вас есть класс человека с полем `firstname`, можете искать людей по их именам. Вместо того чтобы определять метод `find` для каждого поля, у вас может быть обобщенный метод поиска, который принимает в качестве аргумента имя поля: `people.find_by("firstname", "martin")`. Это работает, но выглядит немного странно, так как вы ожидаете, что `"firstname"` должно быть частью имени метода, а не параметром. Динамический отклик позволяет написать `people.find_by_firstname("martin")` без определения соответствующего метода заранее. Вы перекрываете обработчик отсутствующего метода и выясняете, не начинается ли имя этого метода с `find_by`, извлекаете имя поля и превращаете указанный вызов в вызов полностью параметризованного метода. Все это может быть сделано в одном методе или в нескольких, или в отдельных методах, таких как `people.find_by.firstname("martin")`.

Суть динамического отклика — в возможности перемещения информации из параметров в имя метода, что в некоторых случаях может сделать выражения более легкими для чтения и понимания. Неприятность заключается в том, что вряд ли вы захотите выполнять сложный поиск самостоятельно, разбираясь при необходимости в последовательности имен методов. Если вам нужно что-то более сложное, чем работа с простым

списком, рассмотрите возможность применения чего-то более структурированного (типа вложенных функций ([Nested Function \(361\)](#)) или замыканий ([Nested Closure \(403\)](#))). Динамический отклик работает лучше, когда вы выполняете одну и ту же базовую обработку каждого вызова, например создаете запросы на основе имен свойств. Если вы обрабатываете динамически полученные вызовы по-разному (например, у вас есть различные коды для обработки имени и фамилии), используйте явные методы, не полагаясь на динамический отклик.

4.12. Проверка типов

Рассмотрев некоторые вопросы, требующие применения динамического языка программирования, перейдем к статическим языкам и посмотрим, как извлечь выгоду из статической проверки типов.

Это длительный, потенциально бесконечный спор о том, что лучше: иметь статическую проверку типов в языке или не иметь. Я не хочу рассматривать здесь этот вопрос. Одни считают, что проверка типов во время компиляции очень важна и ценна, а другие утверждают, что при этом нельзя найти массу ошибок и все равно необходимы хорошие тесты.

Есть еще один аргумент в пользу использования статической типизации. Одним из преимуществ современных сред разработки является обеспечение ими отличной поддержки на основе статической типизации. Я могу ввести имя переменной, воспользоваться специальной комбинацией клавиш и получить список методов, которые можно вызывать для этой переменной в зависимости от ее типа. Интегрированная среда разработки может сделать это, потому что знает типы символов в коде.

Однако в DSL большинство подобных символов не имеют такой поддержки, поскольку их нужно представлять в виде строк или символьных типов данных и хранить в собственной таблице символов. Рассмотрим фрагмент на языке Ruby для приведенного выше примера системы безопасности (с. 39).

```
state :waitingForLight do
  transitions :lightOn => :unlockedPanel
end
```

Здесь `:waitingForLight` является символьным типом данных. Если перевести этот код на Java, то получится что-то вроде

```
state("waitingForLight")
  .transition("lightOn").to("unlockedPanel");
```

Наши символы представляют собой просто примитивные строки. Чтобы воспользоваться связыванием методов в цепочки, необходимо завернуть `waitingForLight` в метод. Вводя в интегрированной среде разработки целевое состояние, нужно ввести `unlockedPanel` вручную, а не выбирать его из списка, предоставляемого механизмом автозаполнения.

Так что я предпочел бы следующий код.

```
waitingForLight
  .transition(lightOn).to(unlockedPanel)
;
```

Он не только более удобочитаем благодаря отсутствию метода `state` и кавычек; я также могу получить с помощью механизма автозаполнения список запускаемых событий и целевых состояний. Можно также в полной мере использовать возможности интегрированной среды разработки.

Для этого нужен способ объявления символьных типов (таких, как `state`, `command`, `event`) в механизме обработки DSL и символов, используемых в конкретном сценарии DSL (таких, как `lightOn` или `waitingForLight`). Одним из способов сделать это является класс таблицы символов (Class Symbol Table (461)). При этом подходе каждый символьный тип определяется как класс. В ходе написания сценария я помещаю его в класс и объявляю поля для своих символов. Так, чтобы определить список состояний, я начинаю с создания класса `States` для символьного типа. Я определяю состояния, применяемые в сценарии, путем объявления полей.

```
Class BasicStateMachine...
    States idle, active, waitingForLight,
           waitingForDrawer, unlockedPanel;
```

Результат, как и многие конструкции DSL, выглядит довольно странно. Обычно я никогда не выступаю за применение множественных имен для класса, как сделано здесь для `States`. Но в данном случае это приводит к стилю редактирования, который будет более тесно увязан с общим опытом программирования на языке программирования Java.

Глава 5

Реализация внешнего DSL

Используя внутренний DSL, можно сделать для определения языка многое, но всегда будет ограничены необходимостью соответствия синтаксической структуре базового языка. Внешние DSL обеспечивают большую синтаксическую свободу — возможность использовать любой понравившийся вам синтаксис.

Реализация внешних DSL отличается от реализации внутренних DSL тем, что процесс синтаксического анализа работает исключительно с текстовыми входными данными, не ограниченными каким-либо конкретным языком. Методы, которые можно использовать для синтаксического анализа текста, по сути те же, которые применялись для синтаксического анализа языков программирования на протяжении десятилетий. Имеется также давно работающее языковое сообщество, развивающее соответствующие инструменты и технологии.

Но есть одна загвоздка. Инструменты и труды этого сообщества почти всегда предполагают, что вы работаете с языком общего назначения. DSL в лучшем случае упоминаются мимоходом. Хотя многие из принципов в равной степени относятся как к языкам программирования общего назначения, так и к предметно-ориентированным языкам, есть и определенные различия. Кроме того, для работы с DSL не нужен такой большой объем знаний, как для работы с языками программирования общего назначения, так что изучать весь материал для создания предметно-ориентированного языка необязательно.

5.1. Стратегия синтаксического анализа

Выполняя синтаксический анализ внешнего DSL, необходимо получить поток исходного текста и разбить его на некоторые структуры, которые можно использовать для выяснения, что же гласит этот текст. Такое первоначальное структурирование называется синтаксическим анализом. Рассмотрим следующий код, который может быть вариацией программы описанного в главе 1, “Вводный пример” (с. 29), конечного автомата.

```
event doorClosed D1CL
event drawerOpened D2OP
command unlockPanel PNUL
command lockPanel PNLK
```

Синтаксический анализ заключается в распознавании того, что строка `event doorClosed D1CL` представляет собой определение события и говорит об этом отдельно от определения команды.

Простейший способ сделать это — это способ, который, я уверен, вы уже применяли сами, даже если никогда раньше не занимались синтаксическим анализом всерьез. Разделите введенный текст на строки и обработайте каждую строку. Если она начинается со слова `event`, значит, это событие; если она начинается с `command`, то это команда. Затем

можно соответствующим образом разбить строку и получить из нее важную информацию. Этот стиль я называю управляемой разделителями трансляцией (*Delimiter-Directed Translation* (213)). Общая идея заключается в выборе некоторых символов-разделителей, которые разбивают ввод на инструкции (обычно — символ новой строки), разбиении введенного текста на инструкции с применением данного разделителя и последующей обработкой каждой инструкции по отдельности для выяснения, что же она означает. Как правило, в строке имеется некоторый очевидный маркер, указывающий, с какого рода инструкцией приходится иметь дело.

Трансляция, управляемая разделителями, очень проста в применении и использует инструменты, с которыми знакомо большинство программистов — разбиение строк и регулярные выражения. Ограничением этого метода является то, что он не предоставляет способа обработки иерархического контекста входной информации.

Предположим, что определения сформулированы следующим образом.

```
events
  doorClosed D1CL
  drawerOpened D2OP
end

commands
  unlockPanel PNUL
  lockPanel PNLK
end
```

Теперь разбиения на строки недостаточно. Информации в строке `doorClosed D1CL` не хватает, чтобы сказать, что определено — событие или команда. Это можно сделать различными способами (и я расскажу об одном в примере к управляемой разделителями трансляции), но вам придется заняться этим самостоятельно. Чем более иерархический контекст вы получаете, тем больше усилий требуется затратить на самостоятельное решение.

Следующий шаг в обработке DSL с такого рода структурой заключается в использовании синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)). Определим сначала формальную грамматику входного языка, например так.

```
list      : eventList  commandList,
eventList : 'events'  eventDec* 'end';
eventDec  : identifier identifier;
commandList : 'commands' commandDec* 'end';
commandDec : identifier identifier;
```

Если вы читали какую-нибудь книгу о языках программирования, то сталкивались с понятием грамматики. Грамматика — это способ определения корректного синтаксиса языка программирования. Грамматики почти всегда записываются в том или ином варианте формы Бэкуса–Наура (BNF (237)). Каждая строка представляет собой правило (продукцию); в ней содержится имя, за которым следуют корректные элементы правила. Так, в приведенном выше примере строка `list : eventList commandList;` гласит, что элемент `list` состоит из `eventList`, за которым следует `commandList`. Элементы в кавычках представляют собой литералы, а `“*”` указывает, что предыдущий элемент может повторяться несколько раз. Так, продукция `eventList : 'events' eventDec* 'end';` гласит, что список событий состоит из слова `events`, за которым следует некоторое количество `eventDec` и слово `end`.

Грамматика — хороший способ представления синтаксиса языка независимо от того, используете ли вы синтаксически управляемую трансляцию. На самом деле это полезно и для внутренних DSL, как показано в таблице внутренних элементов DSL в разделе

“Использование грамматик для выбора внутренних элементов” на с. 95. Особенно хорошо она работает в случае синтаксически управляемой трансляции, поскольку при этом ее можно механически преобразовать в синтаксический анализатор.

Синтаксические анализаторы, генерируемые на основе синтаксически управляемой трансляции, очень хорошо обрабатывают иерархические структуры наподобие приведенных; в конце концов, такого рода вещи имеют важное значение для языков общего назначения. В результате можно легко и просто справляться с тем, что крайне сложно сделать с помощью управляемыми разделителями трансляции.

Как перейти от грамматики к синтаксическому анализатору? Как упоминалось ранее, это практически механический процесс, и имеется много способов преобразовать BNF в некоторый алгоритм синтаксического анализа. Этому вопросу посвящены многолетние исследования и разработано множество различных технологий. Для этой книги я отобрал три общих подхода.

Синтаксический анализатор на основе рекурсивного спуска (*Recursive Descent Parser* (253)) является классическим способом выполнения этого преобразования. Алгоритм рекурсивного спуска — легко понятный подход к синтаксическому анализу, который принимает каждое правило грамматики и **представляет его в виде потока управления внутри функции**. Каждое правило грамматики превращается в функцию синтаксического анализатора, и имеются точные шаблоны, которым нужно следовать, чтобы превратить каждый оператор BNF в поток управления.

Более интересным и современным способом является использование комбинатора синтаксических анализаторов (*Parser Combinator* (263)). Здесь каждое правило превращается в объект, и мы объединяем объекты в структуру, которая отражает грамматику. Нам все еще нужны элементы синтаксического анализатора рекурсивного спуска, но они упакованы в объекты комбинатора, которые можно просто скомпоновать. Это позволяет реализовывать грамматику без детальных знаний алгоритмов синтаксического анализатора рекурсивного спуска.

Третий вариант позволяет справиться с большинством описанных в этой книге задач. Генератор синтаксических анализаторов (*Parser Generator* (277)) принимает разновидность BNF и использует ее в качестве собственного DSL. Вы описываете грамматику с помощью этого DSL, и генератор синтаксических анализаторов создает соответствующий синтаксический анализатор.

Генератор синтаксических анализаторов представляет собой наиболее интеллектуальный подход, соответствующие инструменты весьма хорошо продуманы и могут очень эффективно обрабатывать сложные языки. Использование BNF в качестве DSL позволяет легко понимать и поддерживать язык, так как его синтаксис четко определен и автоматически связан с синтаксическим анализатором. С другой стороны, они требуют времени на обучение и, так как они в основном используют генерацию кода, это приводит к усложнению процесса построения. Кроме того, для используемой вами платформы может не быть хорошего генератора синтаксических анализаторов, а написать его самостоятельно — задача не из тривиальных.

Синтаксический анализатор рекурсивного спуска может быть менее мощным и эффективным, но его мощности и эффективности достаточно для предметно-ориентированных языков. Таким образом, он оказывается вполне разумным вариантом, если генератор синтаксических анализаторов недоступен или кажется слишком тяжеловесным для такой задачи. Самая большая проблема синтаксического анализатора рекурсивного спуска в том, что грамматика теряется в потоке управления, а это делает код гораздо менее понятным, чем хотелось бы.

Поэтому я предпочитаю там, где нельзя или не хочется использовать генератор синтаксических анализаторов, прибегать к помощи комбинатора синтаксических анализаторов. Он следует в основном тому же алгоритму, что и синтаксический анализатор рекурсивного спуска, но позволяет явно представлять грамматику в коде, который объединяет комбинаторы. Хотя этот код может быть не столь понятен, как BNF, все же он может быть достаточно близок к нему, в частности, если вы используете технологии внутренних DSL.

При использовании любого из описанных трех методов синтаксически управляемая трансляция позволяет существенно легче обрабатывать языки с той или иной структурой, чем трансляция, управляемая разделителями. Самый большой недостаток синтаксически управляемой трансляции в том, что этот метод не так широко известен, как он того заслуживает. Многие полагают, что он довольно сложен в использовании. Я думаю, что этот страх часто вызван тем, что синтаксически управляемая трансляция описана, как правило, в контексте синтаксического анализа языка общего назначения, работать с которым действительно гораздо сложнее, чем с DSL. Я надеюсь, что, прочтя эту книгу, вы отважитесь поработать с синтаксически управляемой трансляцией, после чего обнаружите, что не так страшен черт, как его малют.

В основном в этой книге я буду использовать генератор синтаксических анализаторов. Я считаю, что продуманность и зрелость инструментария и явно указанная грамматика упрощают разговор о различных объясняемых мною концепциях. В частности, я использую генератор синтаксических анализаторов ANTLR — хорошо проработанный, широко доступный инструмент с открытым исходным кодом. Одним из его преимуществ является то, что он представляет собой сложную форму синтаксического анализатора рекурсивного спуска, а это означает, что он хорошо обеспечивает понимание, которого можно достичь с помощью синтаксического анализатора рекурсивного спуска или комбинатора синтаксических анализаторов. Особенно хорош ANTLR в качестве первого шага для новичка в этой области.

5.2. Стратегия получения вывода

Для того чтобы проанализировать некоторый ввод, необходимо знать, что мы хотим получить в результате, каким должен быть выход. Я уже утверждал, что в большинстве случаев вывод должен представлять собой семантическую модель (*Semantic Model* (171)), которую затем можно либо непосредственно интерпретировать, либо использовать в качестве входных данных для генерации кода. Я не буду повторяться, укажу только, что это существенное отличие от исходных предположений, принятых в языковом сообществе.

В рамках этого сообщества имеется сильный акцент на генерации кода, и синтаксические анализаторы обычно создаются для генерации выходного кода без какой-либо обозримой семантической модели. Это разумный подход для языков общего назначения, но для DSL я считаю его неподходящим. Об этом различии очень важно постоянно помнить, читая материалы языкового сообщества, в которое входит и большая часть документации для такого инструментария, как генераторы синтаксических анализаторов (*Parser Generator* (277)).

Учитывая, что наша продукция является семантической моделью, наши варианты сводятся к одному или двум шагам. Одношаговый путь представляет собой встроенную трансляцию (*Embedded Translation* (305)), когда вызовы для создания семантической модели в процессе синтаксического анализа размещаются непосредственно в синтаксическом анализаторе. При таком подходе в процессе синтаксического анализа происходит постепенное построение семантической модели. Как только разобрана достаточная для распознавания части семантической модели входная информация, тут же создается эта часть. Зачастую же

для того, чтобы в действительности создавать объекты семантической модели, требуются некоторые промежуточные данные синтаксического анализа, — для этого обычно приходится хранить информацию в таблицах символов (*Symbol Table* (177)).

Альтернативный двухшаговый подход — построение дерева (*Tree Construction* (289)). При таком подходе выполняется синтаксический анализ вводимого текста и создается синтаксическое дерево, которое накапливает основные структуры этого текста. Одновременно заполняется таблица символов для обработки перекрестных ссылок между различными частями дерева. Затем наступает очередь второго этапа, состоящего в обходе синтаксического дерева и заполнении семантической модели.

Большим преимуществом использования метода построения дерева является разбиение задачи синтаксического анализа на две более простые задачи. При распознавании входного текста можно сосредоточиться только на построении синтаксического дерева. Действительно, многие генераторы синтаксических анализаторов обеспечивают построение дерева для DSL, что еще больше упрощает эту часть процесса. Обход дерева для заполнения семантической модели оказывается более привычной задачей для программиста; кроме того, у него в этот момент имеется полное дерево, которое можно исследовать, чтобы определить, что и как следует делать. Если вы когда-либо писали код для обработки XML, то можете сравнить встроенную трансляцию с применением SAX, а построение дерева — с использованием DOM.

Имеется и третий вариант — встроенная интерпретация (*Embedded Interpretation* (311)). Эта технология запускает процесс интерпретации во время самого синтаксического анализа, а его выход является искомым конечным результатом. Классический пример вложенной интерпретации — калькулятор, который получает арифметические выражения и выдает результаты из вычислений. Таким образом, встроенная интерпретация не производит семантической модели. Хотя встроенная интерпретация и встречается время от времени на практике, это случается крайне редко.

Можно использовать и встроенную трансляцию, и построение дерева без семантической модели; более того, при применении генерации кода это достаточно распространено. В большинстве примеров применения генераторов синтаксических анализаторов поступают именно таким образом. Хотя это может иметь смысл, особенно для более простых случаев, я рекомендую этот подход лишь в редких случаях. Обычно я считаю чрезвычайно полезным применение семантических моделей.

Итак, в большинстве случаев выбор состоит в предпочтении встроенной трансляции или построения дерева. Решение зависит от затрат на построение такого промежуточного синтаксического дерева и выгод, получаемых от его использования. Большшим преимуществом построения дерева является то, что задача синтаксического анализа распадается на две части. Обычно легче объединить решение двух простых задач, чем решать одну сложную. Это утверждение становится все более верным по мере возрастания сложности полной задачи трансляции. Чем более сложным становится DSL, и чем больше расстояние между DSL и семантической моделью, тем более полезным становится построение промежуточного синтаксического дерева, в особенности если у вас есть инструментальная поддержка создания абстрактного синтаксического дерева.

Я думаю, что все сказанное — убедительный аргумент в пользу построения дерева. Распространенный аргумент против — занимаемая синтаксическим деревом память — для небольших DSL на современном оборудовании представляется смехотворным. Но несмотря на множество доводов в пользу построения дерева, я убежден в этом решении не до конца — порой мне кажется, что создание и обход дерева вызывает больше проблем, чем преимуществ. Я должен написать код для создания дерева и код для его обхода, и чаще проще просто собрать семантическую модель без всякого дерева.

Так что я сам не могу сделать однозначный выбор. Сравнение разных методов вызывает у меня смешанные чувства, и я никак не могу прийти к окончательному решению. Наилучший совет, который я могу дать, — попробовать и то, и другое и посмотреть, что вам больше нравится.

5.3. Концепции синтаксического анализа

Начав читать о синтаксическом анализе и применении генераторов синтаксических анализаторов (*Parser Generator* (277)), вы очень быстро встретитесь с целым букетом фундаментальных понятий из этой области. Для того чтобы понимать, что такое синтаксически управляемая трансляция (*Syntax-Directed Translation* (229)), нужно разбираться в множестве фундаментальных понятий, хотя и не в той мере, как описано в традиционных книгах, посвященных компиляции и компиляторам (так как мы имеем дело с DSL, а не с языками общего назначения).

5.3.1. Лексический анализ

Обычно синтаксически управляемая трансляция (*Syntax-Directed Translation* (229)) подразделяется на два этапа: лексический анализ (известный также как сканирование или токенизация) и синтаксический анализ (используется еще какое название, как грамматический разбор (*parsing*)). На лексическом этапе введенный текст преобразуется в поток токенов. Токены представляют собой тип данных с двумя основными атрибутами: типом и содержанием. В нашем языке конечного автомата текст `state idle` превращается в два токена.

```
[content: "state", type: state-keyword]
[content: "idle", type: identifier]
```

Довольно легко написать лексический анализатор с использованием таблицы регулярных выражений (*Regex Table Lexer* (247)). Это просто список правил, которые со-поставляют регулярные выражения типам токенов. Вы читаете входной поток, находите первое регулярное выражение, соответствующее считанной информации, создаете токен соответствующего типа и переходите к следующей части потока.

Затем синтаксический анализатор берет получившийся поток токенов и организует его в синтаксическое дерево на основе правил грамматики. Однако тот факт, что лексический анализатор выполняет свою работу первым, имеет некоторые серьезные последствия. Во-первых, это означает, что при использовании текста следует быть осторожным. Например, у меня могло бы быть объявление состояния `state initial state`. Это нехорошее название, так как по умолчанию лексический анализатор будет классифицировать второе слово “`state`” как ключевое слово, а не как идентификатор. Чтобы избежать этого, я должен использовать некоторую схему альтернативной токенизации (*Alternative Tokenization* (325)). Существуют различные способы альтернативного разбиения входного потока на токены, существенно зависящие от используемого синтаксического анализатора.

Вторым следствием этого является то, что пробелы в общем случае отбрасываются до того, как к работе приступит синтаксический анализатор. Это затрудняет работу с синтаксическими пробелами. **Синтаксический пробел** — это пробельный символ, который является частью синтаксиса языка, такой, например, как символ новой строки в качестве разделителя инструкций (разделители новой строки (*Newline Separators* (339))) или отступы, используемые для указания структуры в стиле языка программирования Python.

Синтаксические пробелы — весьма запутанная область, так как в ней смешиваются синтаксическая структура языка и форматирование. В некоторой степени это может иметь смысл — так, форматирование текста позволяет “на глаз” судить о его структуре. Однако достаточно часто это приводит к осложнениям. Вот почему многие программисты, связанные с разработкой языков программирования, просто ненавидят синтаксические пробелы. Я включил в книгу некоторую информацию о разделителях новой строки как о распространенной форме синтаксических пробелов, но мне не хватает времени, чтобы углубиться в эту тему. Все, на что меня хватило, — это несколько заметок в главе 31, “Прочие вопросы”, с. 343.

Причина выделения лексического анализатора в том, что гораздо легче написать каждый из двух элементов, чем один общий. Это частный случай разложения сложной задачи на более простые. Такое разделение также повышает производительность, особенно на ограниченных аппаратных платформах, для которых первоначально предназначались многие из указанных инструментов.

5.3.2. Грамматики и языки

Должен заметить, что я говорил о грамматике языка вообще, в то время как многие полагают, что для конкретного языка следует говорить о конкретной грамматике.¹ Почекумо-то у них складывается неправильное представление о единственности грамматики языка. Хотя грамматика формально определяет синтаксис языка, не так уж сложно создать несколько грамматик, распознавающих один и тот же язык.

Давайте обратимся к входному тексту рассматривавшейся ранее системы безопасности.

```
events
  doorClosed D1CL
  drawOpened D2OP
end
```

Я могу написать для этого входа грамматику наподобие следующей.

```
eventBlock : Event-keyword eventDec* End-keyword;
eventDec   : Identifier Identifier;
```

Но я могу написать и такую грамматику.

```
eventBlock : Event-keyword eventList End-keyword;
eventList  : eventDec*
eventDec   : Identifier Identifier;
```

Оба варианта являются корректными грамматиками для данного языка. Они обе распознают входной текст и преобразуют его в синтаксическое дерево. Получающиеся в результате синтаксические деревья будут различными и, таким образом, будет отличаться и генерация кода.

Есть много причин, по которым вы можете получать различные грамматики. Основной из них является то, что различные генераторы синтаксических анализаторов (*Parser Generator* (277)) используют различные с точки зрения синтаксиса и семантики грамматики. Даже при применении одного генератора синтаксических анализаторов у вас могут быть разные грамматики в зависимости от того, как вы разлагаете свои правила на составные части (пример этого я привел выше). Точно так же, как и для любого другого кода, вы можете выполнять рефакторинг грамматик, чтобы сделать их более простыми для понимания.

¹ В оригинале обыгрывается наличие в английском языке неопределенного (*a*) и определенного (*the*) артиклей. — Примеч. пер.

Другой аспект, который влияет на разложение вашей грамматики на правила, — это получающийся код; у меня часто возникали ситуации, когда я менял грамматику для того, чтобы упростить код, транслирующий исходный текст в семантическую модель.

5.3.3. Регулярные, контекстно-свободные и контекстно-зависимые грамматики

Сейчас наступил неплохой момент для того, чтобы немного погрузиться в теорию языков, в частности в то, как языковое сообщество классифицирует грамматики. Эта схема, именуемая **иерархией Хомского**, была разработана в 1950-х годах лингвистом Ноамом Хомским (Noam Chomsky). Она основана на наблюдениях за естественными языками, а не за языками программирования, но при этом классификация использует математические свойства грамматики, применяемые для определения ее синтаксической структуры.

Нас интересуют три категории грамматик — регулярные, контекстно-свободные и контекстно-зависимые. Они образуют иерархию в том плане, что все грамматики, которые являются регулярными, — контекстно-свободны, а все грамматики, которые являются контекстно-свободными, — контекстно-зависимы. Иерархия Хомского относится только к грамматикам, но программисты используют ее и для языков. Сказать, что язык является регулярным, означает, что вы можете написать для него регулярную грамматику.

Разница между классами зависит от определенных математических характеристик грамматики. Я оставлю пояснение тонкостей для специализированных книг, посвященных языкам; для наших целей ключевое различие заключается в том, какой из фундаментальных алгоритмов необходим для синтаксического анализатора.

Регулярная грамматика очень важна для нас, поскольку ее можно обработать с помощью конечного автомата. Это важно, поскольку регулярные выражения представляют собой конечные автоматы, а следовательно, регулярный язык может быть проанализирован с использованием регулярных выражений.

С точки зрения компьютерных языков регулярные грамматики имеют один большой недостаток: они не могут справиться с вложенными элементами. Регулярный язык может разобрать выражение наподобие $1+2*3+4$, но не сможет справиться с $1+(2*(3+4))$. Вы могли слышать, что регулярные грамматики “не умеют считать”. В терминах синтаксического анализа это означает, что вы не можете создать конечный автомат для анализа языка, который имеет вложенные блоки. Очевидно, что для компьютерных языков этого мало, так как любой язык общего назначения должен иметь возможность выполнять арифметические действия. Это также влияет и на блочную структуру — программа наподобие

```
for (int i in numbers) {
    if (isInteresting(i)) {
        doSomething(i);
    }
}
```

нуждается во вложенных блоках, так что она не является регулярной.

Для обработки вложенных блоков необходимо перейти к контекстно-свободной грамматике. Я считаю это название несколько запутывающим, потому что, с моей точки зрения, контекстно-свободная грамматика добавляет иерархический контекст в вашу грамматику, что позволяет ей “считать”. **Контекстно-свободная грамматика** может быть реализована с помощью **стекового конечного автомата**, который является конечным автоматом с использованием стека. Контекстно-свободные грамматики применяют большинство синтаксических анализаторов языка, а также большинство генераторов синтаксических анализаторов (*Parser Generator* (277)), а синтаксический анализатор рекурсивного спуска (*Recursive Descent Parser* (253)) и комбинатор синтаксических анализаторов (*Parser*

Combinator (263)) генерируют стековые конечные автоматы. В результате большинство современных языков программирования анализируется с помощью контекстно-свободных грамматик.

Хотя контекстно-свободные грамматики используются очень широко, они не в состоянии обрабатывать все синтаксические правила, которые хотелось бы. Распространенной неприятной ситуацией для контекстно-свободных грамматик является правило, согласно которому сначала следует объявить переменную, а уже затем ее использовать. Проблема в том, что объявление переменной часто происходит за пределами конкретной ветви иерархии, в которой вы находитесь при использовании этой переменной. Хотя контекстно-свободная грамматика может хранить иерархический контекст, для обработки данного случая контекста не хватает, так что необходимо прибегать к таблицам символов (Symbol Table (177)).

Следующим шагом в иерархии Хомского являются контекстно-зависимые грамматики. Контекстно-зависимая грамматика может справиться с этой ситуацией, но мы не знаем, как в общем случае написать контекстно- зависимый синтаксический анализатор. В частности, мы не знаем, как сгенерировать синтаксический анализатор контекстно- зависимой грамматики.

Такое погружение в теорию классификации языков сделано, в первую очередь, потому, что это дает некоторое представление об инструментах, использующихся для обработки DSL. В частности, оно говорит, что при использовании вложенных блоков нужно нечто такое, что в состоянии работать с контекстно-свободными языками. Оно также утверждает, что если вам нужны вложенные блоки, то лучше всего воспользоваться синтаксически управляемой трансляцией (Syntax-Directed Translation (229)), а не трансляцией, управляемой разделителями (Delimiter-Directed Translation (213)).

Предполагается также, что если у вас есть только регулярный язык, то вам не нужен стековый конечный автомат для ее обработки. Впрочем, может оказаться, что вам будет проще использовать стековый конечный автомат в любом случае. Такие конечные автоматы достаточно просты, чтобы, привыкнув к ним, использовать их даже для регулярного языка.

Разделение языков — это одна из причин отделения лексического анализа от синтаксического. Лексический анализ обычно выполняется конечным автоматом, в то время как для синтаксического анализа используются стековые конечные автоматы. Это ограничивает возможности лексических анализаторов, но делает их быстрее. Как всегда, имеются и исключения; в частности, для большинства примеров в этой книге я применяю ANTLR, а ANTLR использует стековые конечные автоматы и для лексического, и для синтаксического анализа.

Существуют инструменты синтаксического анализа, которые работают только с регулярными грамматиками. Одним из наиболее известных примеров является Ragel. Кроме того, вы можете использовать для распознавания регулярных грамматик лексические анализаторы. Однако если вы занялись синтаксически управляемой трансляцией, то я бы предложил вам начать с контекстно-свободных инструментов.

Хотя понятия регулярных и контекстно-свободных грамматик наиболее распространены, есть и представляющая определенный интерес относительно новая форма грамматики — **грамматика синтаксического анализа выражений** (Parsing Expression Grammar — PEG). Это отдельная форма грамматики, которая в состоянии обработать большинство контекстно-свободных ситуаций и некоторые контекстно-зависимые. Синтаксические анализаторы PEG, как правило, не отделяют лексические анализаторы, и кажется, что во многих ситуациях PEG более приемлема, чем контекстно-свободные грамматики. Однако в настоящее время PEG все еще относительно нова, а ее инструментов пока что мало,

и они плохо отработаны. Все, конечно, может измениться, но по этой причине я пока что не рассматриваю PEG в своей книге. Наиболее известными синтаксическими анализаторами PEG являются синтаксические анализаторы Packrat.

(Граница между PEG и более традиционными синтаксическими анализаторами, впрочем, не столь уж непроницаема. Так, ANTLR, например, вобрал в себя многие идеи из PEG.)

5.3.4. Нисходящий и восходящий синтаксический анализ

Есть много способов написания синтаксического анализатора, а в результате имеется много типов генераторов синтаксических анализаторов (Parser Generator (277)) с интересными различиями. Одна из существенных отличительных особенностей синтаксического анализатора — является ли он нисходящим или восходящим. Это влияет не только на способ его работы, но и на виды грамматик, с которыми он может работать.

Нисходящий синтаксический анализатор начинает с правила грамматики наиболее высокого и использует его для поиска соответствий. Так, в грамматике списка событий

```
eventBlock : Event-keyword eventDec* End-keyword;
eventDec   : Identifier Identifier;
```

для исходного текста

```
events
  doorClosed D1CL
  drawOpened D2OP
end
```

синтаксический анализатор сначала попытается проверить соответствие исходному тексту `eventBlock` и при этом обнаружит ключевое слово `event`. Увидев ключевое слово `event`, синтаксический анализатор узнает, что дальше должно идти `eventDec`, и выполняет поиск соответствующего правила, которое говорит о том, что далее должен следовать идентификатор. Короче говоря, нисходящий синтаксический анализатор использует правила как указания, что же ему следует искать дальше.

Думаю, что не шокирую вас, если скажу, что восходящий синтаксический анализатор работает наоборот. Он начинает с чтения ключевого слова `event`, затем проверяет, достаточно ли данной входной информации для поиска соответствующего правила. Если (пока) недостаточно, он откладывает его в сторону (это называется *сдвиг*) и считывает следующий токен (идентификатор). Этого все еще недостаточно для полного соответствия правилу, поэтому снова выполняется сдвиг. После считывания второго идентификатора выясняется, что вдвоем они соответствуют правилу `eventDec`, так что теперь можно выполнить *свертку* двух идентификаторов в `eventDec`. Аналогично распознается вторая строка ввода. Затем, когда достигается ключевое слово `end`, можно свернуть все выражение в блок события.

Вы будете часто слышать, как нисходящие синтаксические анализаторы называют LL-синтаксическими анализаторами, а восходящие — LR-синтаксическими анализаторами. Первая буква указывает направление сканирования входа, а вторая — как распознаются правила (L — слева направо, т.е. сверху вниз, R — справа налево, т.е. снизу вверх). Вы также услышите, что восходящий синтаксический анализ называют синтаксическим анализом “перенос-свертка”, так как перенос-свертка — это подход, с которым вы, вероятнее всего, встретитесь в процессе восходящего синтаксического анализа. Есть несколько вариантов LR-синтаксических анализаторов, таких как LALR, GLR и SLR. Я не буду здесь вдаваться в их подробности.

Восходящие синтаксические анализаторы обычно считаются более сложными в написании и понимании, чем нисходящие. Это связано с тем, что большинству людей

труднее представить себе порядок, в котором правила обрабатываются в восходящем направлении. Хотя вы не должны беспокоиться о написании синтаксического анализатора при использовании генераторов синтаксических анализаторов, все же в целях отладки нужно хотя бы грубо представлять принцип его работы. Вероятно, самым известным семейством генераторов синтаксических анализаторов является семейство Yacc, которое генерирует восходящие (LALR) анализаторы.

Алгоритм рекурсивного спуска является алгоритмом нисходящего синтаксического анализа. В результате синтаксический анализатор рекурсивного спуска (Recursive Descent Parser (253)) является нисходящим синтаксическим анализатором, как и комбинатор синтаксических анализаторов (Parser Combinator (263)). Если этого недостаточно, то и генератор синтаксических анализаторов ANTLR основан на рекурсивном спуске, и, таким образом, генерируемые им синтаксические анализаторы являются нисходящими.

Большим недостатком нисходящих синтаксических анализаторов является то, что они не могут справиться с **левой рекурсией**, которая представляет собой правило следующего вида.

```
expr: expr '+' expr;
```

Правила, подобные этому, загоняют синтаксический анализатор в бесконечную рекурсию в попытках сопоставления `expr`. Программисты не считают это ограничение большой проблемой. Существует простой, механический метод, называемый левой факторизацией, который можно применить для избавления от левой рекурсии. Правда, в результате получаются грамматики, в которых не так легко разобраться. Хорошая новость для нисходящих анализаторов заключается в том, что вы сталкиваетесь с этой проблемой только при работе с вложенными операторными выражениями (Nested Operator Expression (333)), и как только вы поймете соответствующие идиомы, решать возникающие проблемы можно будет практически механически. В результате грамматики будут по-прежнему не такими ясными, как для восходящего синтаксического анализатора, но знание идиом позволит вам гораздо быстрее справиться с проблемами.

В общем случае разные генераторы синтаксических анализаторов имеют различные ограничения на типы грамматик, с которыми они могут работать. Эти ограничения связаны с используемыми ими алгоритмами синтаксического анализа. Есть также много других различий, например то, как вы записываете действия, как вы можете перемещать данные вверх или вниз по синтаксическому дереву и на что похож синтаксис грамматики (BNF или EBNF). Все эти факторы оказывают влияние на то, как вы пишете вашу грамматику. Возможно, наиболее важным моментом является понимание, что вы не должны относиться к грамматике как к фиксированному определению DSL. Часто необходимо изменить грамматику, чтобы улучшить получаемый результат. Как и любой другой код, грамматика будет меняться в зависимости от того, что вы хотите с ней делать.

Если вы уже хорошо знакомы с описанными концепциями, они, вероятно, сыграют важную роль в вашем решении о том, какие инструменты синтаксического анализа использовать. Для не столь “продвинутых” пользователей, вероятно, нет особой разницы, какой инструмент использовать, но их все равно полезно помнить, поскольку они влияют на способ работы с выбранным инструментом.

5.4. Смешивание с другим языком

Одна из самых больших опасностей, с которыми вы сталкиваетесь в процессе работы над внешними DSL, — ваш предметно-ориентированный язык может случайно развиться и стать языком общего назначения. Даже если дело не зайдет так далеко, DSL может

легко стать слишком сложным, в особенности если у вас есть много частных случаев, которые нуждаются в особой обработке, но при этом встречаются относительно редко.

Представим, что у нас есть DSL, который распределяет товары между продавцами на основе запросов и штатов, в которых живут клиенты. У нас могли бы быть правила наподобие следующих.

```
scott handles floor_wax in WA;
helen handles floor_wax desert_topping in AZ NM;
brian handles desert_topping in WA OR ID MT;
otherwise scott
```

Теперь представим, что Скотт начал регулярно играть в гольф с большой шишкой из Baker Industries, что дает ему привилегированный доступ ко всем компаниям с названиями, начинающимися с “Baker”: “Baker This”, “Baker That” и пр. Мы решили воспользоваться этим путем назначения ему всех покупателей товара `floor_wax` в штатах Новой Англии, названия компаний которых начинаются с “Baker”.

Таких частных случаев может набраться десятки, и все они требуют расширения DSL в своем направлении. Но включение в язык специальных “примочек” для отдельных случаев может существенно его усложнить. В таких редко встречающихся случаях зачастую имеет смысл обрабатывать их с помощью языка, используя код на другом языке (*Foreign Code* (315)). При этом в DSL встраивается небольшой фрагмент кода на языке программирования общего назначения. Этот код не обрабатывается синтаксическим анализатором DSL; вместо этого он просто считывается в виде строки и передается в семантическую модель (*Semantic Model* (171)) для последующей обработки. В нашем случае это может вылиться в следующее правило (с участием Javascript в качестве языка программирования общего назначения).

```
scott handles floor_wax in MA RI CT when {/^Baker/.test(lead.name)};
```

Этот код не столь понятен, каким было бы расширение DSL, но если такие случаи станут распространеными, мы всегда сможем расширить этот язык позже.

В данной ситуации в качестве языка общего назначения я использовал Javascript. Применение в подобных случаях динамических языков полезно тем, что они позволяют читать и интерпретировать DSL-сценарий. Можно использовать вставки и со статическим языком, но тогда необходимо использовать генерацию кода и “сплетать” базовый код со сгенерированным. Эта методика хорошо знакома тем, кто использует генераторы синтаксических анализаторов (*Parser Generator* (277)), поскольку именно таким образом работает большинство упомянутых генераторов.

В этом примере использован код общего назначения, но ту же методику можно применять и с другим DSL. Такой подход позволит вам применять различные DSL для разных аспектов вашей проблемы, что соответствует философии использования нескольких небольших DSL, а не одного большого предметно-ориентированного языка.

К сожалению, с помощью современных технологий не очень легко совместно использовать несколько внешних DSL таким способом. Современные технологии синтаксического анализа не совсем подходят для смешивания различных языков с модульными грамматиками (см. раздел “Модулярная грамматика” на с. 345).

Одной из проблем при использовании чужого кода является то, что нужно токенизировать этот код не так, как код основного языка, так что необходимо использовать один из подходов альтернативной токенизации (*Alternative Tokenization* (325)).

Самый простой из подходов альтернативной токенизации состоит в размещении внешнего кода в некоторых ясно выделяемых разделителях, которые могут быть распознаны лексическим анализатором как одна строка (так я поступил выше, заключив код

Javascript в фигурные скобки). Такой подход позволяет легко отличить чужой код, но вносит в язык определенный шум.

Альтернативная токенизация пригодна не только для обработки внедряемого кода. В некоторых случаях, в зависимости от контекста синтаксического анализа, можно интерпретировать то, что обычно воспринимается как ключевое слово, как часть названия (как, например, в `state initial state`). Вам может помочь заключение в кавычки (`state "initial state"`), но другие реализации альтернативной токенизации, которые я буду рассматривать при обсуждении этого шаблона, могут снизить синтаксическую зашумленность.

5.5. XML DSL

В самом начале этой книги было сказано, что многие из файлов конфигурации в формате XML представляют собой эффективный DSL. С тех пор мои упоминания о XML DSL носили случайный характер — я просто ожидал, пока придет пора поговорить о внешних DSL.

Я не утверждаю, что все файлы конфигурации представляют собой DSL. В частности, я хотел бы провести различие между списками свойств и DSL. Список свойств — это простой список пар “ключ/значение”, возможно, разбитый по категориям. В нем практически нет синтаксической структуры — ничего из той таинственной природы языка, которая является ключевой для DSL. (Хотя я должен сказать, что для списка свойств XML слишком шумен, и для таких вещей я предпочитаю подход с применением INI-файлов.)

В основе многих файлов конфигурации лежат языки, и, таким образом, эти файлы представляют собой DSL. Когда они выполнены в XML, я рассматриваю их как внешний DSL. XML не является языком программирования; это синтаксическая структура без семантики. Таким образом, он обрабатывается путем чтения токенов, а не путем интерпретации для последующего выполнения. Обработка с помощью DOM, по сути, представляет собой построение дерева (Tree Construction (289)), с помощью SAX — приводит к встроенной трансляции (Embedded Translation (305)). Я рассматриваю XML в качестве носителя синтаксиса для DSL, во многом так же, как базовый язык является для внутреннего DSL носителем синтаксиса (а сам внутренний DSL является носителем семантики).

Проблема с применением XML в качестве носителя синтаксиса в том, что он содержит слишком много синтаксического шума — много угловых скобок, кавычек и косых черт. Любые вложенные элементы требуют открывающих и закрывающих дескрипторов. В результате на синтаксические структуры, в отличие от реального содержания, расходуется слишком много символов. Это осложняет понимание, что противоречит основному назначению DSL.

Несмотря на это имеется несколько аргументов в пользу XML. Первый — люди не должны писать XML самостоятельно; имеется масса пользовательских интерфейсов как для получения информации из XML, так и просто для его визуализации в удобной для восприятия человеком форме. Это разумный аргумент, хотя и уводящий нас с территории DSL; XML при этом становится механизмом сериализации, а не языком. Конкретная задача может в качестве альтернативы использованию DSL работать с формами и полями пользовательского интерфейса. Впрочем, о наличии пользовательских интерфейсов для XML больше говорят, чем их применяют.

Я часто слышу о наличии готовых синтаксических анализаторов XML, и что поэтому не нужно писать собственные анализаторы. Я думаю, что этот аргумент вытекает из недостаточного понимания того, что же такое синтаксический анализ. В этой книге я рассматриваю синтаксический анализ — весь маршрут от исходного текста до семантической модели (Semantic Model (171)). Анализатор XML проходит только часть этого пути — как правило, до DOM. Мы же все еще должны написать код для обхода DOM и

выполнения полезных действий. Это то, что могут делать генераторы синтаксических анализаторов (*Parser Generator* (277)); ANTLR может легко принять исходный текст и построить синтаксическое дерево, которое является эквивалентом DOM. Мой опыт показывает, что, как только вы станете более или менее знакомы с генераторами синтаксических анализаторов, их применение будет отнимать не больше вашего времени, чем инструменты синтаксического анализа XML. Другое дело, что программисты, как правило, лучше знают библиотеки для анализа XML, чем генераторы синтаксических анализаторов, но я считаю, что время, которое нужно затратить на обучение работе с генераторами синтаксических анализаторов, — вполне приемлемая цена.

При работе с внешними пользовательскими DSL раздражает несогласованность в таких вещах, как применение кавычек и управляющих символов. Любой, кому приходилось работать с конфигурационными файлами в Unix, легко меня поймет и оценит тот факт, что XML предусматривает единую надежно работающую схему.

Я только слегка касаюсь темы обработки ошибок и диагностики в этой книге, но это не должно быть причиной для игнорирования того факта, что XML-процессоры в этом, как правило, весьма сильны. Чтобы получить хорошую диагностику в типичном пользовательском языке, вам придется хорошо поработать; насколько хорошо — зависит от того, насколько хорош инструментарий, с которым вы работаете.

XML сопутствуют технологии, которые позволяют легко проверить корректность XML без выполнения, путем сравнения со схемой. Имеются различные форматы схем (DTD, XML Schema, Relax NG), и все они позволяют выполнить проверку соответствия XML различным правилам, а также поддерживают более интеллектуальные инструменты редактирования. (Я пишу эту книгу в XML с поддержкой схемы Relax NG, которую обеспечивает мой редактор Emacs.)

Помимо инструментов синтаксического анализа, которые генерируют деревья или события, вы можете использовать связывающие интерфейсы, которые могут легко транслировать XML-данные в поля объектов. Это менее полезно в случае DSL, так как структура семантической модели редко совпадает с таковой в DSL, чтобы позволить связывать элементы XML с семантической моделью. Вы можете использовать связывание со слоем трансляции, но вряд ли это что-то даст вам при обходе дерева XML.

Если вы используете генератор синтаксических анализаторов, то грамматика DSL может определить многие проверки, которые обеспечивают XML-схемы. Но некоторые инструменты могут воспользоваться преимуществами грамматики. Конечно, мы можем написать кое-что и самостоятельно, но плюсом XML является то, что такие средства для него уже существуют. Зачастую худший, но более распространенный подход в конечном итоге более полезен, чем просто высокие технологии.

Я согласен со всем этим, но по-прежнему считаю, что синтаксический шум XML слишком велик для DSL. Ключевым моментом DSL является удобочитаемость; инструментарий помогает при записи, но и удобочитаемость имеет важное значение. XML имеет свои достоинства — это действительно хорошая разметка текста, но как носитель синтаксиса DSL, на мой взгляд, он привносит слишком много шума.

Говоря о носителях синтаксиса, стоит упомянуть о получивших распространение в последнее время синтаксисах для кодирования структурированных данных, таких как JSON (www.json.org) и YAML (www.yaml.org). Многим (и мне в том числе) эти синтаксисы нравятся из-за меньшей, чем у XML, синтаксической зашумленности. Однако эти языки сильно ориентированы на структурирование данных, и в результате у них отсутствует гибкость, необходимая для по-настоящему свободного языка. DSL так же отличается от сериализации данных, как свободный API — от API командных запросов.

Глава 6

Выбор между внутренними и внешними DSL

Теперь, после ознакомления с деталями реализации внутренних и внешних DSL, вы можете лучше понять их сильные и слабые стороны. У вас достаточно знаний для того, чтобы не только выбрать один из двух методов, но и решить, подходит ли вам DSL вообще.

Одной из главных трудностей является отсутствие информации, на которой базируется ваш выбор. Лишь некоторые программисты работали с DSL, да и те, как правило, использовали только один или два метода и поэтому не могут реально сравнивать различные стили. Проблема осложняется еще и тем, что многие из методов, описанных в этой книге, не получили широкого распространения. Я надеюсь, что эта книга поможет вам упростить создание DSL, но пока что я не могу сказать, как она повлияет на решение использовать DSL и на выбор одного из видов DSL. Словом, мои размышления на эту тему более спекулятивны, чем хотелось бы.

6.1. Обучение

На первый взгляд, расходы на обучение говорят в пользу применения внутренних DSL. В конце концов, внутренние DSL — это на самом деле просто своеобразного вида API, и вы используете средства уже известного вам языка. В случае внешнего DSL вы должны изучить синтаксические анализаторы, грамматики и генераторы синтаксических анализаторов (*Parser Generator* (277)).

Какая-то доля правды в этом есть, но в целом картина более сложная. Безусловно, имеется несколько новых концепций, которые необходимо изучать для работы с синтаксически управляемой трансляцией (*Syntax-Directed Translation* (229)), а работа синтаксических анализаторов с грамматиками иногда может показаться шаманством. Все не так страшно, как многие себе представляют, но если вы не работали ранее с такого рода инструментами, то я бы все же рекомендовал вам выполнить ряд простейших примеров для знакомства с ними, прежде чем делать какие-либо оценки по поводу реальной работы.

К сожалению, изучение синтаксически управляемой трансляции усложняется плохой документацией большинства генераторов синтаксических анализаторов. Даже в тех случаях, когда имеется сколь-нибудь полная документация, она, как правило, написана для людей, работающих с языками общего назначения, а не DSL. Для многих инструментов документация имеется только в виде чьей-то диссертации. По сути, ничего не сделано для того, чтобы генераторы синтаксических анализаторов стали доступными тем, кто хочет их использовать для своих DSL, но не имеет соответствующего опыта работы.

Бывает мнение, что вместо указанных инструментов можно использовать управляемую разделителями трансляцию (*Delimiter-Directed Translation* (213)). В этом случае ин-

струменты гораздо более знакомы — разбиение строк, регулярные выражения, и никакой грамматики. Однако управляемая разделителями трансляция имеет пределы применения, причем не слишком широкие. Я считаю, что все же стоит не пожалеть времени и сил на изучение синтаксически управляемой трансляции, но нельзя забывать и об управляемой разделителями трансляции, в частности для регулярных языков.

Использование XML-носителей синтаксиса — это еще один способ избежать расходов на обучение синтаксически управляемой трансляции. В этом случае я просто уверен, что изучение синтаксически управляемой трансляции стоит затрат на него, поскольку в результате будут получаться существенно более удобочитаемые языки.

С другой стороны, внутренние DSL необязательно столь просты, как вы думаете. Хотя вы используете знакомый язык, вы делаете это весьма странным способом. Часто для достижения свободы внутренние DSL полагаются на разнообразные трюки базового языка. Поэтому, даже если вы очень хорошо знаете язык, вам, возможно, потребуется потратить некоторое время на изучение доступных в нем трюков. Шаблоны в этой книге, обращая внимание читателей на основные методы, должны помочь вам начать работу, но никаких трюков конкретных языков программирования вы здесь не найдете. Их обнаружение и использование — отдельная тема для каждого конкретного языка. Хорошая новость состоит в том, что вы можете обучаться в удобном для вас темпе, не спеша, изучая новые методы по ходу разработки DSL. Это контрастирует с синтаксически управляемой трансляцией, где вы должны изучить гораздо больше материала, чтобы просто начать работу.

Так что я не могу утверждать, что внутренние предметно-ориентированные языки проще в смысле обучения.

Рассматривая вопросы обучения, помните, что оно необходимо не только вам, но и тем, кто будет работать с вашим кодом. Использование внешнего DSL, пожалуй, окажется менее доступным для тех, кто не захочет прилагать усилия для его изучения.

6.2. Стоимость построения

Если вы используете DSL-технологии впервые, основные ваши расходы будут связаны с обучением. Как только вы освоите эту технологию, расходы на обучение исчезнут и на первый план выйдут другие расходы, связанные с созданием DSL.

Когда мы говорим о стоимости создания DSL, важно отделять стоимость создания модели от стоимости создания DSL, который представляет собой слой над моделью. Здесь я принимаю наличие модели как данность. Да, во многих случаях модель создается совместно с DSL, но при этом она имеет собственную мотивировку.

В случае внутреннего DSL дополнительные усилия включают создание слоя построителей выражений (Expression Builder (349)) над моделью. Построители выражений относительно просто написать, но основные усилия уходят не на то, чтобы заставить их работать, а на махинации с языком, чтобы получить нечто хорошо работающее. Стоимость построителя выражений отсутствует, если вы помещаете свободные методы непосредственно в модель, но это может привести к другим расходам, если программисты считут эти методы запутанными по сравнению с API командных запросов.

В случае внешнего DSL эквивалентная стоимость обусловливается построением синтаксического анализатора. Если вы работаете с применением синтаксически управляемой трансляции (Syntax-Directed Translation (229)), то довольно быстро напишете грамматику и код трансляции. Мне представляется, что стоимость разработки синтаксического анализатора аналогична стоимости построения слоя построителя выражений.

Если вы знакомы с синтаксически управляемой трансляцией, я думаю, для вас это будет даже легче, чем использовать XML-носитель синтаксиса, и уж точно легче, чем ис-

пользовать трансляцию, управляемую разделителями (Delimiter-Directed Translation (213)), кроме разве что случая очень простого языка.

Резюмируя, в настоящий момент я считаю, что, если вы знакомы с соответствующими методами, большой разницы в стоимости создания внутреннего или внешнего DSL нет.

6.3. Осведомленность программистов

Многие утверждают, что в случае внутреннего DSL работающие с ним программисты используют язык, с которым они уже знакомы, что облегчает работу по сравнению с применением внешнего DSL. В какой-то степени это верно, но я не думаю, что разница столь велика, как думает большинство. Стиль непривычного свободного интерфейса требует по меньшей мере привыкания, хотя и меньше усилий для изучения, как его следует строить. Внешний DSL также не сложен для изучения, так как по определению DSL должен быть достаточно прост. Применение в нем синтаксических соглашений вашего обычного языка программирования может помочь сделать его более доступным для вас.

Помимо синтаксических элементов, наибольшая разница часто заключается в имеющемся наборе инструментов. Если ваш базовый язык имеет интеллектуальную интегрированную среду разработки, то вы получите возможность работать с DSL с помощью знакомых инструментов. Возможно, для этого вам придется прибегнуть к более сложной методике наподобие класса таблицы символов (Class Symbol Table (461)), но, таким образом, вы сможете продолжать использовать сильные стороны интегрированной среды разработки в своей работе. В случае внешнего DSL вы вряд ли получите что-то, кроме базового уровня поддержки при редактировании. Скорее всего, вам придется работать с DSL в обычном текстовом редакторе. Поддерживать подсветку синтаксиса несложно, и большинство текстовых редакторов в этом отношении хорошо настраиваемы, но о таких возможностях, как автозаполнение с учетом типа, почти наверняка придется забыть.

6.4. Общение с экспертами предметной области

Внутренние DSL всегда связаны с синтаксисом базового языка. В результате во время работы почти всегда будут существовать определенные ограничения в возможности выразить те или иные вещи наряду с некоторым количеством синтаксических шумов. Это вряд ли существенно повлияет на конечных пользователей, но эксперты предметной области — это совсем другое дело. Степень ограничений и синтаксические шумы зависят также от базового языка; одни языки подходят для DSL лучше, чем другие.

Но даже лучшие внутренние DSL не предлагают такую же синтаксическую гибкость, как внешние DSL. Размер зоны комфорта зависит от конкретных специалистов в предметной области, но значение общения с ними настолько велико, что я склоняюсь к применению внешних DSL, если они в состоянии облегчить такое общение.

Если вы не слишком хорошо ориентируетесь в построении внешних DSL и не уверены в том, что внутренний DSL обеспечит общение с экспертами предметной области, можете сначала попробовать использовать внутренний DSL, а затем, если вы решите, что оно того стоит, переключиться на внешний DSL. Так как вы можете использовать одну и ту же семантическую модель (Semantic Model (171)) для обоих языков, дополнительные затраты на создание двух DSL на самом деле окажутся не такими уж и большими.

6.5. Смешивание в базовом языке

Внутренние DSL в действительности представляют собой не более чем соглашение использовать определенные свободные методы для выполнения некоторых действий. В них нет ничего, что препятствовало бы произвольно смешивать код в стиле DSL с регулярным императивным кодом. Эта тончайшая граница между DSL и основным языком имеет свойства, которые могут быть как полезны, так и наоборот в зависимости от того, что именно вы пытаетесь сделать.

Преимущество этой тонкой границы в том, что она позволяет свободно использовать базовый язык, если нет необходимой доступной конструкции внутреннего DSL. Так, если нужно выразить в DSL некоторую арифметику, создавая для этого специальные конструкции DSL нет смысла — просто используйте возможности базового языка. Для построения абстракции поверх DSL можно применять возможности абстракции базового языка.

Это преимущество особенно приятно, когда нужно разместить внутри DSL фрагменты императивного кода. Хорошим примером этого является применение DSL для описания процесса построения программного обеспечения. Языки сборки, использующие сеть зависимостей (*Dependency Network* (495)), такие как Make и Ant, применяются программистами уже очень давно. И Make, и Ant — внешние DSL, и оба они очень хороши в выражении сети зависимостей, необходимой для сборки. Однако содержание многих задач сборки требует более сложной логики, и часто зависимости сами нуждаются в слое абстракции поверх них. В результате Ant страдает от сползания в обобщенность, включая всевозможные императивные конструкции, которые не соответствуют его природе и синтаксису.

Здесь наблюдается контраст с внутренним DSL, таким как язык Rake, который представляет собой внутренний DSL Ruby для построения программного обеспечения. Возможность свободно смешивать сеть зависимостей с императивным кодом во вложенных замыканиях (*Nested Closure* (403)) упрощает описание сложных действий сборки. Объекты и методы Ruby, применяемые для создания абстракций поверх сети зависимостей, помогают описать высокоуровневые структуры сборки.

Смешивание внешнего DSL с кодом на базовом языке программирования невозмож но. Можно встраивать базовый код в сценарии DSL в качестве внешнего кода (*Foreign Code* (315)). Аналогично можно встраивать код DSL в код общего назначения в виде строк, как в повседневном программировании мы постоянно используем вставку таких вещей, как регулярные выражения и SQL. Но подобное смешивание неудобно. Инструменты, как правило, не понимают ваших действий и в результате работают достаточно неуклюже. Интеграция символов из двух сред — задача очень сложная, поэтому обеспечить, например, обращение к переменной базового кода внутри фрагмента DSL очень трудно. Если вы планируете смешивать базовый код и код DSL, то учите, что почти всегда решение заключается в применении внутреннего DSL.

6.6. Границы строгой выразительности

Способность свободно смешивать базовый код и код DSL не всегда положительна. В действительности это работает только тогда, когда пользователи DSL хорошо знают базовый язык; но это не так, когда ваш DSL-код читают эксперты предметной области. Вставки фрагментов на базовом языке программирования в DSL, как правило, приводят к повышению языкового барьера, который DSL призваны убирать.

Смешивание также бесполезно в случаях, когда код DSL должен быть создан отдельной группой программистов. Действительно, часто преимуществом DSL является то, что он ограничивает диапазон возможных выполняемых действий. Это ограничение может облегчить понимание языка и послужить защитой от ошибок. DSL со строгими границами может снизить количество необходимых тестирований и их видов. Правила ценообразования в DSL не в состоянии отправить произвольное сообщение на ваш сервер интеграции или изменить процесс обработки заказов. В случае языка общего назначения все это возможно, поэтому вам придется строго следить за переходом границ “нарушителями конвенции”. Ограничения внешнего DSL снижают количество вещей, за которыми вы должны следить. В большинстве случаев это хорошо, так как защищает от ошибок, а также помогает обеспечить безопасность.

6.7. Настройка времени выполнения

Одной из главных причин высокой популярности XML DSL является то, что они позволяют перенести изменение контекста выполнения кода с времени компиляции на время выполнения. Это важный фактор в ситуациях, когда вы используете компилируемый язык и хотите изменить поведение системы без перекомпиляции. Внешние DSL позволяют сделать это, поскольку вы можете легко анализировать их во время выполнения основной программы, транслируя в семантическую модель (*Semantic Model* (171)), а затем выполняя ее. (Конечно, если вы работаете на интерпретируемом языке, то в нем во время выполнения делается все, так что рассматриваемый здесь вопрос вообще не уместен.)

Один из подходов заключается в использовании интерпретируемых языков в сочетании с компилируемым. Вы можете написать внутренний DSL на интерпретируемом языке. В таком сценарии многие из распространенных преимуществ внутренних DSL могут быть ослаблены. Если только большая часть команды не знакома с динамическим языком, вы не получите никаких характерных для внутреннего DSL выгод от знакомства программистов с языком. Инструменты для динамических языков часто беднее, чем для статических. Вы не сможете легко смешивать динамический язык и конструкции статического языка, а применение только динамического языка означает, что вы не сможете установить прочные границы вокруг DSL. Это не значит, что вы не должны использовать внутренние DSL таким образом — есть много случаев, к которым эти потенциальные проблемы не имеют отношения. Но все это приводит к тому, что внешний DSL лучше связывать со статическим базовым языком.

6.8. Скатывание в обобщенность

Одним из наиболее успешных современных DSL является Ant. Это язык для процесса построения приложений для Java, который представляет собой внешний DSL с синтаксисом XML. В дискуссии о DSL Джеймс Дункан Дэвидсон (James Duncan Davidson), создатель Ant, спросил: “Как нам предотвратить бедствия, происходящие с Ant?”

Ant имеет ошеломляющий успех и одновременно является кошмаром. В свое время он заполнил огромный пробел в развитии Java, но с тех пор его успех заставил многие команды столкнуться с его недостатками. У Ant есть много проблем, и, пожалуй, самой серьезной является его синтаксис XML (который в то время я вместе со всеми считал хорошей идеей). Но реальной проблемой Ant является то, что со временем его возможности настолько расширились, что он больше не обладает ограниченностью выразительных возможностей, характерной для DSL.

Это обычная дорога в ад, вымощенная благими намерениями. Программисты, знакомые с Unix, часто приводят аналогичный пример Sendmail. Это происходит потому, что требований, предъявляемых к DSL, становится все больше и больше, а это приводит к появлению новых возможностей и большей сложности, и постепенно, капля за каплей, из DSL утекают вся его ясность и простота.

Эта опасность всегда существует для внешних DSL и, как и большинство вопросов в области проектирования, не имеет простого решения. Для того чтобы ее избегать нужны постоянное внимание и стремление не позволить языку стать слишком сложным. Существуют альтернативные решения расширению и усложнению языка. Одно из них заключается в разработке **других языков для более сложных случаев**. Вместо расширения одного языка можно ввести другие языки для частных и сложных случаев. Такой новый язык может быть слоем над старым базовым DSL, и его выход может быть кодом базового DSL. Это может оказаться полезным методом для построения абстракций на языке, в котором отсутствуют соответствующие возможности. При таком росте сложности хорошим выбором часто являются внутренние DSL, потому что они позволяют смешивать DSL и элементы общего назначения.

Так как внутренние DSL сливаются с базовым языком общего назначения, они также не застрахованы от проблем такого рода. Проблемы могут возникнуть, когда смешение с базовым языком становится настолько сильным, что теряется всякий смысл DSL.

6.9. Соединение DSL

Я еще раз напоминаю, что DSL должны быть малыми и очень ограниченными в своих возможностях. Ну и, чтобы была выполнена реальная работа, они должны быть интегрированы с одним или несколькими языками общего назначения. Можно также скомпоновать вместе предметно-ориентированные языки.

В случае внутреннего DSL композиция разных предметно-ориентированных языков выполняется так же просто, как и смешивание DSL с базовым языком. Можно также использовать возможности абстракции базового языка, чтобы упростить работу полученной комбинации.

В случае внешнего DSL ситуация сложнее. Для создания такой комбинации с помощью синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)) необходимо иметь возможность написать независимые грамматики для разных языков и скомпоновать их. Большинство генераторов синтаксических анализаторов (*Parser Generator* (277)) не поддерживают такие возможности (еще одно следствие ориентации на предметно-ориентированные языки общего назначения). В результате, для того чтобы скомпоновать несколько DSL, следует использовать шаблон *Foreign Code* (315), что является более неуклюжим решением, чем необходимо. (В настоящее время ведется работа по созданию средств для поддержки композиций, но пока что они находятся в довольно незрелом виде.)

6.10. Подведение итогов

Мое заключение заключается в том, что заключения нет... Я не вижу четких, глобальных преимуществ применения внутреннего или внешнего DSL. Я даже не уверен, что есть какие-то общие принципы, которых можно было бы посоветовать придерживаться. Я надеюсь, что дал вам достаточно информации для размышлений, чтобы помочь вам судить, что будет наилучшим образом соответствовать вашей конкретной ситуации.

Хочу, однако, подчеркнуть, что эксперименты в обоих направлениях не такие уж дорогостоящие, как может показаться. Если используется семантическая модель (*Semantic Model* (171)), относительно легко наслойте над ней несколько DSL, как внутренних, так и внешних. Это предоставит много возможностей для экспериментов по поиску оптимального для ваших условий подхода.

Подход, который считает полезным Гленн Вандерберг (*Glenn Vanderburg*), заключается в использовании внутренних DSL на ранней стадии, когда вы все еще пытаетесь понять, что вы хотите сделать. Так вы получите легкий доступ к возможностям базового языка и более цельную среду для работы. После того как вы разберетесь в ситуации и будете нуждаться в некоторых преимуществах внешнего DSL, вы сможете его построить. Как обычно, наличие семантической модели существенно облегчает этот процесс.

Еще один не упомянутый пока что вариант — применение языковых инструментальных средств. Мы поговорим об этом в главе 9, “Языковые инструментальные средства”, на с. 143.

Глава 7

Альтернативные вычислительные модели

Когда речь заходит о преимуществах использования DSL, то часто можно услышать, что они поддерживают более декларативный подход к программированию. Признаюсь, у меня проблемы со словом “декларативный”. По-моему, оно часто используется в слишком широком смысле. В общем случае, однако, оно означает “нечто, отличающееся от императивного”.

Основные языки программирования следуют императивной вычислительной модели. Императивная модель определяет вычисления как последовательность шагов: сделай это, сделай то, если красный, сделай другое... Условные конструкции и циклы изменяют эти шаги, а сами шаги могут быть сгруппированы в функции. Объектно-ориентированные языки добавляют объединение данных и процессов, а также полиморфизм, но основываются они по-прежнему на императивной модели.

Императивная модель постоянно критикуется, в частности со стороны академических кругов, но продолжает оставаться основной расчетной моделью с первых дней вычислений. Я думаю, это связано с тем, что ее легко понять: последовательности действий являются тем, чему достаточно просто следовать.

Говоря о простоте понимания, следует учитывать, что в действительности есть два вида понимания. Первое — это понимание назначения программы, т.е. чего мы пытаемся достичь с ее помощью. Вторая форма понимания связана с реализацией — как программа работает для достижения поставленной цели. Императивная модель программирования особенно хорошо подходит для второго вида понимания: вы можете прочитать код и увидеть, что он делает. Для получения более подробной информации можно пройти программу пошагово с отладчиком: последовательность инструкций исходного кода в точности соответствует тому, что происходит в отладчике.

Императивный подход не всегда так хорошо работает, когда речь идет о понимании предназначения. Если имеется в виду последовательность действий, то все в порядке; но часто это не лучший способ выразить свои намерения. В таких случаях часто стоит рассмотреть различные вычислительные модели.

Начну с простого примера. Часто возникают ситуации, когда нужно указать последствия различных комбинаций условий. Небольшой пример — подсчет очков для оценки страховки автомобиля, когда вы можете увидеть что-то напоминающее рис 7.1.

Есть мобильный телефон	Да	Да	Нет	Нет
Есть красный автомобиль	Да	Нет	Да	Нет
Очки	7	3	2	0

Рис. 7.1. Простая таблица решения по страховке автомобиля

Таблицы такого вида — распространенное средство представления человеку описанной проблемы. Превратив таблицу в императивный код на языке программирования C#, мы получим что-то наподобие следующего.

```
public static int CalcPoints(Application a) {
    if (a.HasCellPhone && a.HasRedCar) return 7;
    if (a.HasCellPhone && !a.HasRedCar) return 3;
    if (!a.HasCellPhone && a.HasRedCar) return 2;
    if (!a.HasCellPhone && !a.HasRedCar) return 0;
    throw new ArgumentException("unreachable");
}
```

Мой обычный стиль написания логических условий более лаконичен.

```
public static int CalcPoints2(Application a) {
    if (a.HasCellPhone)
        return (a.HasRedCar) ? 7 : 3;
    else
        return (a.HasRedCar) ? 2 : 0;
}
```

В данном случае я все же предпочел бы первый, более длинный, способ написания, потому что это в большей степени соответствует цели эксперта в предметной области. Табличная природа его мышления ближе к макету этого исходного текста.

Это сходство между таблицей и кодом отнюдь не означает, что они представляют собой одно и то же. Императивная модель заставляет выполняться различные инструкции `if` в определенном порядке, который не вытекает из таблицы решений. Это означает, что в свое представление таблицы я добавляю не имеющий отношения к ней самой артефакт реализации. Для данной таблицы это не так уж и важно, но может иметь большое значение для других альтернативных вычислительных моделей.

Потенциально более серьезным недостатком императивного представления является то, что он удаляет некоторые полезные возможности. Одной из них в таблице решений является то, что вы можете проверить ее, чтобы убедиться, что не пропустили какую-то из перестановок или случайно не продублировали одну из них.

Альтернативой использованию императивного кода является создание абстракции таблицы решений и ее настройка для данного конкретного случая. В этой ситуации я мог бы представить таблицу примерно так.

```
var table = new DecisionTable<Application, int>();
table.AddCondition((application) => application.HasCellPhone);
table.AddCondition((application) => application.HasRedCar);
table.AddColumn(true, true, 7);
table.AddColumn(true, false, 3);
table.AddColumn(false, true, 2);
table.AddColumn(false, false, 0);
```

Теперь у меня есть более точное представление исходной таблицы решений. Я больше не указываю порядок вычисления условий в императивном коде (что, кстати, может быть

удобно для использования возможностей параллельных вычислений). Что более важно, для таблицы решений можно проверить корректность набора условий и сообщить, не пропустил ли я что-нибудь в своей конфигурации. В качестве премии контекст выполнения переносится с времени компиляции на время выполнения, и, таким образом, правила могут быть изменены без перекомпиляции.

Я называю этот стиль представления адаптивной моделью (*Adaptive Model* (478)). Термин “адаптивная объектная модель” уже использовался какое-то время для описания соответствующих действий в виде объектно-ориентированной модели — ознакомьтесь с работой Джо Йодера (Joe Yoder) и Ральфа Джонсона (Ralph Johnson) [26]. Но вам не нужно обязательно использовать объекты для того, чтобы сделать что-то вроде описанного выше. Не менее распространено хранение структур данных, определяющих правила поведения, в базах данных. Большинство объектно-ориентированных моделей имеют объекты, которые содержат как поведение, так и данные, но в случае адаптивной модели главным является то, что поведение во многом определяется экземплярами модели и тем, как они связаны друг с другом. Вы не можете понять, какое поведение следует ожидать, не глядя на конфигурации экземпляров.

Для использования адаптивной модели не нужен DSL. На самом деле DSL и адаптивные модели представляют собой отдельные понятия, которые могут использоваться независимо одно от другого. Несмотря на это, мне кажется, что для вас должно быть очевидно, что адаптивные модели и DSL идут рука об руку, как вино и сыр. В этой книге я постоянно говорил только о том, что при выполнении синтаксического анализа DSL строится семантическая модель (*Semantic Model* (171)). Часто эта семантическая модель является адаптивной моделью, которая предоставляет альтернативную вычислительную модель для части программной системы.

Большим минусом использования адаптивной модели является то, что определяемое ею поведение неявное; вы не можете просто посмотреть на код, чтобы увидеть, что происходит. Кроме того, в то время как намерения обычно проще для понимания, о реализации этого не скажешь. Это становится важным, когда что-то идет не так и необходима отладка. Часто найти недостатки при использовании адаптивной модели намного труднее. Я нередко слышу жалобы программистов о том, что они не могут найти программу или понять, как она работает. В результате адаптивные модели имеют репутацию сложных в поддержке. Я неоднократно слышал о программах, тративших месяцы на выяснение того, как работает некоторая адаптивная модель. Если им это удастся, адаптивная модель сможет сделать их работу очень продуктивной. Но до тех пор, пока они не разберутся в реализации модели (чего многие программисты вообще никогда не делают), их жизнь будет похожа на кошмар.

Это понимание реализации адаптивных моделей является реальной проблемой, справедливо удерживающей многих людей от их использования, несмотря на реальные выгоды, которые можно получить, привыкнув к ним и тому, как они работают. Одним из преимуществ, которые я вижу в применении DSL, является то, что они облегчают понимание и программирование адаптивной модели. DSL по крайней мере позволяет увидеть программы. Они не снимают вопросы понимания работы адаптивной модели, но, позволяя более четко увидеть конкретные конфигурации, могут обеспечить вам значительную поддержку.

Альтернативные вычислительные модели также являются одной из веских причин для использования DSL, и именно поэтому я затратил на них достаточно большую часть этой книги. Если ваша проблема может быть легко выражена с помощью императивного кода, то обычный язык программирования работает просто отлично. Ключевые преимущества DSL — большая производительность и общение с экспертами предметной области —

активно работают при использовании альтернативных моделей вычислений. Эксперты предметной области часто думают о своих проблемах неимперативным способом, например с применением таблиц решений. Адаптивная модель позволяет более непосредственно отразить их способ мышления в программе, а DSL упрощает передачу такого представления экспертам (да и вам тоже).

7.1. Несколько альтернативных моделей

Существует много вычислительных моделей, но я не собираюсь делать из этой книги всеобъемлющий обзор. Все, что я могу сделать, — это предоставить несколько наиболее распространенных образцов. Это может помочь вам в общих ситуациях, но, я надеюсь, они пробудят ваше воображение и вы сможете сами придумать вычислительные модели для своей предметной области.

7.1.1. Таблицы решений

Поскольку я уже упоминал таблицы решений (Decision Table (485)), я начну именно с них. Это довольно простая форма альтернативной вычислительной модели, но она хорошо подходит для DSL. На рис. 7.2 показан относительно простой пример.

Привилегированный клиент	x	x	Да	Да	Нет	Нет
Приоритетный заказ	Да	Нет	Да	Нет	Да	Нет
Международный заказ	Да	Да	Нет	Нет	Нет	Нет
Оплата	150	100	70	50	80	60
Оповещение	Да	Да	Да	Нет	Нет	Нет

Рис. 7.2. Таблица решений для обработки заказов

Таблица состоит из нескольких строк условий, за которыми следует некоторое количество строк следствий. Семантика очень проста: в данном случае вы принимаете заказ и сравните его с условиями. С данными заказа должен совпадать один столбец условий, после чего применяются следствия этого столбца. Так, если у нас привилегированный клиент с приоритетным отечественным заказом, то его оплата — 70 долларов и об этом заказе следует оповестить представителя для его обработки.

В данном случае каждое условие является логическим, но более сложные таблицы решений могут иметь и другие формы условий, например такие, как числовые диапазоны.

Таблицы решений особенно понятны для не программистов, поэтому они хорошо подходят для общения с экспертами предметной области. Табличная природа делает естественным их редактирование в электронных таблицах, так что это частный случай DSL, когда непосредственное редактирование экспертом предметной области более вероятно, чем не вероятно.

7.1.2. Система правил вывода

Общим понятием моделирования логики путем разделения на правила, каждое из которых имеет условие и последующие действия, является система правил вывода (Production

Rule System (503)). Каждое правило может быть указано индивидуально в стиле, напоминающем набор инструкций *if -then* императивного кода.

```
if
  passenger.frequentFlier
then
  passenger.priorityHandling = true;

if
  mileage > 25000
then
  passenger.frequentFlier = true;
```

Используя систему правил вывода, мы указываем эти правила в терминах их условий и действий, но оставляем их выполнение и связывание базовой системе. В только что показанном примере имеется связь между правилами, заключающаяся в том, что если истинно второе правило (на сленге — “сработало”), то это может повлиять на то, должно ли быть выполнено первое правило.

Характеристика, когда срабатывание одних правил влияет на срабатывание других правил, называется **выводом** (chaining) и является важной особенностью системы правил вывода. Связывание позволяет писать правила индивидуально, не думая об их более широких следствиях, а затем предоставляет системе возможность самой устраниить эти последствия.

Это опасное преимущество. Система правил вывода опирается на большое количество неявной логики, которая зачастую может делать то, что не было предусмотрено. Иногда такое непредвиденное поведение может оказаться полезным, а иногда — очень вредным и приводить к неправильным результатам. Эти ошибки часто возникают из-за того, что при написании правил не было принято во внимание их взаимодействие.

Проблемы, связанные с неявным поведением, — общая проблема альтернативных вычислительных моделей. Мы в основном используем императивные модели, но все равно делаем много ошибок. С альтернативными моделями мы получаем еще больше проблем, потому что часто не в состоянии, просто глядя на код, сразу сообразить, что произойдет. Хорошо это или плохо, но куча правил, встроенных в базу правил, часто приводит к удивительным результатам. Одним из следствий этого, применимым к большинству случаев реализации альтернативных моделей вычислений, является то, что очень важно иметь механизм отслеживания, который позволил бы увидеть, что именно происходит при выполнении модели. В случае системы правил вывода это означает возможность записи срабатывающих правил и быстрое получение этих записей при необходимости. Таким образом, озадаченный пользователь или программист может увидеть всю цепочку правил, которые привели к неожиданному выводу.

Системы правил вывода существуют уже долгое время, и имеется много реализующих их продуктов, которые предоставляют также сложные инструменты для получения и выполнения правил. Несмотря на это все еще может быть полезно написать небольшую систему правил вывода в пределах собственного кода. Как и в любом случае, когда вы создаете собственные альтернативные модели вычислений, обычно вполне можно создать нечто относительно простое, работая в небольшом масштабе определенной предметной области.

Хотя связывание в цепочки, безусловно, является важной частью системы правил вывода, на самом деле оно не является обязательным. Иногда полезно написать систему правил вывода без такого связывания. Хорошим примером может служить набор правил проверки. При проверке часто рассматривается набор условий, для которых действие представляет собой генерацию ошибки. Хотя в этом случае связывание в цепочку не тре-

буется, рассмотрение поведения как набора независимых правил оказывается по-прежнему полезным.

Вы можете сказать, что таблица решений (Decision Table (485)) представляет собой форму системы правил вывода, каждый столбец которой соответствует одному правилу. Хотя это и правда, я думаю, что упускается из виду главное. При наличии системы правил вывода в каждый момент времени вы сосредоточиваетесь на поведении одного правила, в то время как в случае таблицы решений вы концентрируетесь на всей таблице в целом. Это смещение при рассмотрении является неотъемлемой частью двух указанных моделей, что делает их совершенно различными мысленными инструментами.

7.1.3. Конечный автомат

Эта книга началась с рассмотрения еще одной популярной альтернативной вычислительной модели — конечного автомата (State Machine (517)). Конечные автоматы моделируют поведение объекта путем его разделения на множество состояний; поведение вызывается событиями так, что в зависимости от состояния объекта, в котором он находится, каждое событие приводит к переходу в другое состояние.

На рис. 7.3 показано, что можно отменить заказ в состояниях накопления и оплаты, из каждого из которых имеется переход в отмененное состояние.

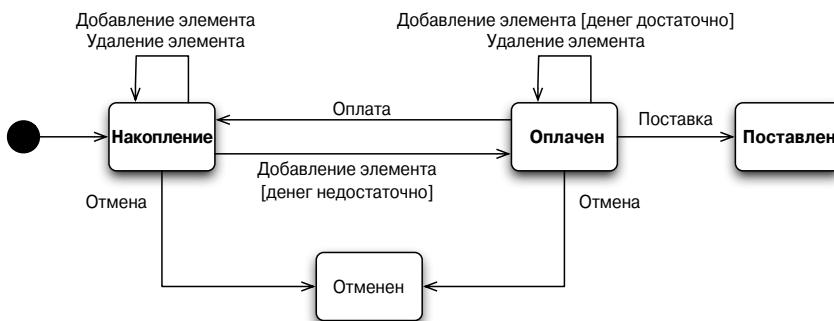


Рис. 7.3. UML-диаграмма конечного автомата заказа

Конечный автомат имеет базовые элементы — состояния, события и переходы. Но существует много вариаций, построенных на основе этой базовой структуры. Вариации, в частности, проявляются в том, как конечный автомат инициирует действия. Конечный автомат представляет собой распространенный выбор, поскольку многие системы могут рассматриваться как реакция на события путем прохода через ряд состояний.

7.1.4. Сеть зависимостей

Одной из наиболее распространенных альтернативных моделей, используемых в повседневной работе разработчиков программного обеспечения, является сеть зависимостей (Dependency Network (495)). Она знакома практически любому программисту, так как именно эта модель лежит в основе таких инструментов, как Make, Ant и производных от них. В этой модели мы рассматриваем задачи, которые необходимо выполнить, и предпосылки выполнения каждой задачи. Например, задача запуска тестов может в качестве предпосылок иметь компиляцию и загрузку данных, а они, в свою очередь, в качестве предварительного условия могут иметь генерацию одного и того же кода (рис. 7.4). При на-

личии указанных зависимостей можно запросить выполнение теста, после чего система определяет, какие другие задачи необходимо выполнить и в каком порядке. Кроме того, хотя генерация кода встречается в списке дважды (будучи дважды указанной в качестве предварительного условия), система знает, что запустить ее нужно только один раз.

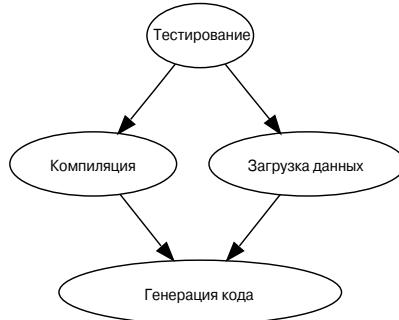


Рис. 7.4. Возможная сеть зависимостей для построения программного обеспечения

Таким образом, сеть зависимостей оказывается хорошим выбором, когда у вас имеются вычислительно дорогие задачи с зависимостями между ними.

7.1.5. Выбор модели

Мне трудно сформулировать какие-то советы о том, когда следует выбирать конкретную вычислительную модель. Все они сводятся к одному — какая из них лучше всего соответствует вашей точке зрения на проблему. Наилучший способ определить, подходит ли некоторая модель, — испытать ее на практике. Сначала просто попробуйте описать ее поведение на бумаге с помощью простого текста и диаграмм. Если модель проходит этот простейший настольный тест, значит, она стоит того, чтобы ее построить и испытать в действии. Мне кажется, что главное при этом — добиться корректной работы семантической модели (*Semantic Model (171)*), но простой DSL может вам в этом помочь. Тем не менее, в первую очередь, я бы прилагал больше усилий для настройки модели, а уже затем озадачивался получением очень легко читаемого DSL. После того как у вас будет в наличии приемлемая семантическая модель, вам будет относительно легко экспериментировать с различными DSL для ее управления.

Есть еще очень много альтернативных вычислительных моделей, кроме описанных. Если бы у меня было больше времени на работу над этой книгой, я потратил бы его на описание нескольких из них. Мне кажется, если кто-то напишет книгу о вычислительных моделях, она будет иметь большой успех.

Глава 8

Генерация кода

До сих пор, рассматривая реализации DSL, мы говорили о синтаксическом анализе исходного текста, как правило, с целью наполнения семантической модели ([Semantic Model \(171\)](#)) и обеспечения интересующего ее поведения. Во многих случаях, как только появляется возможность наполнить семантическую модель, работу можно считать завершенной — мы можем просто выполнить семантическую модель, чтобы получить искомое поведение.

Хотя непосредственное выполнение семантической модели, как правило, — простейший способ, он далеко не всегда доступен. Может оказаться, что определяемая исходным текстом DSL логика должна выполняться в широком диапазоне сред, там, где трудно или невозможно построить семантическую модель или синтаксический анализатор. В таких ситуациях можно прибегнуть к генерации кода, чтобы с его помощью запустить указанное в исходном тексте DSL поведение практически в любой среде.

При использовании генерации кода существуют две среды: одну я буду называть процессором DSL, а вторая — целевая среда. Процессор DSL — это среда, в которой работают синтаксический анализатор, семантическая модель и генератор кода. Это должна быть среда, комфортная для разработки данных инструментов. Целевая среда представляет собой ваш сгенерированный код и его окружение. Генерация кода позволяет отделить целевую среду от вашего процессора DSL, потому что вы не можете корректно построить процессор DSL в целевой среде.

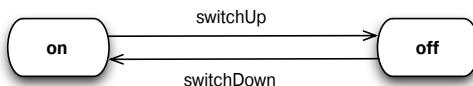
Существуют различные формы целевых сред. Одной из них является встраиваемая система, в которой просто нет ресурсов для запуска процессора DSL. Другой — когда целевая среда должна быть языком, который не подходит для обработки DSL. По иронии судьбы целевая среда сама по себе может оказаться DSL. Поскольку DSL имеют ограниченные выразительные возможности, они обычно не обеспечивают средства абстракции, необходимые для более сложной системы. Даже если вам удастся расширить DSL, придав ему средства абстракции, это может быть сделано за счет существенного усложнения DSL, вероятно, достаточного для того, чтобы превратить его в язык общего назначения. Так что, возможно, лучше работать с абстракциями в другой среде и генерировать код для целевой DSL. Хорошим примером этого является определение условий запроса в DSL и генерация SQL. Это делается для того, чтобы наши запросы эффективно работали с базой данных, но использование SQL при этом — не лучший способ представления этих запросов.

Ограничения целевой среды — не единственная причина генерации кода. Другой причиной может быть недостаточное знание целевой среды. Может оказаться проще определить поведение на более привычном языке, а затем генерировать менее знакомый. Еще одна причина для генерации кода — лучшее обеспечение статического контроля. Мы могли бы охарактеризовать интерфейс некоторой системы с помощью DSL, но остальная часть системы общается с ним посредством C#. В этом случае можно сгенериро-

вать API C#, чтобы получить возможность проверок времени компиляции и поддержку интегрированной среды разработки. В случае изменений в определении интерфейса можно регенерировать код C#, и компилятор поможет вам выявить возможные проблемы в коде.

8.1. Выбор объекта генерации

Одно из первых решений, которые нужно принять при генерации кода, — какой именно код необходимо генерировать. С моей точки зрения, есть два типа генерации кода, которые можно использовать: генерация, осведомленная о модели (*Model-Aware Generation* (545)), и генерация, игнорирующая модель (*Model Ignorant Generation* (557)). Разница между ними заключается в наличии явного представления семантической модели (*Semantic Model* (171)) в целевой среде.



Rис. 8.1. Очень простой конечный автомат

В качестве примера давайте рассмотрим конечный автомат (*State Machine* (517)). Для его реализации обычно применяются две классические альтернативы: вложенные условные конструкции и таблицы состояния. Если взять очень простую модель состояний, такую, как на рис. 8.1, вложенные условные конструкции будут выглядеть следующим образом.

```

public void handle(Event event) {
    switch (currentState) {
        case ON: switch (event) {
            case DOWN:
                currentState = OFF;
            }
        case OFF: switch (event) {
            case UP : currentState = ON;
        }
    }
}
  
```

У нас есть две условные конструкции, вложенные одна в другую. Во внешней выясняется текущее состояние конечного автомата, а во внутренней проверяется полученное событие. Это генерация, игнорирующая модель, поскольку логика конечного автомата встроена в поток управления языка; здесь нет явного представления семантической модели.

В случае генерации, осведомленной о модели, мы размещаем некоторое представление семантической модели в генерируемом коде. Оно не обязано быть в точности таким же, как и в процессоре DSL, но оно является некоторой формой представления данных. В этом случае конечный автомат более сложный.

```

class ModelMachine...
private State currentState;
private Map<State, Map<Event, State>> states =
    new HashMap<State, Map<Event, State>>();

public ModelMachine(State currentState) {
    this.currentState = currentState;
}
void defineTransition(State source, Event trigger,
  
```

```

        State target) {
    if (! states.containsKey(source))
        states.put(source, new HashMap<Event, State>());
    states.get(source).put(trigger, target);
}
public void handle(Event event) {
    Map<Event, State> currentTransitions =
        states.get(currentState);
    if (null == currentTransitions) return;
    State target = currentTransitions.get(event);
    if (null != target) currentState = target;
}
}

```

Здесь переходы хранятся в виде вложенных отображений. Внешнее отображение представляет собой отображение состояний, ключ которого — название состояния, а значение — второе отображение. Это внутреннее отображение использует имя события в качестве ключа и целевое состояние — в качестве значения. Это сырья модель состояний — у меня может не быть явных состояний, переходов и даже классов, — но структура данных определяет поведение данного конечного автомата. В результате этот управляемый данными код носит совершенно общий характер и требует для работы настройки с помощью некоторого другого кода.

```

modelMachine = new ModelMachine(OFF);
modelMachine.defineTransition(OFF, UP, ON);
modelMachine.defineTransition(ON, DOWN, OFF);

```

После размещения представления семантической модели в сгенерированном коде последний разделяется на обобщенный код платформы и конкретный код настройки, как я и говорил выше. Генерация, осведомленная о модели, сохраняет разделение на общий и конкретный коды, в то время как генерация, игнорирующая модель, соединяет эти части, представляя семантическую модель в потоке управления.

В результате, если я использую генерацию, осведомленную о модели, то все, что мне нужно сгенерировать, — конкретный код настройки. Я могу полностью построить базовый конечный автомат в целевой среде и протестировать его там. В случае генерации, игнорирующей модель, я должен генерировать гораздо больше кода. Я могу переместить некоторый код в библиотеку функций, которые не нужно генерировать, но большая часть критического поведения должна быть сгенерирована.

Таким образом, намного легче генерировать код, используя генерацию, осведомленную о модели. Сгенерированный код, как правило, очень прост. Необходимо создать общий раздел, но, поскольку вы можете запускать и тестировать его независимо от системы генерации кода, обычно это намного проще сделать.

Поэтому я склоняюсь к как можно более частому использованию генерации, осведомленной о модели. Однако зачастую это не представляется возможным. Нередко генерация кода используется потому, что целевой язык не может представлять модель так же легко, как и данные. И даже если ему это удается, могут возникнуть ограничения обработки. Встроенные системы часто используют генерацию кода, игнорирующую модель, потому что накладные расходы на обработку кода, сгенерированного с учетом модели, слишком велики.

Есть и другой фактор, который следует по возможности не упускать из виду при использовании генерации, осведомленной о модели. Для того чтобы изменить конкретное поведение системы, можно заменить только артефакт, соответствующий конфигурации кода. Представим, что генерируется код на языке программирования C; код настройки можно поместить в библиотеку, отличную от библиотеки обобщенного кода, что позволит изменять конкретное поведение без замены всей системы (хотя потребуется некоторо-

рый механизм связывания времени выполнения для того, чтобы справиться с поставленной задачей).

Можно пойти еще дальше и сгенерировать представление, которое полностью считывается во время выполнения. Можно сгенерировать простую текстовую таблицу, как, например, показано ниже.

```
off switchUp on
on switchDown off
```

Это позволило бы изменить конкретное поведение системы во время выполнения ценой введения в систему кода для загрузки файла данных при запуске.

Сейчас вы, вероятно, думаете, что я только что сгенерировал другой DSL, который анализируется в целевой среде. Но я так не считаю. На мой взгляд, маленькая приведенная выше таблица в действительности предметно-ориентированным языком не является, поскольку не предназначена для манипуляций, осуществляемых человеком. Текстовый формат делает ее удобочитаемой, но это просто более полезная возможность для отладки. В первую очередь, таблица должна легко анализироваться, чтобы ее можно было быстро загружать в целевой системе. При разработке такого формата визуальное восприятие человеком отодвигается на второй план, а на первый выходит простота анализа. В случае же DSL приоритетной задачей является визуальное восприятие кода человеком.

8.2. Как генерировать код

После выбора типа генерируемого кода следует принять решение о стиле процесса генерации. Существует два основных стиля генерации текстового выхода, которым вы можете следовать: генерация с помощью преобразователя (*Transformer Generation* (523)) и шаблонная генерация (*Templated Generation* (529)). В случае генерации с помощью преобразователя вы пишете код, который считывает семантическую модель (*Semantic Model* (171)) и генерирует инструкции исходного текста на целевом языке программирования. Так, для примера с состояниями вы можете захватить события и генерировать выходной код для объявления каждого события; то же относится и к командам, и к состояниям. Поскольку состояния содержат переходы, генерация для каждого состояния должна также включать навигацию к переходам и генерацию кода для каждого из них.

В случае шаблонной генерации вы начинаете с написания образца выходного файла. В этом выходном файле, там, где должен быть код, специфичный для конкретного конечного автомата, вы размещаете специальные маркеры, которые позволяют вам обращаться к семантической модели для генерации соответствующего кода. Если вы работали с шаблонными веб-страницами с таким инструментарием, как ASP, JSP и другие, вы должны знать этот механизм. При обработке шаблонов ссылки в нем заменяются сгенерированным кодом. В случае шаблонной генерации вами управляет структура вашего вывода, в отличие от генерации преобразователя, когда вами может управлять как ввод, так и вывод либо оба.

Оба подхода к генерации кода вполне работоспособны, и при выборе между ними обычно стоит поэкспериментировать с каждым и посмотреть, какой оказывается лучшим для вас. Я считаю, что шаблонная генерация лучше всего работает, когда в выводе есть много статического кода и только несколько динамических фрагментов, в частности поскольку я могу посмотреть на файл шаблона и понять, что же именно генерируется. Как следствие этого, я думаю, шаблонная генерация чаще используется при генерации, игнорирующей модель (*Model Ignorant Generation* (557)). В противном случае (на самом деле — по большей части) я предпочитаю генерацию преобразователя.

Я рассматривал эти подходы как противоположные, но это не означает, что вы не можете их смешивать. На самом деле это происходит постоянно. Если вы используете генерацию преобразователя, то, вероятно, для записи небольших фрагментов кода используете инструкции в строковом формате, — а это и есть миниатюрные случаи шаблонной генерации. Несмотря на это я считаю, что следует иметь четкое представление об общей стратегии и осознанно переходить к использованию другой стратегии в фрагментах. Как и в большинстве ситуаций в программировании, как только вы перестаете думать над тем, что вы делаете, вы тут же начинаете творить полный беспорядок, который будет невозможно поддерживать.

Одной из самых серьезных проблем в использовании шаблонной генерации является то, что базовый код, используемый для генерации переменного вывода, может начать подавлять статический шаблонный код. Если вы приступаете к генерации Java для генерации C, то шаблон должен быть в основном на C; следует свести к минимуму любые появления Java в шаблоне. Я считаю, что здесь приобретает особое значение шаблон *Embedment Helper* (537). Все сложности выяснения, как генерировать переменные элементы шаблона, должны быть скрыты в классе, который вызывается с помощью простых вызовов методов в шаблоне. Таким образом, вы обеспечите минимум Java в своем C.

Это не только сохраняет ваш шаблон ясным, но и, как правило, упрощает работу с кодом генерации. Встроенный помощник (*Embedment Helper* (537)) может представлять собой обычный класс, редактируемый с помощью инструментов, предназначенных для работы с Java. Это особенно важно в случае интеллектуальных интегрированных сред разработки. Если вы вставляете много кода Java в файл на языке программирования C, среда разработки, как правило, не в состоянии вам помочь. Вы можете даже не получить подсветку синтаксиса. Все выноски в шаблоне должны представлять собой отдельные вызовы методов; все остальное должно находиться внутри встроенного помощника.

Хорошим примером, в котором это весьма важно, являются файлы грамматики для синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)). Я часто сталкиваюсь с файлами грамматик, переполненными кодом действий, по сути — с блоками внешнего кода (*Foreign Code* (315)). Эти блоки вплетены в генерируемый синтаксический анализатор, но их размер полностью скрывает структуру грамматики; встроенный помощник в этом случае оказывает неоценимую помощь, позволяя обойтись малым кодом действий.

8.3. Смешивание сгенерированного и рукописного кодов

Иногда удается сгенерировать весь код, который необходимо выполнить в целевой среде, но чаще всего приходится смешивать сгенерированный и рукописный коды.

В этом случае рекомендуется придерживаться следующих общих правил.

- Не изменяйте сгенерированный код
- Храните сгенерированный код четко отделенным от рукописного кода

Главное в использовании генерации кода из DSL — DSL становится официальным источником поведения. Любой генерируемый код — просто артефакт. Редактируя результат генерации кода вручную, вы теряете вносимые изменения при регенерации. Это требует дополнительной работы по генерации, что не только плохо само по себе, но и приводит к нежеланию в случае необходимости изменять DSL, сводя тем самым “к нулю” весь смысл наличия DSL. (Иногда при создании рукописного кода полезно начать со сгенерированного шаблона, но это необычная ситуация в случае DSL.)

Поэтому сгенерированный код никогда не следует изменять. (Единственным исключением является вставка операторов трассировки для отладки.) Его имеет смысл хранить отдельно от рукописного кода. Я предпочитаю четко разделять файлы на полностью сгенерированные и полностью рукописные. Я не вношу сгенерированный код в репозиторий исходного кода, так как он может быть регенерирован во время сборки, и предпочитаю хранить сгенерированный код в отдельной ветви дерева исходных кодов.

В процедурной системе, в которой код организован в виде файлов функций, этого достичь довольно легко. Однако объектно-ориентированный код с классами, в которых смешаны структуры данных и поведение, зачастую осложняет такое разделение. Обычно имеется один логический класс, но одни его части должны быть сгенерированы, а другие — быть рукописными.

Часто самый простой способ справиться с этим — использовать несколько файлов для своего класса; тогда можно будет разделить генерируемый и рукописный коды по своему усмотрению. Однако не все среды программирования позволяют это сделать, например не позволяет Java, C# позволяет только в последних версиях — под названием “частичные классы”. Если вы работаете с Java, вы не сможете просто разделить класс на файлы.

Одним из популярных вариантов действий в такой ситуации является пометка отдельных участков класса как сгенерированных или рукописных. Я всегда считал этот механизм неуклюжим, потому что часто он приводит к ошибкам, так как люди могут изменить сгенерированный код. Это также обычно означает, что невозможно избежать внесения сгенерированного кода в систему управления версиями, что только путает ее.

Хорошим решением может служить использование шаблона *Generation Gap* (561)², позволяющего разделить сгенерированный и рукописный коды с помощью наследования. В базовой форме шаблона вы генерируете суперкласс и вручную пишете подкласс, который может дополнить и переопределить сгенерированное поведение. Это сохраняет разделение генерируемого и рукописного кодов на уровне файлов, обеспечивая большую гибкость сочетания обоих стилей в одном классе. Недостатком является необходимость ослабления правил видимости. Методы, которые в противном случае могли бы быть закрытыми, должны стать защищенными, чтобы их можно было перекрывать и вызывать из подкласса. Я нахожу такое ослабление небольшой платой за отдельное хранение генерируемого и рукописного кодов.

Трудность раздельного хранения генерируемого и рукописного кодов, похоже, зависит от структуры вызовов между сгенерированным и рукописным кодами. Простой поток управления, как при генерации, игнорирующей модель (*Model Ignorant Generation* (557)), когда сгенерированный код вызывает рукописный код в одностороннем потоке, упрощает разделение двух артефактов. Так, если у вас имеются сложности с раздельным хранением рукописного и сгенерированного кодов, возможно, стоит подумать о том, как упростить поток управления.

8.4. Генерация удобочитаемого кода

Когда речь идет о генерации кода, время от времени возникает вопрос, насколько сгенерированный код должен быть удобочитаемым и хорошо структурированным. Здесь обычно сталкиваются два мнения — мнение о том, что сгенерированный код должен быть максимально четким и удобочитаемым, как рукописный код, и мнение, что все это

² По сути, это игра слов, поскольку *generation* переводится и как *генерация*, и как *поколение*, а сочетание *generation gap* в английском языке означает конфликт поколений, проблему отцов и детей. — Примеч. пер.

не имеет значения для сгенерированного кода, так как он никогда не должен изменяться вручную.

В этой дискуссии я склоняюсь к тем, кто считает, что сгенерированный код должен быть хорошо структурированным и удобочитаемым. Хотя вы не должны трогать сгенерированный код руками, вы можете захотеть понять, как он работает. Или что-то пойдет не так и потребует отладки (а отлаживать четкий и хорошо структурированный код гораздо легче).

Поэтому я предпочитаю, чтобы сгенерированный код был почти так же хорош, как код, который я бы хотел писать вручную, — с понятными именами переменных, хорошей структурой и прочими обычно используемыми при написании кода мелочами.

Есть и исключения. Возможно, основным из них является то, что я менее заинтересован втрате времени для получения правильной структуры сгенерированного кода. При генерации я меньше беспокоюсь по поводу дублирования; конечно, я не хочу, чтобы в сгенерированном коде встречалось очевидное и легко устранимое дублирование, но я не буду переживать о нем в той же степени, как при самостоятельном написании кода. В конце концов, я не должен беспокоиться о возможности его изменения, а только о возможном чтении. Если мне кажется, что некоторое дублирование сделает код более понятным, я, конечно, оставлю его нетронутым. Я также буду использовать комментарии, тем более что я могу гарантировать, что сгенерированные комментарии будут соответствовать сгенерированному коду, а не окажутся случайным остатком от версии, которой уже давно нет. Комментарии могут ссылаться на структуры в семантической модели (*Semantic Model* (171)). Я также при выборе между ясной структурой и производительностью предпочтую последнюю — впрочем, как и в рукописном коде.

8.5. Предварительный анализ генерации кода

На протяжении большей части этого раздела я рассматривал генерацию кода как вывод сценария DSL, но генерация кода может играть важную роль в решении и других задач. В некоторых случаях при написании сценария DSL необходима интеграция с какой-то внешней информацией. Если вы пишете DSL, связывающий территории с продавцами, то вам может понадобиться координация своих действий с корпоративной базой данных, используемой продавцами. Вы можете захотеть убедиться, что используемые в вашем сценарии символы соответствуют таковым в корпоративной базе данных. Один из способов сделать это состоит в использовании генерации кода для генерации необходимой при написании сценариев информации. Часто такой вид проверки может быть выполнен при заполнении семантической модели (*Semantic Model* (171)), но иногда полезно иметь эту информацию и в исходном коде, в особенности для навигации по коду и статической типизации.

Примером этого может быть ситуация, когда вы пишете внутренний DSL на Java/C# и хотите, чтобы ваши символы, относящиеся к продавцам, были статически типизированными. Вы можете сделать это путем генерации перечислений с продавцами с последующим импортом этих перечислений в свои файлы сценариев [18].

8.6. Источники дополнительной информации

Самая большая книга по методам генерации кода — [17]. Кроме того, можно ознакомиться с набором шаблонов Маркуса Вольтера (Marcus Voelter) в [27].

Глава 9

Языковые инструментальные средства

Методы, о которых шла речь до сих пор, в той или иной форме известны уже давно. Инструменты для их поддержки, такие как генераторы синтаксических анализаторов (*Parser Generator* (277)) для внешних DSL, также имеют долгую историю. В этой же главе я собираюсь затратить некоторое время на рассмотрение набора новеньких блестящих инструментов, которые я называю языковыми инструментальными средствами.

Языковые инструментальные средства, по сути, представляют собой инструменты, которые помогают создать собственный DSL и обеспечивают его поддержку в стиле современных интегрированных сред разработки. Идея заключается в том, что эти инструменты не просто предоставляют интегрированную среду разработки для помощи в создании DSL; они поддерживают построение интегрированных сред для редактирования этих DSL. Таким образом, тот, кто пишет сценарий DSL, имеет такую же поддержку, как и пользователь сред следующего за IntelliJ поколения.

Сейчас, когда я пишу эти строки, языковые инструментальные средства еще очень молоды. Большинство из них едва прошли стадию бета-тестирования. И даже те, которые уже работают некоторое время, пока что не получили достаточного опыта, чтобы можно было делать какие-то серьезные выводы. Но их потенциал огромен — это инструменты, которые могут изменить лицо программирования. Я не знаю, будет ли это начинание успешным, но я уверен, что оно стоит того, чтобы за ним следить.

Об языковых инструментальных средствах особенно трудно писать. Я долго думал о том, о чем именно должна идти речь в этой книге. В конце концов я решил, что факт новизны этих инструментов означает, что я не должен много писать о них в такой книге, как эта. Многое из того, о чем я пишу сейчас, станет устаревшим, когда вы будете его читать. Как всегда, я ищу основные принципы, которые должны остаться неизменными, но их трудно вычленить в столь быстро меняющейся области. Поэтому я решил просто посвятить данной теме одну эту главу и не предоставлять никаких подробных сведений в справочном разделе книги. Я также решил охватить лишь несколько аспектов языковых инструментальных средств, которые кажутся мне относительно стабильными. Тем не менее вы должны принять эту главу с осторожностью и узнать о последних событиях и достижениях в этой области в веб.

9.1. Элементы языковых инструментальных средств

Хотя языковые инструментальные средства сильно отличаются друг от друга своим видом, у них есть общие элементы. В частности, языковые инструментальные средства позволяют определить три аспекта сред DSL.

- **Схема семантической модели** (*Semantic Model (171)*) определяет структуру данных семантической модели вместе со статической семантикой, как правило, с применением метамодели.
- **Среда редактирования DSL** определяет богатый опыт редактирования для людей, пишущих сценарии DSL, либо путем редактирования исходного текста, либо путем проекционного редактирования.
- **Поведение семантической модели** определяет, что делает DSL-сценарий, путем создания семантической модели, чаще всего с использованием генерации кода.

Языковые инструментальные средства применяют семантическую модель как базовую часть системы. В результате они предоставляют инструменты, которые помогают определить эту модель. Вместо семантической модели с языком программирования, как предполагается во всей этой книге, языковые инструментальные средства определяют семантическую модель в специальной структуре метамоделирования, что позволяет использовать для работы с моделью инструменты времени выполнения. Эта структура метамоделирования помогает обеспечить высокую степень инструментального оснащения.

В результате возникает разделение между схемой и поведением. Схема семантической модели, по сути, является моделью данных без особого поведения. Поведенческие аспекты семантической модели приходят извне структуры данных, в основном в форме генерации кода. Некоторые инструменты позволяют создавать интерпретаторы, но до сих пор самым популярным способом запуска семантической модели является генерация кода.

Одним из наиболее интересных и важных аспектов языковых инструментальных средств являются их среды редактирования. Это, пожалуй, ключевой аспект привносимого ими в разработку программного обеспечения, обеспечивающий гораздо более богатый набор инструментов для заполнения семантической модели и работы с ней. Они варьируются от чего-то близкого к текстовым редакторам, до графических редакторов, которые позволяют писать сценарии DSL в виде диаграмм, и до сред, использующих для обеспечения опыта, более близкого к работе с электронными таблицами, чем с обычным языком программирования, то, что я называю иллюстративным программированием.

Рассматривая языковые инструментальные средства глубже, мы столкнемся с проблемами, вызванными новизной и изменчивостью данного инструментария. Однако есть несколько общих принципов, которые, как мне кажется, останутся актуальными более продолжительное время. Это определение схемы и проекционное редактирование.

9.2. Языки определения схем и метамодели

В этой книге постоянно подчеркивались преимущества применения семантической модели (*Semantic Model (171)*). Каждое языковое инструментальное средство рассматривалось с точки зрения использования семантической модели и предоставления инструментов для ее определения.

Существует заметное различие между моделями языковых инструментальных средств и семантическими моделями, которые использовались в этой книге до сих пор. Как фанат объектно-ориентированного программирования я, естественно, строю объектно-ориентированные семантические модели, сочетающие в себе как структуры данных, так и поведение. Однако языковые инструментальные средства работают не так. Они предоставляют среду для определения схемы модели, т.е. ее структуры данных, как правило, с использованием определенного DSL — языка определения схемы. Затем они оставляют поведенческую семантику в качестве отдельного упражнения, как правило, посредством генерации кода.

Здесь в общую картину добавляется слово “мета”, и все начинает выглядеть так, как на рисунках Эшера³. Это сделано потому, что язык определения схемы имеет семантическую модель, которая является моделью сама по себе. Семантическая модель языка определения схемы является метамоделью для семантической модели DSL. Но язык определения схемы сам нуждается в схеме, которая определяется с помощью семантической модели, метамодель которого представляет собой язык определения схемы, метамодель которого... (бесконечная рекурсия).

Если предыдущий абзац не является для вас совершенно понятным и логичным, попробую еще раз и медленно...

Начну с фрагмента примера тайника, а именно — с **перехода из активного состояния в состояние ожидания подсветки**. Этот фрагмент диаграммы показан на рис. 9.1.

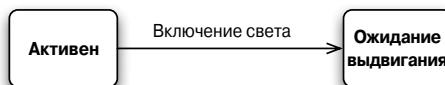


Рис. 9.1. Простая диаграмма состояний выключателя освещения

Здесь показаны два состояния и переход между ними. С помощью семантической модели, которая была приведена во введении (раздел “Модель конечного автомата”, с. 31), я интерпретирую приведенную модель как два экземпляра класса состояния и один экземпляр класса перехода, используя Java-классы и поля, определенные для семантической модели. В данном случае схема семантической модели представляет собой определения Java-классов. В частности, мне нужны четыре класса: состояния, события, строки и перехода. Вот упрощенный вид этой схемы.

```

class State {
    ...
}

class Event {
    ...
}

class Transition {
    State source, target;
    Event trigger;
    ...
}
  
```

Код Java — это лишь один из способов представления данной схемы. Можно, например, прибегнуть к способу, заключающемуся в использовании диаграммы классов (рис. 9.2).

Схема модели определяет, что может находиться в ее содержимом. Я не могу добавить защиту к своим переходам на диаграмме состояния иначе как добавив ее в схему. Это то же самое, что и любое определение структуры данных: классы и экземпляры, таблицы и строки, типы записей и записи. Схема определяет, что именно входит в экземпляры.

³ Мауриц Корнелис Эшер (Maurits Cornelis Escher, 1898–1972) — нидерландский художник-график. Известен, прежде всего, своими концептуальными литографиями, гравюрами на дереве и металле, на которых он мастерски исследовал пластические аспекты понятий бесконечности и симметрии, а также особенности психологического восприятия сложных трехмерных объектов. — *Примеч. пер.*

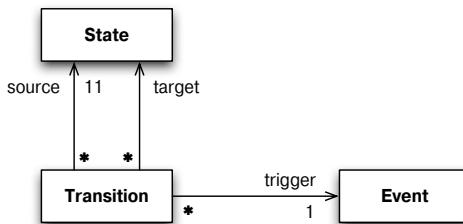


Рис. 9.2. Диаграмма классов для схемы простого конечного автомата

В нашем случае схема представляет собой определения Java-классов, но может существовать схема, являющаяся набором *объектов* Java, а не классов. Это позволит мне управлять схемой во время выполнения. Грубую версию этого подхода можно создать с помощью трех классов Java — для классов, полей и объектов.

```

class MClass...
    private String name;
    private Map<String, MField> fields;

class MField...
    private String name;
    private MClass target;

class MObject...
    private String name;
    private MClass mclass;
    private Map<String, MObject> fields;
  
```

Я могу использовать эту среду для создания схемы состояний и переходов.

```

private MClass state, event, transition;
private void buildTwoStateSchema() {
    state = new MClass("State");
    event = new MClass("Event");

    transition = new MClass("Transition");
    transition.addField(new MField("source", state));
    transition.addField(new MField("target", state));
    transition.addField(new MField("trigger", event));
}
  
```

Затем я могу воспользоваться этой схемой для определения простой модели, показанной на рис. 9.1.

```

private MObject active, waitingForDrawer,
        transitionInstance, lightOn;
private void buildTwoStateModel() {
    active = new MObject(state, "active");
    waitingForDrawer =
        new MObject(state, "waiting for drawer");
    lightOn = new MObject(event, "light on");
    transitionInstance = new MObject(transition);
    transitionInstance.set("source", active);
    transitionInstance.set("trigger", lightOn);
    transitionInstance.set("target", waitingForDrawer);
}
  
```

Иногда полезно рассматривать такую структуру как две модели (рис. 9.3). Базовая модель — фрагмент системы мисс Грант; она содержит объекты типа `MObject`. Вторая модель содержит объекты типов `MClass` и `MField` и обычно известна как метамодель. **Метамодель** является моделью, экземпляры которой определяют схему другой модели.

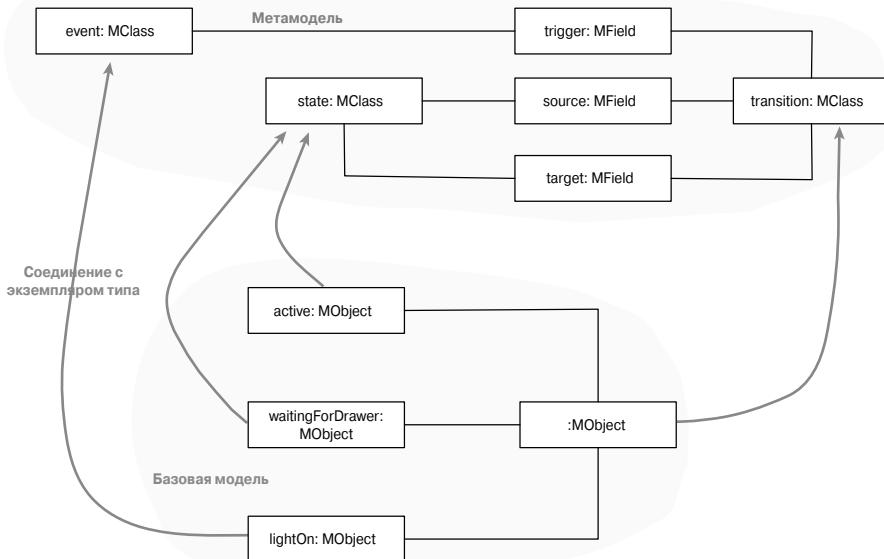


Рис. 9.3. Метамодель и базовая модель конечного автомата

Поскольку метамодель является просто другой семантической моделью, я могу легко определить DSL для ее заполнения (так же, как для ее базовой модели). Я называю такой DSL **языком определения схемы**. Язык определения схемы — это просто форма модели данных с некоторым способом определения сущностей и связей между ними. Имеется много языков определения схем и метамоделей.

При создании DSL вручную обычно нет особого смысла в построении метамодели. В большинстве ситуаций лучшим выбором является использование возможностей структурных определений вашего базового языка. В этом случае вы используете знакомые вам языковые конструкции как для схемы, так и для экземпляров. В моем грубом примере, если я хочу найти исходное состояние перехода, я должен написать нечто вроде `aTransition.get("source")`, а не `aTransition.getSource()`. Это усложняет поиск доступных полей, заставляет меня делать собственную проверку типов и т.д. Получается, что я работаю вопреки языку, а не вместе с ним.

Возможно, самым большим аргументом в пользу отказа от использования метамодели в этой ситуации является то, что я теряю способность сделать свою семантическую модель истинной объектно-ориентированной моделью предметной области. Хотя метамодель и выполняет работу по определению структуры моей семантической модели, определить ее поведение в действительности весьма трудно. Если я хочу иметь соответствующие объекты, которые сочетают как данные, так и поведение, то я гораздо лучше справлюсь с определением схемы, если использую для этого собственный механизм языка.

Эти компромиссы в случае языковых инструментальных средств выглядят иначе. Для того чтобы предоставить хороший инструмент, языковое инструментальное средство должно изучать схему любой определяемой мною модели и работать с ней, и эта работа обычно гораздо проще при использовании метамодели. Кроме того, оснастка языковых инструментальных средств позволяет преодолевать многие из общих недостатков использования метамодели. В результате большинство языковых инструментальных средств прибегают к использованию метамоделей.

Метамодель — это, конечно, всего лишь модель. И как и любая другая модель, она имеет схему для определения ее структуры. В моем грубом примере это схема, которая описывает `MClass`, `MField` и `MObject`. Но нет никакой логической причины, по которой эта схема не может быть определена с помощью метамодели. Это и позволяет использовать собственные инструменты моделирования языкового инструментального средства для работы над определением схемы самой системы, позволяя создавать метамодели с помощью тех же инструментов, которые используются для написания сценариев DSL. В сущности, язык определения схемы представляет собой просто еще один DSL языкового инструментального средства.

Многие языковые инструментальные средства принимают этот подход, который я называю самонастройкой языкового инструментального средства. В общем случае такой подход дает большую уверенность в том, что инструментов моделирования будет достаточно для работы, так как инструмент может определять сам себя.

Но одновременно вам начинает казатьсяся, что вы попали на гравюру Эшера. Если модели определяются с помощью метамодели, которая представляет собой просто модель, определяемую с помощью метамодели, то будет ли у этого конец? На практике инструменты определения схемы в некотором смысле являются специализированными, и есть вещи, которые жестко прошиты в языковом инструментальном средстве, чтобы обеспечить его работу. По сути, главная особенность модели определения схемы — она способна определить сама себя. Так, хотя вы можете представить себя поднимающимся по бесконечной лестнице метамоделей, в какой-то момент вы достигаете модели, которая может определять сама себя. Конечно, выглядит это довольно странно и запутанно, так что мне кажется, что проще об этом слишком сильно не задумываться.

Существует распространенный вопрос “В чем разница между языком определения схемы и грамматикой?” Короткий ответ — грамматика определяет конкретный синтаксис (текстового) языка, а язык определения схемы — структуру схемы семантической модели. Грамматика, таким образом, включает множество вещей, которые описывают входной язык, в то время как язык определения схемы не зависит от любого DSL, используемого для заполнения семантической модели. Грамматика также подразумевает структуру дерева синтаксического анализа; вместе с правилами построения дерева она может определять структуру синтаксического дерева. Но синтаксическое дерево обычно отличается от семантической модели (как уже говорилось в разделе “Работа синтаксического анализатора” на с. 67).

Определяемую схему можно рассматривать в терминах структур данных: классов и полей. Действительно, большая часть определения схемы посвящена логической структуре данных, в которой можно хранить элементы семантической модели. Но есть дополнительный элемент, который может появиться в схеме, — структурные ограничения. Это ограничения на создание корректных экземпляров семантической модели, эквивалентные инвариантам дизайна по контракту (*Design by Contract* [21]).

Структурные ограничения обычно представляют собой правила проверки, которые выходят за пределы того, что может быть выражено в определении структуры данных. Определение структуры данных само по себе предполагает ограничения — мы не можем

сказать в семантической модели ничего такого, что не в состоянии хранить ее схема. Наша рассматривавшаяся выше модель состояний гласит, что для перехода имеется только одно целевое состояние; мы не можем добавить еще одно, потому что его некуда поместить. Это ограничение, которое определяется и обеспечивается структурой данных.

Говоря о структурных ограничениях, мы обычно подразумеваем не те, которые возникают из-за структуры данных. Мы можем сохранить нечто, но это нечто — неверное. Это может быть ограничение на структуру данных, такое как ограничение количества ног у человека значениями 0, 1 или 2, даже если мы сохраняем этот атрибут в целочисленном поле. Ограничения могут быть сколь угодно сложными, включающими ряд полей и объектов (например, что человек не может быть собственным предком).

Языки определения схем часто обладают каким-то способом выражения структурных ограничений. Это может ограниченный способ, например указание для атрибутов диапазонов допустимых значений, а может быть и язык общего назначения, позволяющий выразить любые ограничения. Одно обычное ограничение заключается в том, что структурные ограничения не могут изменять семантическую модель; они могут только запрашивать ее. Таким образом, эти ограничения представляют собой систему правил вывода (Production Rule System (503)).

9.3. Редактирование исходного текста и проекционное редактирование

Одной из наиболее примечательных особенностей многих языковых инструментальных средств является использование ими проекционной системы редактирования, а не редактирования исходного текста, к которому привыкло большинство программистов. Система редактирования **на основе исходных текстов** определяет программу с помощью представления, редактируемого независимо от инструментов, применяемых для работы с этим представлением в работающей системе. На практике это означает текстовое представление, т.е. программа может быть прочитана и отредактирована с помощью любого инструмента для редактирования текста. Это — текст исходного кода программы. Он превращается в выполняемый вид путем передачи компилятору или интерпретатору, но ключевым представлением остается исходный текст, редактируемый и сохраняемый программистами.

В системе **проекционного редактирования** основное представление программы хранится в формате, специфичном для использующего его инструмента. Этот формат представляет собой хранимое представление семантической модели (*Semantic Model (171)*), используемое инструментом. Для того чтобы отредактировать программу, мы запускаем среду редактирования этого инструмента, которая обеспечивает возможность чтения и редактирования соответствующих представлений семантической модели. Эти представления могут быть как в текстовом виде, так и в виде диаграмм, таблиц или форм.

Настольные инструменты для работы с базами данных, такие как Microsoft Access, представляют собой хорошие примеры систем проекционного редактирования. Вы никогда не видите, не говоря уже о возможности редактировать, текстовый исходный код программы Access. Вместо этого вы запускаете Access и используете различные инструменты для изучения схемы базы данных, отчетов, запросов и т.д.

Проекционное редактирование дает вам ряд преимуществ по сравнению с редактированием исходных текстов. Наиболее очевидным является то, что оно допускает редактирование в форме различных представлений. Конечный автомат зачастую лучше рассматривать с применением диаграмм, а проекционное редактирование обеспечивает возможность представить конечный автомат в виде диаграммы и редактировать его непосредственно в этом виде. При использовании подхода с редактированием исходного текста вы можете

вносить изменения только в исходный код; и хотя вы можете пропустить этот текст через визуализатор и увидеть диаграмму, изменить ее непосредственно невозможно.

Такая проекция позволяет управлять редактированием, упрощая ввод корректной информации и не допуская ввод некорректной. Так, текстовая проекция может для вызова метода объекта показать вам только корректные методы этого класса и позволить ввести только допустимое имя метода. Это усиливает обратную связь между редактором и программой и позволяет редактору существенно помогать программисту.

У вас может быть несколько проекций, как одновременно, так и в качестве альтернатив. Демонстрация языкового инструментального средства от Intentional Software показывает условное выражение в С-подобном синтаксисе. С помощью команды меню можно увидеть то же выражение в Lisp-подобном синтаксисе или в табличной форме. Это позволяет использовать ту проекцию, которая наилучшим образом подходит для представления имеющейся у вас информации или которую предпочитает тот или иной программист. Часто желательно использовать несколько проекций для представления одной и той же информации, например показ суперкласса класса в виде поля в форме, а также (на другой панели среды редактирования) в иерархии классов. Редактирование любого представления приводит к обновлению базовой модели и отражается во всех проекциях.

Эти представления являются проекциями базовой модели и тем самым поддерживают семантические преобразования этой модели. Так, переименовать метод можно в терминах модели, а не в терминах текстового представления. Это позволяет вносить множество изменений в семантических терминах, как выполнение операций над семантической моделью, а не как действия с текстом. Это особенно полезно для безопасного и эффективного выполнения рефакторинга.

Проекционное редактирование далеко не ново — оно существовало по крайней мере все то время, которое я занимаюсь программированием. Оно имеет много преимуществ, но наиболее серьезное программирование выполняется на уровне исходных текстов. Проекционные системы связывают вас с конкретным инструментом, что не только заставляет нервничать по поводу замкнутости на конкретном производителе, но и затрудняет создание экосистемы, в которой над общим представлением могли бы сотрудничать несколько инструментов. Текст, несмотря на свои многочисленные недостатки, является общим форматом, так что с ним можно широко использовать самые разные инструменты.

Особенно хорошим примером большой разницы между проекционным и текстовым редактированием является управление исходным кодом. В этой области в течение последних лет было сделано немало интересных разработок, включая параллельное редактирование, представление различий, автоматизированное слияние, транзакционные обновления хранилищ и распределенные системы контроля версий. Все эти инструменты работают с широким спектром сред программирования именно потому, что они имеют дело только с текстовыми файлами. В результате мы видим печальную ситуацию, когда много инструментов, которые могли бы реально использовать интеллектуальные хранилища, различия и слияния, не в состоянии делать это. Это серьезная проблема для больших программных проектов, которая является одной из причин, по которым большие программные системы по-прежнему тяготеют к редактированию исходных текстов.

Такое редактирование имеет и другие преимущества. Если вы отправляете кому-то электронное письмо, чтобы объяснить, как сделать то или это, легко добавить в письмо фрагмент исходного текста, в то время как использовать для объяснений проекции и копии экранов гораздо менее удобно. Некоторые преобразования могут быть хорошо автоматизированы с помощью инструментов обработки текста, что очень полезно, если проекционная система не обеспечивает необходимое преобразование. Кроме того, хотя то,

что проекционная система допускает только корректный вход, может быть весьма полезно, часто в процессе обдумывания удобно ввести нечто как временный шаг, то, что не будет работать сразу. Разница между полезными ограничениями и ограничениями на размышления часто очень тонка.

Одним из триумфов современных интегрированных сред разработки является то, что они позволяют получить лучшее из обоих миров. По существу, вы работаете с исходным текстом, со всеми вытекающими отсюда преимуществами; однако при загрузке вашего исходного текста в интегрированную среду разработки она создает семантическую модель, которая позволяет ей использовать все проекционные технологии, чтобы облегчить редактирование. Я называю этот подход **редактированием исходного текста с помощью модели**. Для этого требуется много ресурсов; соответствующий инструмент должен проанализировать все исходные тексты и требует много памяти для хранения семантической модели, но зато результат приближается к лучшему из обоих миров. Однако обеспечить такие возможности, включая сохранение вносимых программистом изменений, — задача не из легких.

9.3.1. Множественные представления

При рассмотрении исходных текстов и проекционного редактирования я нахожу удобной концепцию ролей представления. Исходный код играет две роли: представления редактирования (представления программы, которую мы редактируем) и представления хранения (представления, которое хранится в перманентной форме). Компилятор преобразует такое представление в виде исходного текста в выполняемое, т.е. в такое, которое можно выполнить на компьютере. В случае интерпретируемого языка исходный текст одновременно является выполняемым представлением.

В какой-то момент, такой как компиляция, генерируется абстрактное представление. Это чисто компьютерно-ориентированная конструкция, которая облегчает обработку программы. Современные интегрированные среды разработки генерируют абстрактное представление для того, чтобы обеспечить помочь программисту при редактировании им исходных текстов. Может быть несколько абстрактных представлений; представление, используемое интегрированной средой разработки для редактирования, может отличаться от синтаксического дерева, используемого компилятором. Современные компиляторы часто создают множественные абстрактные представления для различных целей, например синтаксическое дерево — для одних целей и граф вызовов — для других.

В случае проекционного редактирования эти представления организуются по-разному. Основное представление — это используемая инструментом семантическая модель (**Semantic Model (171)**). Это представление проецируется на несколько представлений редактирования. Модель хранится с использованием отдельного представительства хранения. Представление хранения на определенном уровне может быть удобочитаемым для человека — например, сериализованное в XML, — но это не то представление, которое здравомыслящий человек будет использовать для редактирования.

9.4. Иллюстративное программирование

Возможно, самым интригующим следствием проекционного редактирования является его поддержка того, что я называю иллюстративным программированием. В обычном программировании мы уделяем основное внимание программе, которая представляет собой обобщенную инструкцию о том, что должно быть сделано. Обобщенность заключа-

ется в том, что это текст, описывающий общий случай, приводящий к разным результатам для разных входных данных.

Однако самая популярная в мире среда программирования работает не так. Эта наиболее популярная среда, на мой ненаучный взгляд, — электронные таблицы. Их популярность особенно интересна потому, что большинство программистов электронных таблиц — **непрофессионалы**, люди, которые не считают себя программистами.

Наиболее очевидная особенность электронных таблиц — иллюстративные вычисления с набором чисел. Программа спрятана в строке формул, видимой в каждый момент времени только для одной ячейки. Таблица “сплавляет” выполнение и определение программы в единое целое и заставляет сосредоточиться на выполнении. Предоставление иллюстрации в виде конкретного вывода программы помогает понять, что делает определение программы, так что можно быстрее разобраться в ее поведении. Это свойство, присущее любому интенсивному использованию тестирования, но с той разницей, что в электронной таблице результаты теста видны лучше, чем сама программа.

Я выбрал для описания ситуации термин “иллюстративное программирование” отчасти потому, что термин “пример” используется очень активно (в отличие от термина “иллюстрация”), а также потому, что термин “иллюстрация” подчеркивает разъяснительную природу примера выполнения программы. Иллюстрации предназначены для помощи в объяснении концепции, позволяя взглянуть на ситуацию с другой точки зрения, — и в случае электронных таблиц иллюстративное выполнение помогает увидеть, что делает ваша программа при внесении в нее изменений.

При попытке сделать концепцию явной, наподобие описанной выше, полезно рассмотреть предельные случаи. Один из них — использование проекции информации программы во время редактирования, как, например, в интегрированной среде разработки, которая показывает иерархию классов при работе над кодом. В некотором смысле это похоже на электронные таблицы, так как выводимая иерархия постоянно обновляется в процессе изменения программы, но принципиальная разница в том, что иерархия может быть выведена из статической информации о программе. Иллюстративное программирование требует информации от реального выполнения программы.

Я также рассматриваю иллюстративное программирование как более широкое понятие, чем просто способность легко запустить фрагмент кода в интерпретаторе — эту любимую возможность динамических языков. Интерпретация фрагментов позволяет изучить выполнение программы, но не выносит примеры на первый план, как поступает со своими значениями электронная таблица. Методы иллюстративного программирования выводят иллюстрацию на авансцену редактирования. Программа отступает на задний план, выглядывая только тогда, когда мы хотим изучить части иллюстрации.

Я не считаю, что иллюстративное программирование — это всегда хорошо. Одна из проблем, с которыми я сталкивался при работе с электронными таблицами и с разработчиками графического пользовательского интерфейса, заключается в том, что при хорошей работе по выявлению действий программы они затеняют структуру программы. В результате сложные электронные таблицы и панели пользовательского интерфейса недавно трудно понять и изменить. Зачастую это приводит к неконтролируемому применению метода “скопировать и вставить”.

Все это представляется мне следствием того, что программа отходит на второй план, предоставляя первый иллюстрации, и в результате программисты часто даже не думают позаботиться о программе и ее структуре. Мы страдаем от недостаточного внимания к программам даже в обычном программировании, так что вряд ли вас удивит то, что это происходит и с иллюстративными программами, написанными непрофессионалами. Но эта проблема приводит к появлению программ, которые по мере роста быстро становятся

неподдерживаемыми. Важной задачей для будущих иллюстративных сред программирования является помочь в разработке хорошо структурированных программ за кулисами иллюстраций — хотя иллюстрации также могут заставить нас задуматься о том, что же такое хорошо структурированная программа.

Еще одной трудной задачей может оказаться возможность легко создавать новые абстракции. Одним из моих наблюдений за программным обеспечением с богатым пользовательским интерфейсом является то, что он часто слишком запутан, так как его создатели работают только в терминах экранов и управляющих элементов. Мои эксперименты показывают, что для программы необходимо найти правильные абстракции. Но такие новые абстракции не поддерживаются построителем экрана, который в состоянии проиллюстрировать только те абстракции, о которых знает.

Несмотря на эту проблему иллюстративное программирование представляет собой технологию, к которой следует относиться более серьезно. Нельзя игнорировать тот факт, что электронные таблицы стали настолько популярны среди непрофессиональных программистов. Многие языковые инструментальные средства фокусируют свое внимание на обеспечении работы именно непрофессионалов, и проекционное редактирование приводит к иллюстративному программированию, которое может оказаться жизненно важной частью его конечного успеха.

9.5. Тур по инструментам

До сих пор я неохотно упоминал о каких-либо реальных языковых инструментальных средствах. В столь изменчивой области любая информация, вероятно, окажется устаревшей к тому моменту, когда книга попадет к вам в руки. Но я решил все же сделать это, чтобы вы могли почувствовать все разнообразие инструментов этого мира. Помните, однако, что детальная информация о тех или иных инструментах почти наверняка не будет актуальной, когда вы будете ее читать.

Возможно, самым влиятельным и, определенно, самым сложным является Intentional Workbench от Intentional Software (<http://intentsoft.com>). Этим проектом руководит Чарльз Симони (Charles Simonyi), который хорошо известен своими новаторскими работами в PARC над ранними текстовыми редакторами и ведущей ролью в разработке Microsoft Office. Его целью является среда с высоким уровнем совместной работы, объединяющая программистов и не программистов в едином интегрированном инструменте. В результате Intentional Workbench обладает очень богатыми возможностями проекционного редактирования и сложным репозиторием метамоделирования.

Самая большая критика в адрес Intentional заключается в критике длительности и скрытности их работы. Они также были весьма активны в патентовании, что вызвало массу тревог у работающих в данной области. Первые публичные презентации, относящиеся к началу 2009 года, продемонстрировали весьма неплохой инструмент. Он поддерживает все виды проекций: текст, таблицы, диаграммы, иллюстрации и любые их комбинации.

Хотя Intentional является старейшим языковым инструментальным средством в смысле продолжительности разработки, я считаю, что старейший выпущенный инструмент — это MetaEdit от MetaCase (www.metacase.com). Он, в первую очередь, сосредоточен на графических проекциях, но поддерживает и табличные проекции (но не текст). Необычно то, что он не является самодостаточной средой; для определения схем и проекций требуется прибегать к специальной среде. Microsoft имеет группу инструментов DSL с аналогичным стилем.

Инструмент Meta-Programming System (MPS) от JetBrains (www.jetbrains.com) по-другому подходит к проекционному редактированию, предпочитая структурированное представление текста. Он также гораздо больше нацелен на повышение производительности труда программиста, а не на активное участие в DSL экспертов предметной области. Компания JetBrains значительно улучшила возможности интегрированных сред разработки с их интеллектуальным редактированием кода и средствами навигации и имеет прочную репутацию среди разработчиков инструментов для программирования. Они виляют в MPS основу для многих будущих инструментов. Особенно важным моментом является то, что в основном код MPS — открытый исходный код. Это может быть жизненно важным фактором при принятии разработчиками решения перейти к совсем другой интегрированной среде разработки.

Еще одной системой с открытым исходным кодом является Xtext (www.eclipse.org/Xtext), построенный поверх Eclipse. Xtext заметно отличается тем, что использует редактирование исходных текстов, а не проекционное редактирование. В ней в качестве базового синтаксического анализатора используется ANTLR, и она интегрируется с Eclipse для обеспечения редактирования сценариев DSL в стиле, аналогичном стилю редактирования Java в Eclipse.

Проект SQL Server Modeling от Microsoft (ранее известный как “Oslo”) использует сочетание исходных текстов и проекций. В нем есть язык моделирования, в настоящее время называющийся M, который позволяет определять схему семантической модели (*Semantic Model* (171)) и грамматику для текстового DSL. Затем инструмент создает плагин для интеллектуального редактора, что дает возможность редактирования исходных текстов с участием модели. Получающиеся в результате модели помещаются в хранилище реляционной базы данных, а диаграммный проекционный редактор (Quadrant) может с ними работать. Модели могут быть запрошены во время выполнения, так что вся система может работать полностью без генерации кода.

Я уверен, что этот небольшой тур ни в коей мере не является исчерпывающим, но позволяет бегло ознакомиться с разнообразным инструментарием в этом пространстве. Постоянно появляется много новых идей, и еще слишком рано предсказывать, какое сочетание технических и бизнес-идей приведет к успеху. Пока что чисто с точки зрения технической изощренности лидирует Intentional, но, как мы знаем из жизни, часто побеждает совсем не лучшее техническое решение.

9.6. Языковые инstrumentальные средства и CASE-инструменты

Некоторые люди находят множество параллелей между языковыми инструментальными средствами и CASE-инструментами, которые должны были революционизировать разработку программного обеспечения пару десятилетий назад.

Тем, кто пропустил эту сагу, инструменты CASE (сокращение от “Computer-Aided Software Engineering” — автоматизированное проектирование и создание программ) должны позволить сначала описать дизайн программного обеспечения с использованием различных диаграмм, а затем сгенерировать это программное обеспечение вместо вас. В 1990-х годах таким представлялось будущее программирования.

Да, на первый взгляд, некоторое сходство есть. Центральная роль модели, использование метамодели для ее определения и проекционное редактирование с диаграммами — все это характеристики CASE-инструментов.

Ключевым технологическим отличием является то, что CASE-инструменты не позволяют определить собственный язык. MetaEdit, пожалуй, является наиболее близким к CASE-инструментам языковым инструментальным средством, но его возможности по

определению собственного языка и генерации управляющего кода на базе вашей модели сильно отличаются от того, что предлагают CASE-средства.

Бытует мнение, что большую роль в области DSL и языковых инструментальных средств может играть MDA (Model-Driven Architecture — архитектура, управляемая моделью) от OMG (Object Management Group — рабочая группа по развитию стандартов объектного программирования). Я скептически отношусь к этому, так как нахожу стандарты OMG MDA слишком громоздкими для среды DSL.

Возможно, наиболее важное различие — культурное. Многие люди в мире CASE взглянули на программирование и увидели, что их роль сводится к автоматизации чего-то, что обречено на вымирание. Все больше людей приходят из программирования в языковые инструментальные средства и хотят создавать среды, которые сделают работу программистов более продуктивной (и расширят сотрудничество с клиентами и пользователями). Результатом этого является то, что языковые инструментальные средства, как правило, имеют сильную поддержку средств генерации кода, что имеет важное значение для конечного продукта. Этот аспект во время демонстраций, как правило, забывается, так как он менее захватывающий, чем проекционное редактирование, но к получающимся в результате инструментам следует отнести со всей серьезностью.

9.7. Следует ли использовать языковые инструментальные средства

Я не знаю, устали ли вы от чтения этого предупреждения так же, как я — от его написания, но я скажу это снова. Это новая изменчивая область, и все сказанное мною может легко стать недействительным ко времени, когда вы об этом прочтете. А теперь продолжим.

Я следил за этими средствами в течение нескольких последних лет, потому что считаю, что они имеют исключительный потенциал. При благоприятном стечении обстоятельств они в силах полностью преобразить лицо программирования, изменения сами наши представления о языке программирования. Я должен подчеркнуть, что пока что это только потенциал, который может закончиться, как потенциал термоядерного синтеза, который должен был давно обеспечить все наши энергетические потребности. Однако сам факт наличия такого потенциала требует отслеживания происходящего в этой области.

Однако новизна и изменчивость данной области означает, что пока что необходимо проявлять осторожность. Еще одна причина для осторожности в том, что программа изначально привязана к конкретному инструменту. Любой код, который вы создаете с помощью одного языкового инструментального средства, невозможно экспортовать в другое. Возможно, со временем появится некий стандарт взаимодействия, но обеспечить такое взаимодействие будет очень тяжело. В результате все усилия по работе в некотором языковом инструментальном средстве могут быть полностью напрасны, если у его производителя возникнут проблемы.

Одним из способов смягчения ситуации является рассмотрение языкового инструментального средства как синтаксического анализатора, а не как полной среды DSL. В случае полной среды DSL вы проектируете семантическую модель (*Semantic Model* (171)) в среде определения схемы языкового инструментального средства и генерируете полнофункциональный код. Рассматривая языковое инструментальное средство как синтаксический анализатор, вы все равно строите семантическую модель обычным способом. Затем вы используете языковое инструментальное средство для определения среды редактирования с моделью, которая ориентирована на генерацию, осведомленную о вашей семантической модели (*Model-Aware Generation* (545)). Таким образом, если вы столкнетесь с проблемами, связанными с вашим языковым инструментальным средством, это будут проблемы, влияющие только на синтаксический анализатор. Самое же

главное — в семантической модели, которая останется не задетой, и проблема решится путем применения альтернативного механизма синтаксического анализа.

Приведенные выше мысли, подобно применению языковых инструментальных средств, несколько спекулятивны. Но я считаю, что потенциал этих средств означает, что они по крайней мере стоят того, чтобы с ними поэкспериментировать. Хотя это и рискованные инвестиции, потенциально можно получить значительную прибыль.

Часть II

Общие вопросы

Глава 10

Зоопарк DSL

Как уже говорилось в начале этой книги, мир программного обеспечения полон DSL. Здесь я хочу привести краткий обзор лишь некоторых из них. Я не преследовал цель показать лучшее — это просто те предметно-ориентированные языки, с которыми я сталкивался лично и которые считаю подходящими для того, чтобы представить все разнообразие имеющихся DSL. Это крохотная доля существующих предметно-ориентированных языков, но я надеюсь, что даже она поможет вам получить верное представление об их разнообразии.

10.1. Graphviz

Graphviz — это хороший пример как DSL, так и полезного пакета для всех, кто работает с DSL. Это библиотека для создания визуализации графов. На рис. 10.1 показан пример с сайта Graphviz.

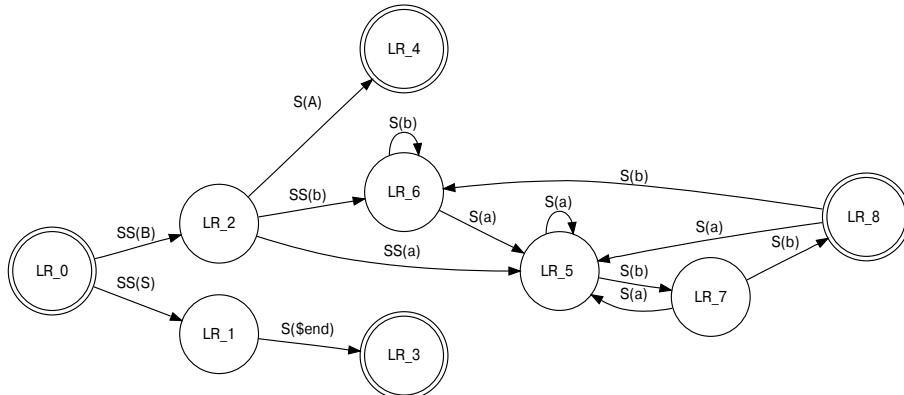


Рис. 10.1. Пример использования Graphviz

Для получения такой диаграммы необходимо предоставить Graphviz приведенный ниже фрагмент на языке DOT, который является внешним DSL.

```
digraph finite_state_machine {
    rankdir=LR;
    size="8,5"
    node [shape = doublecircle]; LR_0 LR_3 LR_4 LR_8;
    node [shape = circle];
    LR_0 -> LR_2 [ label = "SS(B) " ];
    LR_0 -> LR_1 [ label = "SS(S) " ];
    LR_2 -> LR_6 [ label = "S(a) " ];
    LR_2 -> LR_5 [ label = "SS(b) " ];
    LR_1 -> LR_3 [ label = "S($end) " ];
    LR_6 -> LR_4 [ label = "S(b) " ];
    LR_6 -> LR_5 [ label = "SS(a) " ];
    LR_6 -> LR_7 [ label = "S(a) " ];
    LR_5 -> LR_6 [ label = "S(a) " ];
    LR_5 -> LR_7 [ label = "S(b) " ];
    LR_5 -> LR_8 [ label = "S(a) " ];
    LR_7 -> LR_5 [ label = "S(b) " ];
    LR_7 -> LR_8 [ label = "S(a) " ];
    LR_8 -> LR_5 [ label = "S(b) " ];
}
```

```

LR_1 -> LR_3 [ label = "S($end) " ];
LR_2 -> LR_6 [ label = "SS(b) " ];
LR_2 -> LR_5 [ label = "SS(a) " ];
LR_2 -> LR_4 [ label = "S(A) " ];
LR_5 -> LR_7 [ label = "S(b) " ];
LR_5 -> LR_5 [ label = "S(a) " ];
LR_6 -> LR_6 [ label = "S(b) " ];
LR_6 -> LR_5 [ label = "S(a) " ];
LR_7 -> LR_8 [ label = "S(b) " ];
LR_7 -> LR_5 [ label = "S(a) " ];
LR_8 -> LR_6 [ label = "S(b) " ];
LR_8 -> LR_5 [ label = "S(a) " ];
}

```

В этом примере показаны два типа объектов в графе: узлы и дуги. Узлы объявляются с помощью ключевого слова `node`, но их можно и не объявлять. Дуги же объявляются с помощью оператора `->`. И для узлов, и для дуг можно задать атрибуты, перечисленные в квадратных скобках.

Graphviz использует семантическую модель (*Semantic Model* (171)) в форме структуры данных C. Семантическая модель заполняется синтаксическим анализатором с помощью синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)) и встраиваемой трансляции (*Embedded Translation* (305)), написанных на Yacc и C. Синтаксический анализатор позволяет эффективно использовать встроенный помощник (*Embedment Helper* (537)). Поскольку это язык программирования C, не являющийся объектно-ориентированным, здесь нет соответствующего объекта; вместо него используется набор вспомогательных функций, которые вызываются в действиях грамматики. В результате грамматика вполне удобочитаема, с коротким кодом действий, не препятствующим восприятию грамматики. Лексический анализатор написан вручную, что достаточно обычное явление при использовании синтаксических анализаторов, сгенерированных с помощью программы Yacc, несмотря на наличие генератора лексических анализаторов Lex.

Реальная работа Graphviz начинается, когда заполнена семантическая модель узлов и дуг. Пакет определяет, как расположить граф на выходной диаграмме, и включает код, который может вывести граф в различных графических форматах. Все эти действия не зависят от кода синтаксического анализатора; после того как сценарий превратился в семантическую модель, все остальное базируется на соответствующих структурах данных C.

В приведенном выше примере в качестве разделителя инструкций используется точка с запятой, но это совершенно необязательно.

10.2. JMock

JMock представляет собой библиотеку Java для *Mock Objects* [20]. Его авторы написали несколько библиотек мок-объектов, которые развивали их идеи о хорошем внутреннем DSL для определения ожиданий от объекта (эта эволюция хорошо описана в [13]).

Мок-объекты⁴ используются в тестировании. Вы начинаете тест с объявления **ожиданий**, представляющих собой методы, которые, как ожидается, будут вызваны во время теста. Затем мок-объект подключается к реальному тестируемому объекту и подает ему

⁴ *Mock-объект* (от англ. “mock object”, буквально — “объект-пародия”, “объект-имитация”) — тип объектов, реализующих заданные аспекты моделируемого программного окружения. Мок-объект представляет собой конкретную фиктивную реализацию интерфейса, предназначенному исключительно для тестирования. — Примеч. пер.

входные сигналы. После мок-объект сообщает, получал ли он вызовы корректных объектов, поддерживая тем самым верификацию поведения *Behavior Verification* [20].

Для иллюстрации DSL JMock я пройду через пару этапов его развития, начиная с первой библиотеки JMock (JMock 1), которую авторы [13] назвали “кайнозойской эрой” JMock. Вот пример небольшого ожидания.

```
mainframe.expects(once())
    .method("buy").with(eq(QUANTITY))
    .will(returnValue(TICKET));
```

Здесь говорится о том, что в качестве части теста объект `mainframe` (который представляет собой мок-объект) ожидает вызов метода `buy`. Параметром вызова должна быть константа `QUANTITY`. При вызове метод должен вернуть значение константы `TICKET`.

Ожидания Mock должны быть написаны с тестовым кодом, как фрагментарный DSL, поэтому естественным выбором для них является внутренний DSL. JMock 1 использует сочетание связывания методов мок-объекта в цепочки (Method Chaining (375)) (`expects`) и вложенной функции (Nested Function (361)) (`once`). Чтобы вложенные методы могли быть чистыми функциями, используется шаблон Object Scoping (387). Библиотека JMock 1 обеспечивает работоспособность последнего, требуя, чтобы все тесты с использованием мок-объектов были написаны в подклассе ее библиотечного класса.

Для того чтобы улучшить работу связывания методов в цепочки с интегрированными средами разработки, JMock использует прогрессивные интерфейсы. Таким образом, вызов `with` доступен только после вызова `method`, что позволяет интегрированной среде разработки использовать возможность автозаполнения и облегчить корректную запись ожиданий.

Для обработки вызовов DSL и их трансляции в семантическую модель (Semantic Model (171)) мок-объектов и ожиданий JMock использует построитель выражений (Expression Builder (349)). В [13] построители выражений именуются *синтаксическим слоем*, а семантическая модель — *слоем интерпретатора*.

Из взаимодействия связывания методов в цепочки и вложенных функций можно вынести интересный урок расширяемости. Пользователям сложно расширить связывание методов в цепочки, определенное построителем выражений, так как последним определены все методы, которые можно применять. Однако им легко добавлять новые методы во вложенные функции, так как вы определяете их в самом тестовом классе или используете собственный подкласс библиотечного суперкласса, применяемого шаблоном Object Scoping (387).

Этот подход хорошо работает, но все еще имеет некоторые недостатки. В частности, существует ограничение, что все тесты с использованием мок-объектов должны быть определены в подклассе библиотечного класса JMock, чтобы обеспечить перенос в область видимости объекта (Object Scoping (387)). В JMock 2, чтобы избежать этой проблемы, использован новый тип DSL; в этой версии то же ожидание описывается следующим образом.

```
context.checking(new Expectations() {
    oneOf(mainframe).buy(QUANTITY);
    will(returnValue(TICKET));
})
```

В этой версии для переноса в область видимости объекта JMock использует инициализацию экземпляра Java. Хотя это добавляет немного шума в начале выражения, в результате можно определить ожидания без размещения в подклассе. Инициализатор экземпляра принимает вид замыкания (Closure (397)), позволяя прибегнуть к вложенным

замыканиям (**Nested Closure** (403)). Стоит также отметить, что теперь для отделения в ожидании части описания вызова метода от части указания возвращаемого значения используется последовательность функций (**Function Sequence** (357)), а не связывание методов в цепочки.

10.3. CSS

Говоря о предметно-ориентированных языках, я часто использую пример CSS.

```
h1, h2 {
  color: #926C41;
  font-family: sans-serif;
}

b {
  color: #926C41;
}

*.sidebar {
  color: #928841;
  font-size: 80%;
  font-family: sans-serif;
}
```

CSS является прекрасным примером DSL по многим причинам. В первую очередь, это так, потому что большинство CSS-программистов считают себя не программистами, а веб-дизайнерами. CSS, таким образом, — хороший пример DSL, который специалисты в предметной области не только читают, но и пишут.

CSS является также хорошим примером благодаря своей декларативной вычислительной модели, которая существенно отличается от императивной. В нем нет указаний “сделай это, затем сделай то”, как в традиционных языках программирования. Вместо этого просто объявляются соответствующие правила для элементов HTML.

Эта декларативная природа усложняет понимание того, что и как происходит. В моем примере элемент `h2` внутри элемента `div` боковой панели будет соответствовать двум различным правилам выбора цвета. CSS имеет достаточно сложную, но четкую схему определения, какой именно цвет будет использован в такой ситуации. Однако многие находят ее слишком сложной для практического применения, и эту сложность понимания можно назвать темной стороной декларативной модели.

CSS играет четко определенную роль в “экосистеме” веб. Представление о его использовании только для создания веб-приложения является нелепым, хотя это и весьма важная задача. Он очень хорошо справляется со своей работой, но, кроме того, он хорошо работает и в сочетании с другими DSL и языками общего назначения для создания полных программных решений.

CSS также достаточно большой. В нем много элементов — как в базовой семантике языка, так и в семантике различных атрибутов. Хотя предметно-ориентированные языки и имеют ограниченные выразительные возможности, это еще не значит, что их легко изучать.

CSS вписывается в общепринятую практику ограниченной обработки ошибок в DSL. Браузеры спроектированы так, чтобы игнорировать ошибочные входные данные, а это обычно означает, что CSS-файл с синтаксическими ошибками часто просто о них умалчивает.

Как и в большинстве DSL, в CSS отсутствуют какие-либо способы создания новых абстракций — обычное следствие ограниченных выразительных возможностей DSL. Хотя это в основном нормально, есть некоторые возможности, отсутствие которых весьма

раздражает. В приведенном выше образце CSS показан такой пример — невозможность именования цветов, что приводит к необходимости использовать шестнадцатеричные представления. Многие жалуются на отсутствие арифметических функций, которые были бы полезны при работе с размерами и полями. Здесь применимы те же решения, что и в других DSL. Многие простые задачи, такие как именование цвета, могут быть решены с помощью макросов (Macros (195)).

Другим решением является написание похожего на CSS предметно-ориентированного языка, который в качестве вывода генерирует CSS. Примером такого подхода является SASS (<http://sass-lang.com>), обеспечивающий арифметические операции и переменные. Он использует совершенно иной синтаксис, предпочитая блочной структуре CSS синтаксически значащие символы новой строки и отступы. Это распространенное решение — использовать один DSL в качестве надстройки над другим для обеспечения отсутствующих в основном DSL абстракций. Такие предметно-ориентированные языки должны быть схожими с базовыми (SASS использует те же имена атрибутов), так что у их пользователей обычно не возникает проблем с пониманием базового DSL.

10.4. Hibernate Query Language (HQL)

Hibernate — широко используемая объектно-реляционная система отображений, которая позволяет отображать классы Java на таблицы реляционной базы данных. HQL предоставляет возможность писать запросы в форме SQLish в терминах классов Java, которые могут быть отображены на SQL-запросы к реальной базе данных. Такой запрос может выглядеть следующим образом.

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

Это позволяет программистам думать в терминах классов Java, а не таблиц базы данных, а также избегать раздражающих различий между диалектами SQL разных баз данных.

Суть работы HQL — в трансляции запросов HQL в запросы SQL. Hibernate делает это в три этапа.

- Входной текст HQL с помощью синтаксически управляемой трансляции (Syntax-Directed Translation (229)) и построения деревьев (Tree Construction (289)) преобразуется в абстрактное синтаксическое дерево HQL.
- Абстрактное синтаксическое дерево HQL преобразуется в абстрактное синтаксическое дерево SQL.
- Генератор кода генерирует код SQL на основе абстрактного синтаксического дерева SQL.

Во всех этих случаях используется ANTLR. В дополнение к потоку токенов в качестве входа синтаксического анализатора ANTLR, можно использовать ANTLR с абстрактным синтаксическим деревом в качестве входных данных (это то, что ANTLR называет “грамматикой дерева”). Синтаксис построения дерева ANTLR используется для создания как абстрактного синтаксического дерева HQL, так и абстрактного синтаксического дерева SQL.

Этот путь преобразований — *входной текст* \Leftrightarrow *входное абстрактное синтаксическое дерево* \Leftrightarrow *выходное абстрактное синтаксическое дерево* \Leftrightarrow *выходной текст* — обычный путь трансформации исходного текста в исходный текст. Как и во многих сценариях пре-

образования, желательно разбить сложное преобразование на несколько простых, которые можно легко объединить.

Абстрактное синтаксическое дерево SQL в данном случае можно рассматривать как семантическую модель (*Semantic Model* (171)). Смысл запросов HQL определяется преобразованием в SQL-запрос, и абстрактное синтаксическое дерево SQL представляет собой модель SQL. Чаще всего абстрактные синтаксические деревья не являются подходящими для семантической модели структурами, поскольку ограничения синтаксического дерева обычно больше помогают, чем мешают. Но для трансляции исходного текста в исходный текст применение абстрактного синтаксического дерева выходного языка имеет большой смысл.

10.5. XAML

С того самого времени, как появился первый полноэкранный пользовательский интерфейс, программисты экспериментировали с определением макета экрана. Тот факт, что это графическая среда, всегда приводит программистов к использованию некоторого вида графического инструмента для создания макета. Однако зачастую большая гибкость может быть достигнута путем размещения с помощью кода. Беда в том, что код может быть неудобным механизмом. Экран — это, в первую очередь, иерархическая структура, и “сшивание” иерархии в коде часто оказывается более неудобным, чем должно было бы быть. Поэтому с появлением Windows Presentation Framework компания Microsoft представила новый предметно-ориентированный язык для макетирования интерфейсов — XAML.

(Признаюсь, я удивлен банальностью именования компанией Microsoft своих продуктов. Хорошие имена типа “Avalon” и “Indigo” превратились в скучные сокращения WPF и WCF. Не превратится ли само название “Windows” в один прекрасный день в какое-нибудь WTF (от Windows Technology Foundation)?)

XAML-файлы представляют собой XML-документы, которые могут использоваться для макетирования объектной структуры; с помощью WPF экран может быть макетирован так, как в этом примере, взятом из [2].

```
<Window x:Class="xamlExample.Hello"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hello World">
    <WrapPanel>
        <Button Click='HowdyClicked'>Howdy!</Button>
        <Button>A second button</Button>
        <TextBox x:Name='_text1'>An editable text box</TextBox>
        <CheckBox>A check box</CheckBox>
        <Slider Width='75' Minimum='0' Maximum='100' Value='50' />
    </WrapPanel>
</Window>
```

Компания Microsoft — поклонница поверхностей графического проектирования, поэтому при работе с XAML можно использовать как поверхность проектирования, так и текстовое представление или и то, и другое. В качестве текстового представления XAML страдает от синтаксической зашумленности XML, но XML очень хорошо работает с иерархическими структурами наподобие рассматриваемых. Тот факт, что он очень похож на язык HTML, который также используется для макетирования экранов, также является плюсом.

XAML является хорошим примером того, что мой давний коллега Брэд Кросс (Brad Cross) называет композиционным (а не вычислительным) DSL. В отличие от моего первоначального примера конечного автомата, XAML предназначен для организации отно-

сительно пассивных объектов в структуру. Поведение программы обычно не очень зависит от макета экрана. Фактически одной из сильных сторон XAML является то, что он способствует отделению экрана от кода, который управляет его поведением.

Документ XAML логически определяет класс C#, и на деле имеет место некоторая генерация кода. Код генерируется как частичный класс, в данном случае — `xamlExample.Hello`. Можно добавить поведение путем написания кода в другом определении разделяемого класса.

```
public partial class Hello : Window {
    public Hello() {
        InitializeComponent();
    }
    private void HowdyClicked(object sender,
                               RoutedEventArgs e) {
        _text1.Text = "Hello from C#";
    }
}
```

Этот код позволяет связать с элементом поведение. События любого управляющего элемента, определенного в файле XAML, можно связать с методом обработчика в коде (`HowdyClicked`). Код для работы с управляющими элементами может обращаться к ним по имени (`_text1`). Применяя такие имена, я могу поддерживать ссылки отдельно от структуры макета пользовательского интерфейса, что позволяет изменять последний без необходимости обновлять код поведения.

XAML обычно рассматривается в контексте проектирования пользовательского интерфейса и почти всегда описывается вместе с WPF. Однако XAML может использоватьсь для связывания экземпляров любых CLR-классов, поэтому он может применяться и во многих других ситуациях.

Структура, определяемая XAML, представляет собой иерархию. Предметно-ориентированные языки могут определять не только иерархии, но и другие структуры путем упоминания имен. Фактически это делает Graphviz, используя ссылки на имена для определения структуры графа.

Предметно-ориентированные языки для описания схем графических структур (каковым является и рассматриваемый DSL) встречаются довольно часто. Так, Swiby (<http://swiby.codehaus.org>) использует внутренний DSL Ruby для определения схемы экрана. Он использует вложенное замыкание (Nested Closure (403)), которое обеспечивает естественный способ определения иерархической структуры.

Хотя здесь речь идет о DSL для графической схемы, я не могу не упомянуть PIC — старый и довольно увлекательный DSL. Он был создан на заре Unix, когда графический экран был еще исключением, а не правилом. Он позволяет описать диаграмму в текстовом формате, а затем обработать его и получить соответствующее изображение. Приведенный далее фрагмент кода создает диаграмму, показанную на рис. 10.2.

```
.PS
A: box "this"
move 0.75
B: ellipse "that"
move to A.s; down; move;
C: ellipse "the other"
arrow from A.s to C.n
arrow dashed from B.s to C.e
.PE
```

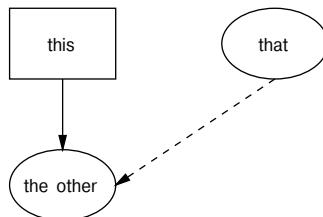


Рис. 10.2. Простая PIC-диаграмма

Написанное вполне понятно и очевидно; единственный намек — точки соединения фигур указываются с помощью концепции компаса, т.е. A.s означает южную оконечность фигуры A. Текстовые описания в стиле PIC не так популярны в эпоху сред WYSIWYG, но в ряде случаев этот подход может быть весьма удобен.

10.6. FIT

FIT (<http://fit.c2.com>) представляет собой систему тестирования, разработанную Вардом Каннингамом (Ward Cunningham) в начале 2000-х годов (FIT — Framework for Integrated Test, т.е. система для интегрированных тестов). Ее назначение — описание сценариев тестирования в форме, понятной эксперту предметной области. Основная идея была расширена с помощью различных инструментов, таких как Fitnesse (<http://fitnesse.org>).

Рассматривая FIT как DSL, следует упомянуть о некоторых вещах, которые делают ее интересной. Первое — это форма; ключевой концепцией FIT является то, что не программистам очень удобно указывать примеры в табличной форме. Так что FIT-программа представляет собой набор таблиц, как правило, внедренных в страницы HTML. В промежутках между таблицами можно разместить любые другие элементы HTML, которые рассматриваются как комментарии. Это позволяет экспертам в предметной области описывать на обычном языке свои намерения, оставлять комментарии и т.п.; одновременно таблицы предоставляют информацию, обрабатываемую DSL.

FIT-таблицы могут принимать различный вид. Ближе всего к программному виду описание действий, которое, по существу, является простым императивным языком. Он очень прост, так как в нем нет ни условных выражений, ни циклов — одни лишь глаголы.

`eg.music.Realtime`

enter	select	2	pick an album
press	same album		find more like it
check	status	searching	
await	search complete		
check	status	ready	
check	selected songs	2	

Каждая таблица подключена к процессору, который транслирует эти глаголы в действия с системой. Глагол `check` — специальный глагол в том смысле, что он выполняет сравнение. При запуске таблицы создается выходной HTML — такой же, как и входная

страница, но все проверенные строки окрашиваются в зеленый или красный цвет в зависимости от успешности выполненного сравнения.

Помимо этой ограниченной императивной формы FIT работает с рядом других стилей таблиц. Вот стиль, который определяет табличные выходные данные из списка объектов (в данном случае для определенного выше поиска).

eg.music.Display

title	artist	album	year	time()	track()
Scarlet Woman	Weather Report	Mysterious Traveller	1974	5.72	6 of 7
American Tango	Weather Report	Mysterious Traveller	1974	3.70	2 of 7

Строка заголовка определяет методы, которые будут вызваны для коллекции объектов в списке. Каждая строка сравнивается с объектом, определяя ожидаемое значение для столбцов атрибутов. При запуске таблицы FIT сравнивает ожидаемые значения с фактическими, вновь используя зеленый или красный цвет для указания результатов сравнения. Эта таблица следует за приведенной ранее императивной таблицей, так что вы получаете императивную таблицу (именуемую в FIT *action fixture*) для навигации по приложениям, за которой следует декларативная таблица ожидаемых результатов (именуемая в FIT *row fixture*) для сравнения с выводом приложения.

Такое использование таблиц в виде исходного кода достаточно необычно, но это подход, который можно было бы применять и почаще. Люди предпочитают использовать табличную форму, будь то примеры тестовых данных или более общие правила обработки, такие как таблица решений (*Decision Table* (485)). Многим экспертам предметных областей очень удобно использовать приложения электронных таблиц для редактирования своих таблиц, которые затем могут быть преобразованы в исходный текст.

Еще одна интересная особенность FIT в том, что это DSL, ориентированный на тестирование. В последние годы наблюдается рост интереса к автоматизированным средствам тестирования с DSL, созданными специально для организации тестов. Во многих из них чувствуется влияние FIT.

Тестирование является естественным выбором для DSL. По сравнению с языками программирования общего назначения, языки тестирования часто требуют различных видов структур и абстракций, таких как простая линейная императивная модель таблиц действий FIT. С тестами часто работают эксперты предметных областей, так что DSL — обычно хорошее решение, особенно если он написан для конкретного приложения.

10.7. Make и другие

Построить и выполнить простую программу — задача тривиальная, но не нужно иметь такой уж большой опыт в программировании, чтобы понять, что для построения сложного приложения требуется много шагов. Поэтому с первых дней Unix для решения подобных задач применялся инструмент Make (www.gnu.org/software/make). Основная проблема при построении приложений заключается в том, что многие его этапы являются весьма дорогостоящими, а их выполнение может не требоваться при каждой сборке. Поэтому естественным выбором программной модели является сеть зависимостей (*Dependency Network* (495)). Программа Make состоит из нескольких целей, связанных зависимостями.

```

edit : main.o kbd.o command.o display.o
      cc -o edit main.o kbd.o command.o
main.o : main.c defs.h
      cc -c main.c
kbd.o : kbd.c defs.h command.h
      cc -c kbd.c
command.o : command.c defs.h command.h
      cc -c command.c

```

В первой строке этой программы говорится, что `edit` зависит от других целей в программе; поэтому, если любая из них устарела, то после ее перестройки необходимо перестроить и `edit`. Сеть зависимостей позволяет свести время построения к минимуму, гарантируя при этом, что все, что должно быть построено, на самом деле будет построено. Make представляет собой хорошо знакомый внешний DSL.

Для меня самое интересное в языках построения типа Make — это не столько их вычислительные модели, как то, что в них DSL должен смещиваться с языками, более близкими к обычным языкам программирования. Наряду с указанием целей и зависимостей между ними (классический DSL-сценарий), вам также необходимо указать, как строится каждая из целей, что предполагает более императивный подход. В Make это делается с помощью сценариев командной оболочки; в приведенном примере это вызовы `cc` (компилиатора C).

В дополнение к смешению языков в определениях целей, когда построение становится более сложным, простой сети зависимостей недостаточно, и требуется построение дальнейших абстракций поверх нее. В мире Unix это привело к появлению Automake toolchain, в котором make-файлы генерируются системой Automake.

Аналогичный прогресс наблюдается и в мире Java. Стандартным языком построения Java является язык Ant, который также представляет собой внешний DSL с использованием XML в качестве носителя синтаксиса. (Это, несмотря на мое неприятие XML-носителей синтаксиса, позволяет избежать проблем Make, вызванных использованием символов табуляции и пробелов в синтаксических отступах.) Ant был задуман как очень простой язык, но дело закончилось встроенными сценариями общего назначения и другими системами наподобие Maven, генерирующими сценарии Ant.

Для своих личных проектов я предпочитаю использовать в качестве системы сборки Rake (<http://rake.rubyforge.org>). Как и в Make, и в Ant, здесь в качестве базовой вычислительной модели применяется сеть зависимостей. Существенным отличием является использование внутреннего DSL в Ruby. Это позволяет писать содержимое целей более естественным образом, а также облегчает построение больших абстракций.

Вот пример из `Rakefile` для сборки данной книги.

```

docbook_out_dir = build_dir + "docbook/"
docbook_book = docbook_out_dir + "book.docbook"

desc "Generate Docbook"
task :docbook => [:docbook_files, docbook_book]

file docbook_book => [":load"] do
  require 'docbookTr'
  create_docbook
end

def create_docbook
  puts "creating docbook"
  mkdir_p docbook_out_dir
  File.open(docbook_book, 'w') do |output|
    File.open('book.xml') do |input|

```

```
root = REXML::Document.new(input).root
dt = SingleDocbookBookTransformer.new(output,
    root, ServiceLocator.instance)
dt.run
end
end
end
```

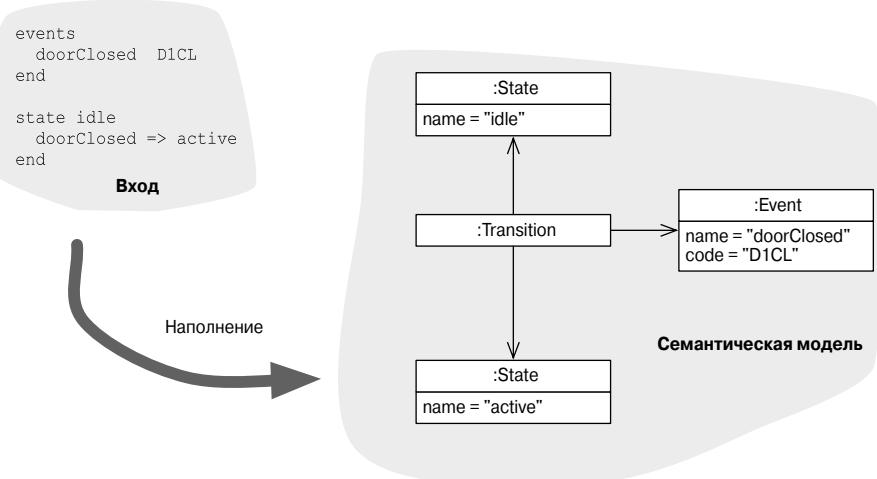
Строка `task :docbook => [:docbook_files, docbook_book]` представляет собой сеть зависимостей, гласящую, что `:docbook` зависит от двух других целей. Цели в Rake могут быть либо задачами, либо файлами (поддерживаются стили сети зависимостей, ориентированные как на задачи, так и на продукты). Императивный код для построения цели содержится во вложенном замыкании (*Nested Closure* (403)) после ее объявления. (Более подробная информация о том, что можно создать с помощью Rake, приведена в [14].)

Глава 11

Семантическая модель

Semantic Model

Модель, наполняемая DSL



11.1. Как это работает

В контексте DSL семантическая модель является представлением (таким, как объектная модель в памяти) субъекта описания DSL. Если DSL описывает состояние конечного автомата, то семантическая модель может быть объектной моделью с классами состояния, события и т.д. Сценарий DSL, определяющий конкретные состояния и события, будет соответствовать конкретному наполнению схемы с экземпляром события для каждого события, объявленного в сценарии DSL. Семантическая модель, таким образом, представляет собой библиотеку или платформу, заполняемую DSL.

В этой книге мои семантические модели представляют собой объектные модели, размещаемые в памяти, но есть и другие способы их представления. У вас может быть структура данных, а поведение конечного автомата при этом реализуется с помощью функций,

которые работают с этими данными. Модель не обязана располагаться в памяти; DSL может наполнить модель, хранящуюся, например, в реляционной базе данных.

Семантическая модель должна быть спроектирована с учетом предназначения DSL. В случае конечного автомата это предназначение — управление поведением с помощью вычислительной модели конечного автомата (*State Machine* (517)). Фактически семантическая модель должна использоваться и без DSL. Вы должны быть способны наполнить семантическую модель и посредством интерфейса командных запросов. Это гарантирует, что семантическая модель полностью охватывает семантику предметной области, а также допускает независимое тестирование модели и синтаксического анализатора.

Семантическая модель представляет собой понятие, очень похожее на *модель предметной области* (*Domain Model* [10]). Я употребляю отдельный термин, поскольку, хотя семантические модели часто являются подмножествами моделей предметной области, быть ими они не обязаны. Я использую термин “модель предметной области” для обозначения поведенчески богатой объектной модели, а семантическая модель может быть просто данными. Модель предметной области отражает базовое поведение приложения, в то время как семантическая модель может играть вспомогательную роль. Хорошим примером этого является объектно-реляционное отображение, которое координирует данные между объектной моделью и реляционной базой данных. Вы можете использовать DSL для описания объектно-реляционного отображения, и результирующая семантическая модель будет состоять из *отображателей данных* (*Data Mappers* [10]), а модель предметной области будет являться предметом отображения.

Семантические модели, как правило, отличаются от синтаксического дерева, потому что они служат для разных целей. Синтаксическое дерево соответствует структуре сценариев DSL. Хотя абстрактное синтаксическое дерево может упростить и несколько реорганизовать входные данные, оно принимает, по сути, один и тот же вид. Семантическая же модель основана на том, что будет сделано с информацией из сценария DSL. Часто это будут существенно отличающиеся структуры, и, как правило, не древовидные. Конечно, иногда абстрактное синтаксическое дерево может быть эффективной семантической моделью для DSL, но это скорее исключение, чем правило.

В традиционном рассмотрении языков и синтаксического анализа семантическая модель не используется. Это — часть различий между работой с DSL и с языками программирования общего назначения. Синтаксическое дерево обычно представляет собой подходящую структуру для генерации кода для языков общего назначения, что не совсем желательно для отдельной семантической модели. Конечно, время от времени используются и такие семантические модели, например представление в виде графа вызовов очень полезно для оптимизации. Такие модели называются промежуточными представлениями и обычно представляют собой промежуточные этапы перед генерацией кода.

Семантическая модель часто предшествует DSL. Это происходит, когда вы решаете, что часть модели предметной области может быть лучше наполнена с помощью DSL, чем с помощью обычного интерфейса командных запросов. Можно также строить DSL и семантическую модель одновременно, прибегая к дискуссиям с экспертами предметной области для уточнения как выражений DSL, так и структуры модели предметной области.

Семантическая модель может либо хранить код для самостоятельного его выполнения (стиль интерпретатора), либо быть основой для генерации кода (стиль компилятора). Даже если вы используете генерацию кода, полезно предоставить возможность интерпретации для упрощения тестирования и отладки.

Семантическая модель — это, как правило, наилучшее место для проверки поведения, поскольку у вас есть вся необходимая информация и структуры для выражения и за-

пуска проверок. В частности, весьма полезно выполнять такую проверку перед запуском интерпретатора или генерацией кода.

Брэд Кросс (Brad Cross) ввел понятия вычислительных и композиционных DSL [6]. Различие между ними связано в основном с видом производимых ими семантических моделей. Композиционные DSL описывают некоторый вид составных структур в текстовом виде. Хорошим примером может служить использование XAML для описания макета пользовательского интерфейса — основной формой этой семантической модели является то, как скомпонованы различные элементы. Пример конечного автомата в большей степени относится к вычислительным DSL, в том смысле, что его семантическая модель больше похожа на код, чем на данные.

Вычислительные DSL приводят к семантической модели, управляющей вычислениями, как правило, с альтернативной вычислительной моделью вместо обычной императивной. Обычно это адаптивная модель (*Adaptive Model* (478)). С помощью вычислительных DSL можно сделать гораздо больше, чем с помощью композиционных, но обычно пользователям сложнее с ними работать.

Имеет смысл рассматривать семантическую модель как имеющую два различных интерфейса. Один из них — **операционный интерфейс**, который позволяет клиентам в процессе работы использовать наполненную модель. Второй — **интерфейс наполнения**, который используется DSL для создания в модели экземпляров классов.

Операционный интерфейс должен, полагая семантическую модель уже созданной, обеспечить другим частям системы возможность легко ею воспользоваться. Я думаю, что при проектировании API неплохо применять мысленный эксперимент, который заключается в том, чтобы предположить, что модель волшебным образом уже создана, а затем спросить себя, как я предпочитаю ее использовать. Это может показаться парадоксальным, но я считаю, что лучше определить операционный интерфейс, прежде чем начать думать об интерфейсе наполнения, несмотря на то что, в первую очередь, система будет работать именно с интерфейсом наполнения. Для меня это общее правило при работе с любыми объектами, а не только с предметно-ориентированными языками.

Интерфейс наполнения используется только для создания экземпляров модели и может использоваться только синтаксическим анализатором (и тестовым кодом для семантической модели). Хотя мы стремимся как можно сильнее разделить семантическую модель и синтаксический анализатор (или анализаторы), между ними всегда есть зависимость — в том плане, что синтаксический анализатор для наполнения семантической модели должен быть о ней осведомлен. Несмотря на это путем создания понятного интерфейса можно уменьшить необходимость изменений в реализации семантической модели, которые, в свою очередь, требуют внесения изменений в синтаксический анализатор.

11.2. Когда это использовать

Мой совет по умолчанию — использовать семантическую модель всегда. Мне всегда неудобно, когда я говорю “всегда”, потому что обычно такие “абсолютные” советы представляются мне признаком консервативности мышления. В случае с семантической моделью я вижу лишь несколько ситуаций (или у меня недостаточное воображение?), в которых можно не хотеть использовать семантическую модель, и все это — очень простые ситуации.

Я считаю, что семантическая модель дает много убедительных преимуществ. Ясная семантическая модель позволяет тестировать семантику и синтаксический анализ DSL по-отдельности. Вы можете проверить семантику путем непосредственного наполнения семантической модели и проведения тестов с нею; и вы можете проверить синтаксический анализатор, определив, наполняет ли он семантическую модель корректными

объектами. Если у вас более одного синтаксического анализатора, можете проверить, производят ли они семантически эквивалентный выход, сравнив наполнения семантической модели. Все это облегчает поддержку нескольких DSL и, кроме того, позволяет разделять язык отдельно от семантической модели.

Семантическая модель повышает гибкость как синтаксического анализа, так и работы. Вы можете выполнять семантическую модель как непосредственно, так и путем генерации кода. В последнем случае семантическая модель может использоваться в качестве симулятора для сгенерированного кода. Семантическая модель также облегчает возможность иметь несколько генераторов кода, поскольку независимость от синтаксического анализатора позволяет избежать необходимости дублировать его код.

Но самым важным в использовании семантической модели является то, что она позволяет разделить размышления о семантике и о синтаксическом анализе. Даже простой DSL содержит достаточно сложностей, чтобы оправдать его разделение на две более простые задачи.

Так о каких исключениях я говорил? Одна такая ситуация — когда выполняется простая императивная интерпретация, и вы хотите просто выполнять каждую инструкцию по мере ее разбора. Хорошим примером может служить классический калькулятор, в котором вычисляются простые арифметические выражения. В случае арифметических выражений, даже если вы не интерпретируете их немедленно, абстрактное синтаксическое дерево практически совпадает с тем, что должно быть в семантической модели, так что никакого преимущества отделение синтаксического дерева от семантической модели для этого случая не дает. Вот пример более общего правила: если вы не задумываетесь о более полезной модели, чем абстрактное синтаксическое дерево, то в создании отдельной семантической модели очень мало смысла.

Наиболее распространенный случай отсутствия семантической модели — при генерации кода. При таком подходе синтаксический анализатор может генерировать абстрактное синтаксическое дерево, а генератор кода может работать непосредственно с ним. Это разумный подход при условии, что абстрактное синтаксическое дерево является хорошей моделью лежащей в основе семантики и вы не возражаете против связи логики генерации кода с абстрактным синтаксическим деревом. Если это не так, может оказаться проще преобразовать абстрактное синтаксическое дерево в семантическую модель и выполнить более простую генерацию кода на ее основе.

Тем не менее я всегда начинаю с предположения о том, что мне необходима семантическая модель. Даже если размышления убеждают меня, что она не нужна, я внимательно слежу за возрастающей сложностью синтаксического анализатора и ввожу семантическую модель, как только вижу, что синтаксический анализ становится достаточно сложным.

Несмотря на мое уважение к семантической модели, будет нечестно скрыть, что использование семантической модели не является частью культуры DSL в мире функционального программирования. Сообщество функционального программирования имеет долгую историю работы с DSL, и мой опыт работы с современными функциональными языками — не более чем случайный эксперимент. Так что, хотя мои наклонности и говорят мне, что семантическая модель будет полезна даже в этом мире, я должен признаться, что моих знаний недостаточно, чтобы быть в этом абсолютно уверенным.

11.3. Вводный пример (Java)

В этой книге есть много примеров семантических моделей, в первую очередь потому, что я их горячий сторонник. Достаточно полезной для иллюстративных целей является модель, которую я использовал во вводном примере — конечный автомат контроллера

системы безопасности. Здесь семантическая модель представляет собой модель конечного автомата. Я не употреблял термин “семантическая модель” при ее рассмотрении на начальном этапе, так как моей целью было введение понятия DSL. В результате я пришел к выводу, что легче полагать, что первой строится модель, а DSL надстраивается над ней. При этом модель по-прежнему остается семантической, но, поскольку теперь мы мыслим “наизнанку”, это не лучший подход для изучения.

Однако при этом сохраняются все классические преимущества семантической модели. Я могу выполнить (и я так и сделал) тестирование модели конечного автомата независимо от написания DSL. Я провел несколько рефакторингов реализации модели, не прикасаясь к коду синтаксического анализа, поскольку мои изменения реализации никак не отражались на интерфейсе наполнения. Даже если бы я действительно изменял эти методы, в большинстве случаев код синтаксического анализатора было бы легко модифицировать следующим образом, потому что интерфейс очень четко указывает границу между моделью и DSL.

Хотя поддержка нескольких DSL для одной и той же семантической модели не слишком распространена, это было требованием к моему примеру. Семантическая модель сравнительно легко справилась с этим заданием. Я получил несколько синтаксических анализаторов для внутренних и внешних DSL. Я мог бы протестировать их, проверяя, создают ли они одинаковое наполнение семантической модели. Я мог бы легко добавить новый DSL и синтаксический анализатор без дублирования кода в других синтаксических анализаторах и без какого-либо изменения семантической модели. Это преимущество работает и для получения выходных данных. В дополнение к непосредственной работе семантической модели в качестве конечного автомата я мог бы использовать ее для создания нескольких примеров генерации кода, а также для визуализации.

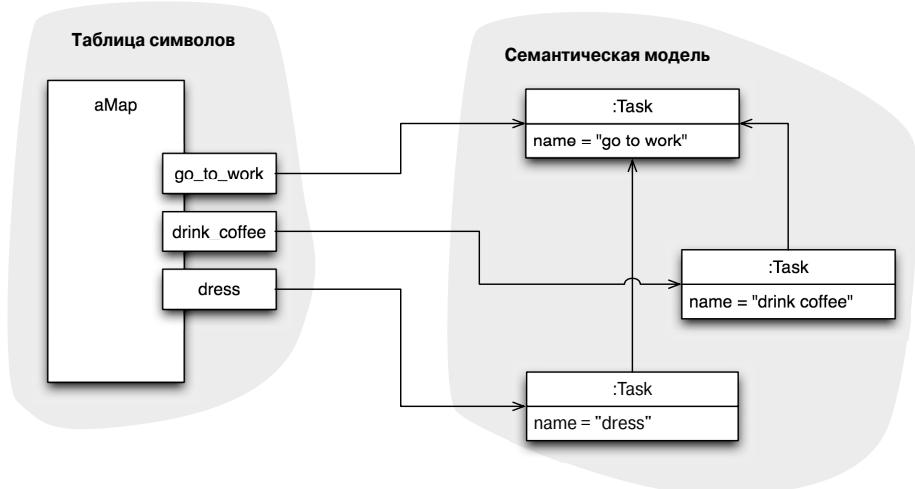
Помимо использования в качестве основы для выполнения и других выходов, семантическая модель является хорошим местом для выполнения проверок корректности. Я могу проверить, нет ли у меня недостижимых состояний или состояний, из которых невозможно выйти. Я также могу проверить, все ли события и команды используются в определениях состояний и переходов.

Глава 12

Таблица символов

Symbol Table

Место для хранения во время синтаксического анализа всех идентифицируемых объектов для разрешения ссылок



Многим языкам приходится ссылаться на объекты во многих точках кода. Если есть язык, который определяет конфигурацию задач и их зависимости, значит, нужен способ, которым одна задача могла бы сослаться на зависимые задачи в своем определении.

Для того чтобы сделать это, используем некоторую форму символов для каждой задачи; во время обработки сценария DSL эти символы помещаются в *таблицу символов*, которая хранит связь между символом и его базовым объектом, содержащим полную информацию.

12.1. Как это работает

Главное назначение таблицы символов — отобразить символ, используемый для обозначения объекта в сценарии DSL, и объект, на который этот символ ссылается. Это отношение естественным образом вписывается в понятие структуры данных отображения,

так что неудивительно, что наиболее распространенной реализацией таблицы символов является отображение с символом в качестве ключа и объектом семантической модели (*Semantic Model* (171)) в качестве значения.

Одним из рассматриваемых вопросов является тип объекта, который должен использоваться в качестве ключа в таблице символов. Для многих языков наиболее очевидным выбором является строка, поскольку текст DSL также является строкой.

В основном что-то отличное от строк используется в языках программирования, которые поддерживают символьный тип данных. Символы в смысле структуры подобны строкам (представляют собой последовательность знаков), но обычно отличаются поведением. Многие операции со строками (конкатенация, получение подстроки и т.д.) для символов не имеют смысла. Основная задача символов — выполнение поиска, и символьные типы данных обычно проектируются с учетом этого предназначения. Строки "foo" и "foo" зачастую представляют собой различные объекты и выяснение их равенства выполняется путем сравнения их содержимого. Символы же :foo и :foo всегда разрешаются в один и тот же объект, а их сравнение выполняется значительно быстрее.

Производительность может быть веским основанием для предпочтения символьных типов данных строкам, но для малых DSL разница может быть несущественной. Главная причина предпочтения символьного типа данных — он ясно выражает ваши намерения относительно его использования. Объявляя нечто, как символ, вы ясно указываете, для чего вы его используете, и, таким образом, ваш код становится более простым для понимания.

В языках, которые поддерживают символы, для них обычно используется особый лiteralный синтаксис. Ruby использует :aSymbol, Smalltalk — #aSymbol, а Lisp рассматривает любой отдельный идентификатор в качестве символа. Этим символы особенно выделяются во внутренних DSL, что является еще одной причиной их использования.

Значения в таблице символов могут быть либо объектами окончательной модели, либо промежуточными построителями. Объекты модели заставляют таблицу символов выступать в роли результирующих данных, что вполне подходит для простых ситуаций. Однако размещение в таблице в качестве значений построителей объектов часто обеспечивает большую гибкость за счет некоторого увеличения количества работы.

Многие языки имеют различные типы объектов, на которые требуется ссылаться. Модель конечного автомата из введения требует идентификации состояний, команд и событий. Наличие нескольких видов объектов, на которые нужно ссылаться, означает, что вам следует сделать выбор между отображением, множественными отображениями и специальным классом.

Использование для таблицы символов одного отображения означает, что поиск любого символа выполняется с помощью одного и того же отображения. Непосредственным следствием этого является то, что вы не можете использовать одно и то же имя символа для объектов разных видов; так, вы не можете иметь событие с тем же именем, что и состояние. Но это может быть полезным ограничением, уменьшающим путаницу в DSL. Однако использование одного отображения усложняет чтение кода обработки, поскольку при ссылке на символ становится менее понятно, с какого рода объектом приходится иметь дело. Поэтому я не рекомендую этот вариант.

При наличии нескольких отображений у вас имеется по одному отображению для каждого вида объектов, на которые вы ссылаетесь. Например, упомянутая модель может иметь три отображения — для событий, команд и состояний. Вы можете рассматривать их и как единую логическую таблицу символов, и как три отдельные таблицы символов. В любом случае я предпочитаю этот выбор одному отображению, так как теперь из кода становится ясно, с каким видом объектов вы имеете дело.

Использование специального класса означает наличие единого объекта для таблицы символов с различными методами для ссылки на хранящиеся объекты различных видов:

`getEvent(String code), getState(String code), registerEvent(String code, Event object)` и т.д. Иногда этот подход может быть полезен, и он обеспечивает естественное место для добавления любого специфичного поведения обработки символов. Однако по большей части я не вижу настоятельной необходимости к нему прибегать.

В некоторых случаях ссылки на объекты встречаются еще до их корректного определения. Это явление называется опережающими ссылками. У DSL, как правило, нет строгого правила, требующего объявления идентификатора до его использования, так что опережающие ссылки зачастую имеют смысл. Допуская в своем предметно-ориентированном языке применение опережающих ссылок, необходимо убедиться в том, что каждая ссылка на символ приводит к заполнению записи в таблице символов, если его там еще нет. Это часто подталкивает к использованию в качестве значений таблицы символов построителей, если только объекты модели не являются очень гибкими.

Если явное объявление символов отсутствует, будьте очень осторожны с ошибочно записанными символами, которые могут стать серьезным источником ошибок. У вас может иметься некоторый способ обнаружения неправильно записанных символов; применив его, можно сберечь много нервов и разбитых в гневе мониторов. Эта проблема является одной из причин для настоятельного требования того или иного объявления символов в языке. Если вы решите прибегнуть к такому требованию, помните о том, что символы необязательно должны быть объявлены перед использованием.

В более сложных языках часто имеются вложенные области видимости, когда символы определены только в некотором подмножестве полной программы. Это очень распространенная практика в языках общего назначения, но в более простых DSL это встречается гораздо реже. Если же вы все-таки хотите сделать это в своем предметно-ориентированном языке, воспользуйтесь *таблицами символов для вложенных областей видимости* [23].

12.1.1. Статически типизированные символы

Создавая внутренний DSL в статически типизированном языке, таком как C# или Java, вы можете легко использовать в качестве таблицы символов неупорядоченное отображение (хэшированный ассоциативный массив, hashmap) со строками в качестве ключей. Стока такого DSL может иметь примерно такой вид.

```
task("drinkCoffee").dependsOn("make_coffee", "wash");
```

Такое применение строк, безусловно, будет работать, но оно обладает некоторыми недостатками.

- Строки вносят синтаксический шум, так как их необходимо заключать в кавычки.
- Компилятор не в состоянии выполнить проверку типов. Если вы допустите опечатку при вводе имени, это обнаружится только во время выполнения. Кроме того, если у вас есть различные виды идентифицируемых объектов, компилятор не сможет обнаружить, что вы ссылаетесь на неверный тип, и это будет обнаружено только во время выполнения.
- Современные интегрированные среды не смогут выполнять автозаполнение строк. Это означает, что вы теряете мощный элемент программной помощи.
- Автоматизированный рефакторинг может не работать со строками.

Этих проблем можно избежать с помощью некоторого вида статически типизированных символов. Хорошим выбором могут оказаться перечисления, равно как и класс таблицы символов (Class Symbol Table (461)).

12.2. Когда это использовать

Применение таблиц символов — общепринятая практика при любой работе с языком, и я думаю, что вы почти всегда будете в них нуждаться.

Иногда они не являются строго необходимыми. При построении дерева (Tree Construction (289)) вы всегда можете углубиться в синтаксическое дерево, чтобы найти нужный объект. Часто эту работу можно выполнить путем поиска по создаваемой семантической модели (Semantic Model (171)). Но иногда вам может потребоваться некоторое промежуточное хранилище (но даже если оно не потребуется, оно часто позволяет облегчить жизнь).

12.3. Дополнительная литература

В [23] приведена детальная информация об использовании различных видов таблиц символов для внешних DSL. Так как таблица символов — важная технология и для внутренних DSL, многие из описанных подходов, скорее всего, окажутся приемлемыми и для внутренних DSL.

В [18] содержится ряд полезных идей по использованию статически типизированных символов в Java. Как обычно, эти идеи часто применимы и в других языках.

12.4. Сеть зависимостей во внешнем DSL (Java и ANTLR)

Вот простая сеть зависимостей.

```
go_to_work -> drink_coffee dress
drink_coffee -> make_coffee wash
dress -> wash
```

Задача слева от " ->" зависит от задач, перечисленных справа. Я выполняю синтаксический анализ с помощью встроенной трансляции (Embedded Translation (305)). Я хочу иметь возможность записывать зависимости в любом порядке и получать списки заголовков, т.е. задач, которые не являются предварительным условием для решения любых других задач. Это хороший пример, в котором для отслеживания задач целесообразно применять таблицы символов.

По привычке я пишу класс-загрузчик для обертки синтаксического анализатора ANTLR; он будет получать входные данные от читателя.

```
class TaskLoader...
private Reader input;
public TaskLoader(Reader input) {
    this.input = input;
}

public void run() {
    try {
        TasksLexer lexer =
            new TasksLexer(new ANTLRReaderStream(input));
        TasksParser parser =
            new TasksParser(new CommonTokenStream(lexer));
        parser.helper = this;
        parser.network();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
```

```
        } catch (RecognitionException e) {
            throw new RuntimeException(e);
        }
    }
```

Загрузчик сам себя помещает в качестве встроенного помощника (*Embedment Helper* (537)) в сгенерированный синтаксический анализатор. Одной из полезных возможностей, которые он обеспечивает, является таблица символов, представляющая собой простое отображение имен задач на задачи.

```
class TaskLoader...
private Map<String, Task> tasks =
    new HashMap<String, Task>();
```

Грамматика этого DSL исключительно проста.

```
grammar file...
network : SEP? dependency (SEP dependency)* SEP?;
dependency
    : lhs=ID '-'> rhs+=ID+
        {helper.recognizedDependency($lhs, $rhs);}
    ;
```

В помощнике содержится код, который обрабатывает распознанную зависимость. Для связывания задач он заполняет таблицу символов.

```
class TaskLoader...
public void recognizedDependency(Token consequent,
                                  List dependencies) {
    registerTask(consequent.getText());
    Task consequentTask = tasks.get(consequent.getText());
    for(Object o : dependencies) {
        String taskName = ((Token)o).getText();
        registerTask(taskName);
        consequentTask.addPrerequisite(tasks.get(taskName));
    }
}

private void registerTask(String name) {
    if (!tasks.containsKey(name)) {
        tasks.put(name, new Task(name));
    }
}
```

После запуска загрузчик можно запросить о заголовках.

```
class TaskLoader...
public List<Task> getResult() {
    List<Task> result = new ArrayList<Task>();
    for(Task t : tasks.values())
        if (!tasksUsedAsPrerequisites().contains(t))
            result.add(t);
    return result;
}

public Set<Task> tasksUsedAsPrerequisites() {
    Set<Task> result = new HashSet<Task>();
    for(Task t : tasks.values())
        for (Task preReq : t.getPrerequisites())
            result.add(preReq);
    return result;
}
```

12.5. Использование символьных ключей во внутреннем DSL (Ruby)

Таблицы символов пришли из мира синтаксического анализа, но они полезны и для внутренних DSL. В этом примере использован Ruby, чтобы показать применение символьного типа данных. Вот простой сценарий DSL с выполняемыми утром задачами и их предусловиями.

```
task :go_to_work => [:drink_coffee, :dress]
task :drink_coffee => [:make_coffee, :wash]
task :dress => [:wash]
```

Ссылки на задачи в DSL осуществляются с помощью символьного типа данных Ruby. Я использую последовательность функций (*Function Sequence* (357)) для объявления списка задач, а детальная информация о каждой задаче указывается с помощью литералов (*Literal Map* (417)).

Описать семантическую модель (*Semantic Model* (171)) очень просто — с помощью одного класса задачи.

```
class Task
  attr_reader :name
  attr_accessor :prerequisites

  def initialize name, *prereqs
    @name = name
    @prerequisites = prereqs
  end

  def to_s
    name
  end
end
```

Сценарий DSL считывается построителем выражений (*Expression Builder* (349)), который использует перенос в область видимости объекта (*Object Scoping* (387)) с `instance_eval`.

```
class TaskBuilder...
  def load aStream
    instance_eval aStream
    return self
  end
```

Таблица символов представляет собой простой словарь.

```
class TaskBuilder...
  def initialize
    @tasks = {}
  end
```

Вызов `task` принимает один аргумент (хеш-ассоциацию) и использует его для заполнения информации о задачах.

```
class TaskBuilder...
  def task argMap
    raise "syntax error" if argMap.keys.size != 1
    key = argMap.keys[0]
    newTask = obtain_task(key)
    prereqs = argMap[key].map{|s| obtain_task(s)}
    newTask.prerequisites = prereqs
```

```

end
def obtain task aSymbol
  @tasks[aSymbol] = Task.new(aSymbol.to_s) unless @tasks[aSymbol]
  return @tasks[aSymbol]
end

```

Реализация таблицы символов с помощью символов для идентификаторов такая же, как и с помощью строк. Однако, если символы доступны, необходимо использовать именно их.

12.6. Использование перечислений для статически типизированных символов (Java)

Майкл Хангер (Michael Hunger) был очень прилежным рецензентом этой книги и постоянно убеждал меня в необходимости описать использование перечислений как статически типизированных символов, поскольку он сам с успехом применял этот подход. Многим статическая типизация нравится возможностью находить ошибки. Я не столь полон энтузиазма, поскольку чувствую, что статическая типизация выявляет мало ошибок, которые не могут быть обнаружены при серьезном тестировании (и которое все равно необходимо — со статической типизацией или без нее). Но одно большое преимущество статической типизации заключается в том, что она хорошо работает с современными интегрированными средами разработки. Приятно просто нажать <Ctrl+Space> и получить список всех символов, которые могут использоваться в этой точке программы.

Я еще раз воспользуюсь примером с задачами с той же семантической моделью (*Semantic Model (171)*), что и ранее. Семантическая модель применяет строки для именования задач, но в DSL я буду использовать перечисления. Это не только предоставит мне возможность автозаполнения в интегрированной среде разработки, но и защитит от опечаток. Перечисление просто.

```

public enum TaskName {
  wash, dress, make_coffee, drink_coffee, go_to_work
}

```

Я могу использовать его для определения зависимостей задач следующим образом.

```

builder = new TaskBuilder() {{
  task(wash);
  task(dress).needs(wash);
  task(make_coffee);
  task(drink_coffee).needs(make_coffee, wash);
  task(go_to_work).needs(drink_coffee, dress);
}}

```

Здесь я применяю инициализатор экземпляра Java для переноса в область видимости объекта (*Object Scoping (387)*). Я также использую статический импорт для перечисления имен задач, что позволяет мне употреблять в сценарии неквалифицированные имена задач. С помощью этих двух методов я в состоянии написать сценарий в любом классе без необходимости использовать наследование (как заставило бы меня поступить применение построителя выражений (*Expression Builder (349)*)).

Построитель задачи строит отображение задач; каждый вызов `task` регистрирует задачу в отображении.

```

class TaskBuilder...
PrerequisiteClause task(TaskName name) {
  registerTask(name);
}

```

```

        return new PrerequisiteClause(this, tasks.get(name));
    }
    private void registerTask(TaskName name) {
        if (!tasks.containsKey(name)) {
            tasks.put(name, new Task(name.name()));
        }
    }
    private Map<TaskName, Task> tasks =
        new EnumMap<TaskName, Task>(TaskName.class);
}

```

PrerequisiteClause представляет собой дочерний класс построителя.

```

class PrerequisiteClause...
private final TaskBuilder parent;
private final Task consequent;

PrerequisiteClause(TaskBuilder parent, Task consequent) {
    this.parent = parent;
    this.consequent = consequent;
}
void needs(TaskName... prereqEnums) {
    for (TaskName n : prereqEnums) {
        parent.registerTask(n);
        consequent.addPrerequisite(parent.tasks.get(n));
    }
}

```

Я сделал дочерний построитель статическим внутренним классом построителя задачи, поэтому он может получить доступ к закрытым членам последнего. Я мог бы пойти дальше и сделать его экземпляром внутреннего класса; тогда я не нуждался бы в ссылке на родителя. Я этого не сделал, так как это могло бы усложнить слежение за кодом для читателей, плохо знающих Java.

Использовать перечисления, как здесь, довольно просто, и хорошо, что это не вынуждает нас прибегать к наследованию и не ограничивает в том, где писать код сценария DSL, что является преимуществом по сравнению с классом таблицы символов (Class Symbol Table (461)).

При таком подходе следует иметь в виду, что если набор символов должен соответствовать некоторому внешнему источнику данных, вы можете написать дополнительный шаг, который считывает внешний источник данных и генерирует код объявления перечисления, так что все поддерживается в синхронизированном состоянии [18].

Следствием этой реализации является единое пространство имен символов. Это прекрасно, когда у вас есть несколько маленьких сценариев, которые разделяют тот же набор символов, но иногда разные сценарии должны иметь разные наборы символов.

Предположим, есть два набора задач, один из которых посвящен утренним делам (читайте выше), а другой — уборке снега (я только что выглянул в окно...). Работая над утренними задачами, я хочу, чтобы интегрированная среда разработки предлагала мне только связанную с ними помочь и чтобы то же самое было с задачами по уборке снега.

Я могу добиться этого, определив свой построитель задачи в терминах интерфейса, который реализует мое перечисление.

```

public interface TaskName {}

class TaskBuilder...
PrerequisiteClause task(TaskName name) {
    registerTask(name);
    return new PrerequisiteClause(this, tasks.get(name));
}

```

```
private void registerTask(TaskName name) {
    if (!tasks.containsKey(name)) {
        tasks.put(name, new Task(name.toString()));
    }
}
private Map<TaskName, Task> tasks =
    new HashMap<TaskName, Task>();
```

Затем я могу определить некоторые перечисления и использовать их для конкретной группы задач путем выборочного импорта того, что мне нужно.

```
import static path.to.ShovelTasks.*;

enum ShovelTasks implements TaskName {
    shovel_path, shovel_drive,
    shovel_sidewalk, make_hot_chocolate
}

builder = new TaskBuilder() {{
    task(shovel_path);
    task(shovel_drive).needs(shovel_path);
    task(shovel_sidewalk);
    task(make_hot_chocolate).needs(shovel_drive,
                                    shovel_sidewalk);
}};
```

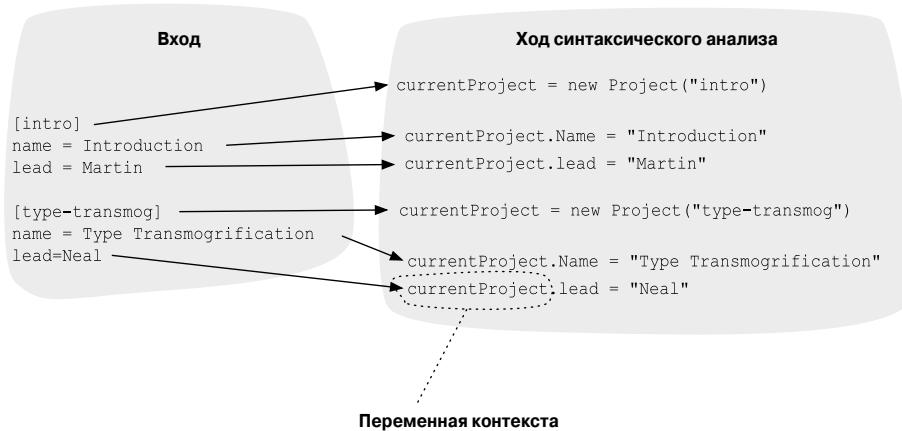
Для того чтобы получить возможность еще больше управлять статическими типами, я мог бы создать обобщенную версию построителя задач, который бы проверял, использует ли он правильный подтип TaskName. Но для использования возможностей интегрированной среды разработки вполне достаточно выборочного импорта правильного набора перечислений.

Глава 13

Переменная контекста

Context Variable

Для хранения необходимого во время синтаксического анализа контекста используется переменная



Представьте себе, что вы анализируете список элементов на основании данных о каждом из них. Каждую часть информации об элементе можно получить независимо, но для этого нужно знать, к какому именно элементу она относится.

Метод переменной контекста делает это путем хранения текущего элемента в переменной и изменения ее значения при переходе к новому элементу.

13.1. Как это работает

Если у вас есть переменная с именем наподобие `currentItem`, которую вы периодически обновляете во время синтаксического анализа при переходе от одного элемента входного сценария к другому, значит, у вас есть переменная контекста.

Переменная контекста может быть объектом семантической модели (**Semantic Model (171)**) или построителем. Семантическая модель внешне более проста, но это верно, только если все ее свойства модифицируемы, когда синтаксическому анализу требуется

их изменить. Если это не так, то, как правило, лучше использовать некоторый вид построителя для сбора информации и создания объекта семантической модели по его окончании — что-то наподобие построителя конструкций (Construction Builder (191)).

13.2. Когда это использовать

Во многих ситуациях в процессе синтаксического анализа необходимо хранить контекст, и здесь очевидным выбором являются переменные контекста. Их легко создавать, и с ними просто работать.

Однако, поработав с переменными контекста, вы поймете, что они вызывают ряд проблем. По своей природе они представляют собой изменяемые состояния, за которыми необходимо внимательно следить. Можно очень легко забыть обновить переменную контекста в нужный момент, а отладка может превратиться в сложную задачу. Обычно имеются альтернативные способы организации синтаксического анализа, позволяющие избежать применения переменных контекста. Хотя я и не утверждаю, что применять переменные контекста нежелательно, я все же предпочитаю использовать методы, которые в них не нуждаются.

13.3. Чтение INI-файла (C#)

Я хотел привести пример использования переменной контекста попроще, и старый формат INI-файлов оказался для этого подходящим. Хотя он и может показаться несколько старомодным — он был “усовершенствован” реестром Windows, — он по-прежнему остается легким и удобочитаемым средством работы с простым списком элементов со свойствами. Альтернативные форматы наподобие XML и YAML могут обрабатывать и более сложные структуры, но за счет усложнения чтения и анализа. Если ваши потребности достаточно просты для применения INI-файла, он остается разумным выбором.

Ниже перечислены коды проектов и некоторые их свойства для моего примера.

```
[intro]
name = Introduction
lead = Martin

[type-transmogrification]
name = Type Transmogrification
lead=Neal
# Стока комментария

[lang] # Комментарий группы
name = Language Background Advice
lead = Rebecca # Комментарий элемента
```

Хотя стандарта для формата INI нет, основными его элементами являются присвоения свойств, разбитые на разделы. В данном случае каждый раздел представляет собой код проекта.

Семантическая модель (Semantic Model (171)) тривиальна.

```
class Project...
    public string Code { get; set; }
    public string Name { get; set; }
    public string Lead { get; set; }
```

Формат INI-файла легко прочесть с помощью трансляции, управляемой разделителями (Delimiter-Directed Translation (213)). Базовая структура анализатора реализует обычный подход-разбиение сценария на строки и анализ каждой из них.

```

class ProjectParser...
    private TextReader input;
    private List<Project> result = new List<Project>();

    public ProjectParser(TextReader input) {
        this.input = input;
    }
    public List<Project> Run() {
        string line;
        while ((line = input.ReadLine()) != null) {
            parseLine(line);
        }
        return result;
    }
}

```

Первые инструкции анализатора строк обрабатывают пустые строки и комментарии.

```

class ProjectParser...
    private void parseLine(string s) {
        var line = removeComments(s);
        if (isBlank(line)) return ;
        else if (isSection(line)) parseSection(line);
        else if (isProperty(line)) parseProperty(line);
        else throw new ArgumentException("Unable to parse: " + line);
    }
    private string removeComments(string s) {
        return s.Split('#')[0];
    }
    private bool isBlank(string line) {
        return Regex.IsMatch(line, @"^\s*$");
    }
}

```

Переменная контекста — `currentProject` — появляется в ходе анализа разделов; именно здесь ей присваивается соответствующее значение.

```

class ProjectParser...
    private bool isSection(string line) {
        return Regex.IsMatch(line, @"^\s*\[");
    }
    private void parseSection(string line) {
        var code =
            new Regex(@"\[.*\]").Match(line).Groups[1].Value;
        currentProject = new Project {Code = code};
        result.Add(currentProject);
    }
    private Project currentProject;
}

```

Затем она используется в процессе анализа свойства.

```

class ProjectParser...
    private bool isProperty(string line) {
        return Regex.IsMatch(line, @"^=");
    }
    private void parseProperty(string line) {
        var tokens = extractPropertyTokens(line);
        setProjectProperty(tokens[0], tokens[1]);
    }
    private string[] extractPropertyTokens(string line) {
        char[] sep = {'='};
        var tokens = line.Split(sep, 2);
        if (tokens.Length < 2)
            throw new ArgumentException("unable to split");
    }
}

```

190 Часть II. Общие вопросы

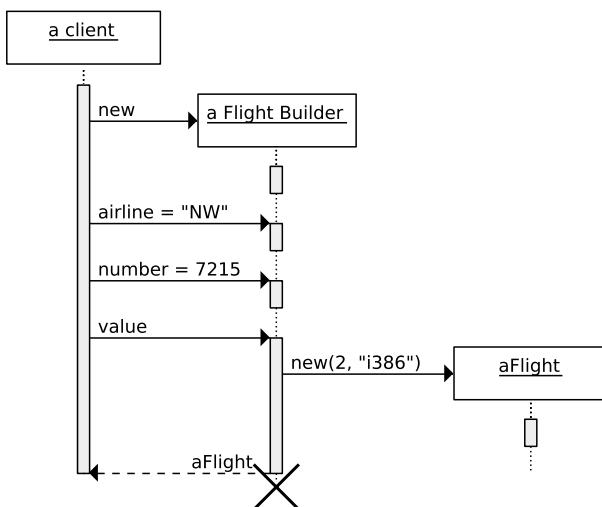
```
for (var i = 0; i < tokens.Length; i++)
    tokens[i] = tokens[i].Trim();
return tokens;
}
private void setProjectProperty(string name, string value) {
    var proj = typeof(Project);
    var prop = proj.GetProperty(capitalize(name));
    if (prop == null)
        throw new ArgumentException("Unable to find property: "
            + name);
    prop.SetValue(currentProject, value, null);
}
private string capitalize(string s) {
    return s.Substring(0, 1).ToUpper() +
        s.Substring(1).ToLower();
}
```

Применение отражения C# делает код более сложным, но зато при добавлении свойств в семантическую модель не нужно обновлять анализатор.

Глава 14

Построитель конструкции Construction Builder

Последовательно создает неизменяемый объект с помощью построителя, который сохраняет аргументы конструктора в полях



14.1. Как это работает

Принцип работы построителя конструкций очень прост. Пусть нужно поэтапно создать неизменяемый объект (который я буду называть продуктом). Возьмите аргументы конструктора продукта и создайте поле для каждого из них. Добавьте дополнительные поля для любых других атрибутов продукта, который вы создаете. Наконец добавьте метод для создания и возврата нового объекта продукта, собранного из всех данных в построителе конструкции.

Можете добавить некоторые элементы управления жизненным циклом построителя конструкции. Такой элемент может, например, проверять, достаточно ли накопленной информации для создания продукта. Вы можете по выдаче продукта устанавливать флаг, который предотвратит его повторную генерацию, или хранить созданный продукт в поле.

При попытке добавить новые атрибуты в построитель, уже создавший продукт, можно генерировать ошибку.

Несколько построителей конструкций могут быть объединены в более глубокие структуры. После этого они смогут производить не единственный объект, а группы связанных объектов.

14.2. Когда это использовать

Построитель конструкции полезен, когда необходимо создать объект с несколькими неизменяемыми полями, но значения этих полей собираются поэтапно. Построитель конструкции обеспечивает место для хранения всех этих данных, пока продукт не будет создан фактически.

Самой простой альтернативой построителю конструкции является сохранение информации в локальных переменных или в несвязанных полях. Это прекрасно работает для одного или двух продуктов, но становится очень запутанным, если необходимо одновременно создать кучу объектов, например, в процессе синтаксического анализа.

Еще одной альтернативой является создание реального объекта модели; после сбора данных для неизменяемого атрибута создается копия объекта модели с измененным атрибутом, заменяющая старый объект. Так можно избежать создания построителя конструкции, но этот подход, как правило, более неудобен в написании и работе. В частности, он не работает при наличии нескольких ссылок на объект, или по крайней мере его применение становится более трудным, так как при этом необходимо заменить все ссылки.

Обычно наилучший способ справиться с такой проблемой — использовать построитель конструкций, но помните, что он нужен только тогда, когда имеются неизменяемые поля. Если это не так, то просто создавайте свои объекты продукта непосредственно.

Несмотря на общее слово “построитель”, этот шаблон представляется мне существенно отличающимся от построителя выражений (*Expression Builder* (349)). Построитель конструкций просто накапливает аргументы конструктора и не пытается обеспечить свободный интерфейс. Построитель выражений, напротив, сосредоточен на обеспечении свободного интерфейса. Конечно, не так уж необычны ситуации, когда один и тот же объект является и построителем конструкции, и построителем выражений, но это не означает, что они представляют собой одну и ту же концепцию.

14.3. Построение простых полетных данных (C#)

Представьте себе приложение, которое использует некоторые данные о полетах. Данные этим приложением только считываются, так что имеет смысл сделать эти классы доступными только для чтения.

```
class Flight...
{
    readonly int number;
    readonly string airline;
    readonly IList<Leg> legs;
    public Flight(string airline, int number, List<Leg> legs) {
        this.number = number;
        this.airline = airline;
        this.legs = legs.AsReadOnly();
    }
    public int Number {get { return number; }}
    public string Airline {get { return airline; }}
    public IList<Leg> Legs {get { return legs; }}
}
```

```
class Leg...
    readonly string start, end;
    public Leg(string start, string end) {
        this.start = start;
        this.end = end;
    }
    public string Start {get { return start; }}
    public string End {get { return end; }}
```

Хотя приложение может только читать полетные данные, вполне возможно, что оно собирает информацию таким образом, что использовать конструкторы для создания полностью сформированных объектов очень сложно. В таких ситуациях простой построитель конструкции позволяет постепенно собирать все необходимые данные и построить конечный объект.

```
class FlightBuilder...
    public int Number { get; set; }
    public string Airline { get; set; }
    public List<LegBuilder> Legs { get; private set; }
    public FlightBuilder() {
        Legs = new List<LegBuilder>();
    }
    public Flight Value {
        get{return new Flight(Airline, Number,
                             Legs.ConvertAll(l => l.Value));}
    }
}

class LegBuilder...
    public string Start { get; set; }
    public string End { get; set; }

    public Leg Value {
        get { return new Leg(Start, End); }
    }
```


Глава 15

Макрос

Macros

Преобразует входной текст в другой текст до языковой обработки с помощью шаблонной генерации (Templated Generation (529))

Определение

```
#define max(x,y) x > y ? x : y
```

Вызов

```
int a = 5, b = 7, c = 0;  
c = max(a,b);
```

Раскрытие

```
c = 5 > 7 ? 5 : 7
```

Язык имеет фиксированный набор форм и структуры, которые он может обработать. В некоторых случаях можно добавить в язык абстракцию путем манипуляций входным текстом с помощью исключительно текстовых преобразований, прежде чем этот текст будет обработан компилятором или интерпретатором языка.

Макросы позволяют определить эти преобразования либо в текстовом виде, либо в виде синтаксических макроопределений, понимающих синтаксис базового языка.

15.1. Как это работает

Макросы являются одним из старейших средств построения абстракций в языках программирования. На заре программирования макросы были распространены так же широко, как и функции. С тех пор они в значительной степени впали в немилость, в основном по уважительным причинам. Тем не менее есть еще места, где они появляются во внутренних DSL, особенно в сообществе Lisp.

Я хотел бы разделить макросы на две основные разновидности: текстовые и синтаксические. Текстовые макросы более знакомы и просты в понимании — они рассматривают текст как просто текст. Синтаксические макросы осведомлены о синтаксической структуре базового языка; таким образом, в этом случае проще обеспечить работу с синтаксически осмыслившими единицами текста и получить синтаксически корректные результаты. Текстовый макропроцессор может работать с любым языком, представимым в виде текста, т.е. почти с любым языком программирования. Синтаксический макропроцессор предназначен для работы только с одним языком; такие макропроцессоры часто включены в инструменты для конкретного языка и даже описаны в спецификации самого языка.

Чтобы понять, как работают макросы, по моему мнению, лучше начать с текстовых макросов, несмотря на то что синтаксические макросы более интересны.

15.1.1. Текстовые макросы

Большинство современных языков не поддерживает текстовые макросы, и большинство разработчиков их избегают. Однако вы можете применять текстовые макросы с любым языком, используя обобщенные макропроцессоры наподобие классического макропроцессора m4 в Unix. Для некоторых технологий могут использоваться процессоры шаблонов, такие как Velocity, которые представляют собой очень простые макропроцессоры. И хотя большинство современных языков избегают макросов, C (а следовательно, и C++) имеет встроенный в основные инструменты препроцессор. Обычно эксперты в области C++ настойчиво советуют избегать (по достаточно уважительным причинам) использования препроцессора, но он все еще включен в язык.

Простейшей формой обработки макросов является замена одной строки другой. Хорошим примером является устранение дублирования при указании цвета в CSS-документах. Например, у вас есть сайт и есть определенные многократно используемые цвета — в границах таблиц, для подсветки текста и т.д. В обычном CSS вам придется повторять цветовой код каждый раз, когда вы его используете.

```
div.leftbox { border-bottom-color: #FFB595 }
p.head { bgcolor: #FFB595 }
```

Такое дублирование затрудняет обновление цвета, а числовое значение усложняет понимание. С макропроцессором можно определить специальное слово для своего цвета и использовать его вместо числового значения.

```
div.leftbox { border-bottom-color: MEDIUM_SHADE }
p.head { bgcolor: MEDIUM_SHADE }
```

По сути, макропроцессор проходит через CSS-файл и заменяет все встреченные MEDIUM_SHADE значением цвета, в результате чего получается тот же текст, что и в приведенном выше примере. Измененный CSS-файл не является корректным кодом CSS, — этот язык не имеет возможности определять символические константы, так что макропроцессор, в сущности, усовершенствовал язык CSS.

В этом примере подстановку можно было бы сделать с помощью простого поиска и замены во входном тексте, используя шлифовку текста (*Textual Polishing* (469)). Хотя замена текста поразительно проста, это распространенное применение макросов в программировании на C, специально для создания символьических констант. Вы можете использовать тот же механизм для добавления общих элементов в файлы, например общих для всех веб-страниц верхних и нижних колонтитулов. Определите маркер в файле, из которого будет получен настоящий HTML, выполните замещения и получите файл HTML.

Этот простой прием удивительно удобен для небольших сайтов, на которых нужно создать общие верхние и нижние колонтитулы без их дублирования на каждой странице.

Более интересными текстовыми макросами являются макросы, допускающие параметризацию. Предположим, что вам часто приходится определять максимальное из двух чисел, так что вы неоднократно пишете следующее выражение на языке программирования C: `a > b ? a : b`. Вы можете записать его в виде макроса для препроцессора C.

```
#define max(x,y) x > y ? x : y

int a = 5, b = 7, c = 0;
c = max(a,b);
```

Разница между макросом и вызовом функции в том, что макрос вычисляется во время компиляции. При этом выполняется замена выражения `max` с подстановкой переданных ему аргументов. Компилятор никогда не увидит выражение с `max`.

(Нужно сказать, что в некоторых средах термин “макрос” используется для подпрограмм. Это раздражает, но такова жизнь...)

Итак, макросы представляют собой альтернативу вызову функции. Преимущество использования макроса — в отсутствии накладных расходов на вызов функции, которые часто беспокоили программистов на C, особенно на раннем этапе развития вычислительной техники. Недостатки же макросов состоят в том, что у них есть много тонких моментов, особенно при использовании параметров. Рассмотрим макрос для возведения числа в квадрат.

```
#define sqr(x) x * x
```

Выглядит просто и должен отлично работать. Но попытайтесь вызвать его таким образом.

```
int a = 5, b = 1, c = 0;
c = sqr(a + b);
```

В этом случае значение `c` равно 11. Дело в том, что макроподстановка дает выражение `a + b * a + b`. Поскольку приоритет умножения выше приоритета сложения, вместо `(a + b) * (a + b)` получается `a + (b * a) + b`. Это один из примеров, когда раскрытие макроса дает нечто совершенно иное, чем то, чего ожидал программист, поэтому я называю его **ошибочным раскрытием** макроса. Такие раскрытия макросов могут корректно работать почти все время, но в отдельных случаях возникают удивительные и очень трудно обнаруживаемые ошибки.

В данном случае можно избежать ошибки, если расставить скобки чаще, чем программист на Lisp.

```
#define betterSqr(x) ((x) * (x))
```

Синтаксические макросы избегают большинства такого рода ловушек в силу знания базового языка программирования. Однако имеются и другие проблемы. Сначала я проиллюстрирую их на текстовых макросах.

Вернемся к примеру с `max`.

```
#define max(x,y) x > y ? x : y

int a = 5, b = 1, c = 0;
c = max(++a, ++b);
printf("%d", c); // => 7
```

Это пример **многократного вычисления**, когда передается аргумент с побочным действием, а тело макроса упоминает этот аргумент несколько раз и, таким образом, вычисляет его

несколько раз. В данном случае и *a*, и *b* увеличиваются два раза. Это хороший пример ошибки, которую очень трудно найти. Особенно огорчает то, что очень трудно найти пути неверного выполнения макроподстановок. Здесь все делается не так, как при вызове функций, и сложности становятся еще большими при наличии вложенных макросов.

Рассмотрим еще один макрос. Он принимает три аргумента: массив из пяти целых чисел, предельное значение и место для записи результата. Сначала выполняется суммирование чисел в массиве, после чего в результирующую переменную помещается либо сумма, либо верхний предел — в зависимости от результата их сравнения.

```
#define cappedTotal(input, cap, result) \
{int i, total = 0; \
for(i=0; i < 5; i++) \
    total = total + input[i]; \
result = (total > cap) ? cap : total;}
```

Макрос вызывается, например, так.

```
int arr1[5] = {1,2,3,4,5};  
int amount = 0;  
cappedTotal (arr1, 10, amount);
```

Все, казалось бы, неплохо работает (не учитывая того факта, что было бы лучше, если бы это была функция). А теперь взглянем на следующий фрагмент.

```
int total = 0;  
cappedTotal (arr1, 10, total);
```

После выполнения кода значение *total* равно 0. Проблема в том, что имя *total* подставляется при раскрытии макроса, но интерпретируется макросом как переменная, определенная в самом макросе. В результате передаваемая в макрос переменная игнорируется — такая ошибка называется **захватом переменной**.

Существует и обратная проблема, которой нет в C, но которая есть в языках, не требующих объявления переменных. Чтобы проиллюстрировать ее, я напишу некоторые текстовые макросы в Ruby. Это упражнение слишком бессмысленно даже по меркам примеров для книги. В качестве макропроцессора будем использовать Velocity — довольно известный инструмент для генерации веб-страниц. Velocity имеет возможность работать с макросами, к которой я и прибегну для иллюстрации сказанного.

Воспользуемся примером *cappedTotal* еще раз. Вот макрос Velocity для кода Ruby.

```
#macro(cappedTotal $input $cap $result)  
total = 0  
{$input}.each do |i|  
    total += i  
end  
$result = total > $cap ? $cap : total  
#end
```

Мягко говоря, это не очень идиоматичный Ruby, но возможно, что новичок, только что пришедший из C в Ruby, поступит именно таким образом. В теле макроса переменные *\$input*, *\$cap* и *\$result* ссылаются на аргументы вызова. Наш гипотетический программист может использовать макрос в программе Ruby так.

```
array = [1,2,3,4,5]  
#cappedTotal('array' 10 'amount')  
puts "amount is: #{amount}"
```

Если теперь воспользоваться Velocity для обработки программы и запустить полученный файл, все вроде бы нормально заработает. Вот во что превращается программа.

```

array = [1,2,3,4,5]
total = 0
array.each do |i|
  total += i
end
amount = total > 10 ? 10 : total
puts "amount is: #{amount}"

```

Теперь наш программист отходит выпить чашку чая, возвращается и пишет следующий код.

```

total = 35
#... строки кода ...
#cappedTotal('array' 10 'amount')
puts "total is #{total}"

```

Он будет удивлен. Код работает — в том смысле, что он корректно устанавливает значение `amount`. Однако рано или поздно программист столкнется с ошибкой, так как переменная `total` незаметно изменяется при выполнении макроса. Это связано с тем, что в теле макроса упоминается переменная `total`, и при его раскрытии и выполнении происходит ее изменение. Последствия этого захвата переменной `total` макросом отличаются от результатов захвата в примере, рассмотренном выше, но оба они связаны с одной и той же основной проблемой.

15.1.2. Синтаксические макросы

Из-за всех этих проблем макросы, в особенности текстовые, стали непопулярны в большинстве сред программирования. Вы все еще встретитесь с ними в языке программирования C, но современные языки полностью их избегают.

Есть два исключения — языки, которые используют синтаксические макросы: C++ и Lisp. В C++ такими синтаксическими макросами являются шаблоны, которые породили множество увлекательных подходов к генерации кода во время компиляции. Я не собираюсь много говорить о шаблонах C++, в частности потому, что я не очень с ними знаком: я работал с C++ до того, как его шаблоны стали распространенным явлением. Кроме того, C++ не является хорошим базовым языком для внутренних DSL; обычно в мире C/C++ используются внешние DSL. В конце концов, C++ является сложным инструментом даже для опытных программистов, которые не поощряют использование внутренних DSL.

Lisp — это совсем другое дело. Программисты на Lisp говорят о внутренних DSL в Lisp с незапамятных времен, поскольку Lisp является одним из старейших языков программирования и все еще активно используется. И это неудивительно, ведь Lisp ориентирован на символьную обработку, т.е. на работу с языком.

Макросы проникли глубже в сердце Lisp, чем в любом другом языке программирования. Многие базовые возможности Lisp реализуются через макросы, так что даже начинающий программист на Lisp будет их использовать, обычно даже не осознав, что это макросы. В результате, когда люди говорят о возможностях языка для реализации внутренних DSL, программисты на Lisp всегда говорят о важности макросов. При неизбежных сравнениях языков они всегда будут считать отсутствие макросов важным аргументом против сравниваемого языка.

(Это также ставит меня в несколько неловкое положение. Хотя я достаточно много баловался с Lisp, я не считаю себя серьезным Lisp-программистом и не являюсь членом сообщества Lisp).

Синтаксические макросы имеют ряд мощных возможностей, и программисты на Lisp активно их используют. Однако большая, если не большая, часть применения макросов в Lisp — шлифовка синтаксиса для работы с замыканиями (*Closure* (397)). Вот простой и глупый пример замыкания на Ruby из [5].

```
aSafe = Safe.new "secret plans"
aSafe.open do
  puts aSafe.contents
end
```

Метод open реализован следующим образом.

```
def open
  self.unlock
  yield
  self.lock
end
```

Ключевым моментом здесь является то, что содержимое замыкания не вычисляется до того момента, пока получатель не вызовет `yield`. Это гарантирует, что получатель сможет открыть сейф перед запуском переданного кода. Сравните это со следующим подходом.

```
puts aSafe.open(aSafe.contents)
```

Этот способ не сработает, потому что код, переданный в качестве параметра, вычисляется до вызова `open`. Передача кода в замыкании позволяет отложить его вычисление. **Отложенное вычисление** означает, что получающий метод выбирает, когда нужно выполнить переданный ему код (и следует ли его выполнять).

Имеет смысл так же поступать и в Lisp. Эквивалентный вызов будет иметь следующий вид.

```
(openf-safe aSafe (read-contents aSafe))
```

Можно ожидать, что это будет реализовано с помощью вызова функции, подобной следующей.

```
(defun openf-safe (safe func)
  (let ((result nil))
    (unlock-safe safe)
    (setq result (funcall func))
    (lock-safe safe)
    result))
```

Однако отложенное вычисление в данном случае не осуществляется. Для того чтобы этого добиться, нужен вызов такого рода.

```
(openf-safe aSafe (lambda() (read-contents aSafe)))
```

Это выглядит несколько запутанно. Чтобы вызов оставался простым и понятным, всю “запутанность” можно перенести в макрос.

```
(defmacro openm-safe (safe func)
  `(let (result)
     (unlock-safe ,safe)
     (princ (list result ,safe))
     (setq result ,func)
     (lock-safe ,safe)
     result))
```

Этот макрос позволяет избежать необходимости оборачивать функции в лямбда-выражения, так что вызов остается простым и понятным.

```
(openm-safe aSafe (read-contents aSafe))
```

Большая (возможно, большая) часть использования макросов Lisp заключается в предоставлении ясного синтаксиса механизма отложенного вычисления. Язык с ясным синтаксисом замыканий не нуждается в таком применении макросов.

Показанный выше макрос будет работать почти всегда, но это “почти” указывает на проблемы, например если мы вызываем его так.

```
(let (result)
  (setq result (make-safe "secret"))
  (openm-safe result (read-contents result)))
```

Проблема связана с захватом переменной — при использовании имени `result` в качестве аргумента мы получим ошибку. Захват переменной — проблема, свойственная макросам Lisp, так что для ее избежания велись упорные работы над различнымиialectами Lisp. Некоторые из этих dialectов, такие как Scheme, имеют “гиgienическую” систему макросов, которая позволяет избежать любого захвата переменной путем переопределения символов за сценой. Common Lisp имеет другой механизм — gensym. По сути, способность генерировать символы для локальных переменных гарантирует отсутствие коллизий. Этот механизм сложнее в использовании, но позволяет программисту сознательно использовать захват переменной (возникают такие ситуации, когда преднамеренный захват переменной полезен, однако обсуждение этого вопроса я оставил Полу Грэхему (Paul Graham) [16]).

Наряду с захватом переменной есть и потенциальная проблема многократного вычисления, если параметр `safe` используется в нескольких точках в раскрытии определения. Чтобы избежать этого, я должен связать параметр с другой локальной переменной (которая также нуждается в механизме gensym), что приводит к следующему результату.

```
(defmacro openm-safe2 (safe func)
  (let ((s (gensym))
        (result (gensym)))
    `(let ((,s ,safe))
       (unlock-safe ,s)
       (setq ,result ,func)
       (lock-safe ,s)
       ,result)))
```

Все эти вопросы делают макросы гораздо более сложными в написании, чем может показаться на первый взгляд. Тем не менее в Lisp активно используются отложенные вычисления с удобным синтаксисом, так как замыкания очень важны для создания управляемых абстракций и альтернативных вычислительных моделей.

Несмотря на то что большая часть макросов Lisp написана для реализации отложенных вычислений, имеются и другие полезные возможности, которые можно делать с помощью макросов Lisp. В частности, макросы обеспечивают программистам на Lisp механизм реализации управления синтаксическим деревом (*Parse Tree Manipulation* (449)).

На первый взгляд, синтаксис Lisp кажется весьма причудливым, но когда вы начнете его использовать, то поймете, что это хорошее представление синтаксического дерева программы. В каждом списке первый элемент представляет собой тип узла синтаксического дерева, а остальные элементы являются дочерними по отношению к нему. Программы Lisp активно используют вложенные функции (*Nested Function* (361)), и в ре-

зультате получается синтаксическое дерево. Благодаря макросам для манипуляций кодом Lisp до вычислений программист может работать с этим синтаксическим деревом.

В настоящее время работу с синтаксическим деревом поддерживает очень мало сред, так что такая поддержка в Lisp является отличительной чертой языка. В дополнение к элементам, поддерживающим DSL, она обеспечивает более фундаментальную работу с языком. Хорошим примером является стандартный распространенный макрос Lisp `setf`.

Хотя Lisp часто используется в качестве функционального языка, т.е. языка, который не имеет побочных эффектов, влияющих на данные, у него есть функции для хранения информации в переменных. Основной функцией здесь является `setq`, которая позволяет устанавливать переменные следующим образом.

```
(setq var 5)
```

Lisp формирует много различных структур данных из вложенных списков, и вы можете обновить данные в этих структурах, обратившись к первому элементу списка с помощью `car` и обновив его с помощью `rplaca`. Но есть много способов обращения к разным частям структур данных. В результате бесценные клетки мозга тратятся на запоминание функций доступа и обновления для каждой из них. Чтобы помочь делу, в Lisp добавили макрос `setf`, который для данной функции доступа автоматически рассчитывает и применяет соответствующие обновления. Таким образом, мы можем использовать (`car (cdr aList)`) для доступа ко второму элементу списка и (`setf (car (cdr aList)) 8`) — для его обновления.

```
(setq aList '(1 2 3 4 5 6))
(car aList) ; => 1
(car (cdr aList)) ; => 2
(rplaca aList 7)
aList ; => (7 2 3 4 5 6)
(setf (car (cdr aList)) 8)
aList ; => (7 8 3 4 5 6)
```

Это впечатляющий прием, который может показаться почти магическим. Однако существуют ограничения, которые снижают процент магии... Вы не можете сделать это с любым выражением. Вы можете поступать так только с выражениями, которые состоят из обратимых функций. Lisp ведет учет обратных функций, таких как `rplaca`, обратная к `car`. Макрос анализирует выражение своего первого аргумента и вычисляет выражение обновления путем поиска обратной функции. Определяя новые функции, вы можете указать языку Lisp обратные к ним функции, а затем использовать `setf` для обновлений.

На самом деле `setf` является более сложным, чем следует из моего краткого описания. Но важным фактом в данном обсуждении является то, что, для того чтобы определить `setf`, нужны макросы, поскольку `setf` зависит от возможности анализировать входное выражение. Эта возможность анализа своих аргументов является ключевым преимуществом макросов Lisp.

Макросы в Lisp хорошо работают с синтаксическим деревом, поскольку синтаксические структуры Lisp очень близки к этому дереву. Однако макросы — не единственное средство работы с синтаксическим деревом. C# является примером языка, который поддерживает работу с синтаксическим деревом, предоставляя возможность получить синтаксическое дерево для выражения и библиотеку — для работы с ним.

15.2. Когда это использовать

На первый взгляд, текстовые макросы весьма привлекательны. Их можно использовать с любым языком, который применяет текст; они выполняют все свои манипуляции

во время компиляции и могут реализовывать очень впечатляющее поведение, которые выходит за рамки возможностей базового языка.

Но текстовые макросы имеют не только преимущества, но и недостатки. Такие тонкие ошибки, как ошибочное раскрытие, захват переменной или множественное вычисление, часто нестабильны и трудно отслеживаемы. В процессе работы с макросами вы не дождитесь поддержки от отладчиков, интеллектуальных интегрированных сред разработки и т.п. Большинство программистов находят вложенные макросы гораздо более сложными, чем вложенные вызовы функций. Конечно, это может быть вызвано просто отсутствием практики работы с макросами, но мне кажется, что причина кроется в самих макросах.

Подводя итоги, я не рекомендую использовать текстовые макросы где-либо, кроме самых простых случаев. Я считаю, что для шаблонной генерации (*Templated Generation* (529)) они работают вполне приемлемо, предоставляя вам возможность не пытаться быть с ними слишком умными, в частности избегая вложенных макросов. В противном случае игра с ними просто не стоит свеч.

Как все вышесказанное относится к синтаксическим макросам? Я склоняюсь к тому, что почти все из сказанного относится и к ним. Хотя с ними менее вероятно получить ошибку раскрытия, другие проблемы никуда не деваются. Это заставляет меня относиться к синтаксическим макросам очень настороженно.

Контрпримером является широкое применение синтаксических макросов в сообществе Lisp. Будучи в нем аутсайдером, я бы не хотел высказывать какие-либо суждения. Мои ощущения сводятся к тому, что они имеют смысл в Lisp, но я не уверен, что логика их использования в Lisp имеет смысл для других языковых сред.

В конце концов, большинство языковых сред синтаксические макросы не поддерживают, так что у вас нет никакого выбора и беспокоиться не о чем. Если у вас они есть (например, в Lisp и C++), то зачастую они могут делать очень полезные вещи, поэтому их нужно хотя бы немного знать. Это, по сути, означает, что выбор — использовать ли синтаксические макросы — в действительности сделан вместо вас вашей языковой средой.

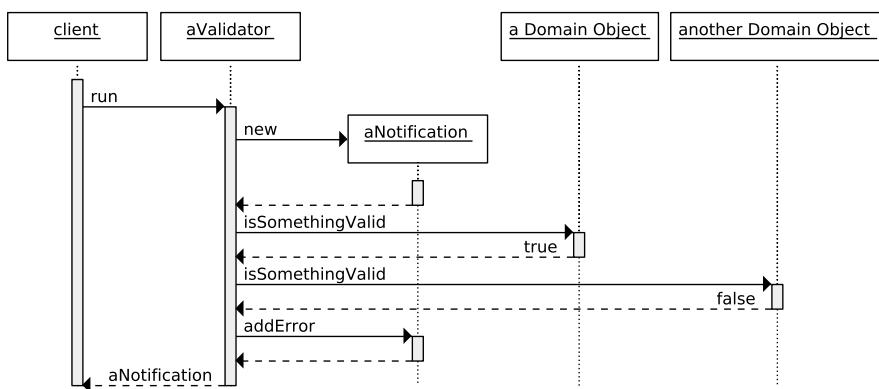
Единственный выбор, остающийся за вами, — это имеет ли смысл выбирать поддерживающий их язык программирования. На данный момент я считаю макросы наихудшим вариантом среди имеющихся альтернатив. Однако этот вывод я сделал только на основании своего личного опыта работы с ними. Должен при этом честно признаться, что я работал с такими языками недостаточно много, чтобы быть абсолютно уверенным в своей правоте.

Глава 16

Уведомление

Notification

Собирает информацию об ошибках и другие сообщения для предоставления отчета вызывающей программе



Я выполнил несколько операций, которые привели к существенным изменениям в объектной модели. Теперь, завершив работу, я хочу убедиться в корректности полученной модели. Я могу инициировать команду проверки; если все в порядке, мне будет достаточно этой информации, но если есть ошибки, я должен знать о них больше. В частности, я хочу знать обо всех ошибках, а не останавливать проверку после первой же из них.

Уведомление — это объект, который собирает ошибки. Если проверка не удалась, она добавляет информацию об ошибке в объект уведомления. Когда проверка заканчивается, она возвращает объект уведомления. Затем я могу запросить этот объект, все ли в порядке, и если нет — покопаться в ошибках.

16.1. Как это работает

Базовая форма уведомления — это коллекция ошибок. В процессе выполнения задачи мне нужна возможность добавлять ошибку в объект уведомления. Это может быть простое добавление строки сообщения об ошибке, а может включать и более активное участие объекта ошибки. Когда задача выполнена, объект уведомления возвращается вызывающему коду. По-

следний вызывает простой логический метод запроса, чтобы узнать, все ли в порядке. Если в наличии имеются ошибки, вызывающий код может опросить о них объект уведомления.

Обычно объект уведомления должен быть доступен для нескольких методов модели. Он может быть, например, передан в качестве аргумента как *параметр-накопитель* [3] или скрыт в поле соответствующего задаче объекта (такого, как объект проверки), если таковой имеется.

Основное назначение уведомления — сбор информации об ошибках, но иногда полезно заодно собирать предупреждения и информационные сообщения. Ошибка указывает, что запрошенная команда не выполнена; предупреждение генерируется, если команда выполнена, но требует внимания вызывающего кода. Информационное сообщение содержит лишь некоторую информацию о выполнении, наличие которой у вызывающей стороны может оказаться удобным.

Во многих отношениях объект уведомления действует как журнальный файл, поэтому многие функции, которые обычно встречаются при регистрации событий, могут быть полезны и здесь.

16.2. Когда это использовать

Уведомление применяется при выполнении сложных операций, которые могут вызвать многочисленные ошибки, и когда вы не хотите прерывать операцию при возникновении первой же ошибки. Для того чтобы прервать работу при первой же ошибке, просто сгенерируйте исключение. Уведомление же позволяет хранить несколько исключений, чтобы дать вызывающему коду полную картину того, к чему привел запрос.

Уведомления особенно полезны, когда пользовательский интерфейс инициирует операцию на нижнем уровне. Нижний уровень не должен пытаться непосредственно взаимодействовать с пользовательским интерфейсом, так что уведомление создает соответствующую функциональность доставки сообщений.

16.3. Очень простое уведомление (C#)

Ниже представлено очень простое уведомление, которое я использовал в книге в паре примеров. Все, что оно делает, — это хранит ошибки в виде строк.

```
class Notification...
    List<string> errors = new List<string>();
    public void AddError(String s, params object[] args) {
        errors.Add(String.Format(s, args));
    }
```

Форматная строка и параметры упрощают применение уведомления для хранения информации об ошибках, так как клиентскому коду не нужно самостоятельно создавать отформатированную строку.

Вызывающий код.....
note.AddError("Отсутствует значение {0}", property);

Я предоставляю также пару методов, возвращающих вызываемому коду логические значения, которые указывают на наличие ошибок.

```
class Notification...
    public bool IsOK {get{ return 0 == errors.Count;}}
    public bool HasErrors {get { return !IsOK;}}
```

Я также предоставляю метод для проверки и генерации исключения при наличии ошибок. Иногда этот механизм подходит для приложения лучше, чем проверка возвращаемого методом логического значения.

```
class Notification...
public void AssertOK() {
    if (HasErrors) throw new ValidationException(this);
}
```

16.4. Уведомление анализа (Java)

Вот еще одно уведомление, которое я использую в примерах к Foreign Code (315). Оно немного больше, чем написанное на C#, и более специфично, так как предназначено для ошибок определенного вида.

Поскольку это часть синтаксического анализатора ANTLR, я помещаю уведомление во встроенный помощник (Embedment Helper (537)) генерируемого синтаксического анализатора.

```
class AllocationTranslator...
private Reader input;
private AllocationLexer lexer;
private AllocationParser parser;
private ParsingNotification notification =
    new ParsingNotification();
private LeadAllocator result = new LeadAllocator();

public AllocationTranslator(Reader input) {
    this.input = input;
}

public void run() {
try {
    lexer =
        new AllocationLexer(new ANTLRReaderStream(input));
    parser =
        new AllocationParser(new CommonTokenStream(lexer));
    parser.helper = this;
    parser.allocationList();
} catch (Exception e) {
    throw new
        RuntimeException("Unexpected exception in parse", e);
}
if (notification.hasErrors())
    throw new
        RuntimeException("Parse failed: \n" + notification);
}
```

Данное уведомление обрабатывает две конкретные ошибки. Первая из них — исключение, сгенерированное самой системой ANTLR. В ANTLR это исключение распознавания. У ANTLR для этого случая имеется поведение по умолчанию, но я хочу также сохранять ошибку в уведомлении. Это можно сделать, предоставив реализацию метода для сообщения об ошибке в разделе `members` файла грамматики.

```
grammer file 'Allocation.g'.....
@members {
    AllocationTranslator helper;

    public void reportError(RecognitionException e) {
```

```

        helper.addError(e);
        super.reportError(e);
    }
}

class AllocationTranslator...
void addError(RecognitionException e) {
    notification.error(e);
}

```

Вторая ошибка возникает во время синтаксического анализа и распознается кодом встроенной трансляции (*Embedded Translation* (305)). В определенный момент грамматика просматривает список продуктов.

```

grammar file.....
productClause returns [List<ProductGroup> result]
: 'handles' p+=ID+
    {$result = helper.recognizedProducts($p);}
;

class AllocationTranslator...
List<ProductGroup>
recognizedProducts(List<Token> tokens) {
List<ProductGroup> result =
    new ArrayList<ProductGroup>();
for (Token t : tokens) {
    if (!Registry.productRepository()
        .containsId(t.getText()))
        notification.error(t, "No product for %s",
                           t.getText());
    continue;
}
result.add(Registry.productRepository()
    .findById(t.getText()));
}
return result;
}

```

В первом случае я передаю уведомлению объект исключения распознавания ANTLR, а во втором случае я передаю токен и строку сообщения об ошибке — вновь с применением форматной строки.

Внутри уведомления содержится список ошибок; в данном случае вместо строки используется более значимый объект.

```

class ParsingNotification...
private List<ParserMessage> errors =
    new ArrayList<ParserMessage>();

```

Для двух данных случаев я применяю разные типы объектов. Для исключения распознавания ANTLR я использую простую оболочку.

```

class ParsingNotification...
public void error (RecognitionException e) {
    errors.add(new RecognitionParserMessage(e));
}

class ParserMessage {}

class RecognitionParserMessage extends ParserMessage {
    RecognitionException exception;
    RecognitionParserMessage(RecognitionException exception) {

```

```

        this.exception = exception;
    }
    public String toString() {
        return exception.toString();
    }
}
}

```

Как можно заметить, суперкласс представляет собой просто пустой маркер для обобщенных действий. Со временем в него можно будет что-то добавить, но пока что достаточно и такой пустышки.

Во втором случае я собираю входящие данные в другой объект.

```

class ParsingNotification...
public void error(Token token, String message,
                   Object... args) {
    errors.add(new TranslationMessage(token,
                                       message, args));
}

class TranslationMessage extends ParserMessage {
    Token token;
    String message;

    TranslationMessage(Token token, String message,
                       Object... messageArgs) {
        this.token = token;
        this.message = String.format(message, messageArgs);
    }
    public String toString() {
        return String.format("%s (near line %d char %d)",
                           message, token.getLine(),
                           token.getCharPositionInLine());
    }
}
}

```

Я обеспечиваю наилучшую диагностическую информацию, передавая токены.

Я также предоставляю обычные методы для выяснения, имеются ли какие-либо ошибки, а также для их вывода.

```

class ParsingNotification...
public boolean isOk() {return errors.isEmpty();}
public boolean hasErrors() {return !isOk();}

public String toString() {
    return (isOk()) ? "OK" : "Errors:\n" + report();
}
public String report() {
    StringBuffer result =
        new StringBuffer("Parse errors:\n");
    for (ParserMessage m : errors)
        result.append(m).append("\n");
    return result.toString();
}
}

```

Я думаю, что наиболее важным моментом здесь является построение уведомления, которое делает вызывающий код максимально простым и компактным. Поэтому я переношу уведомлению все необходимые данные и предоставляю ему возможность самостоятельно разобраться, как составить сообщение об ошибке из этих данных.

Часть III

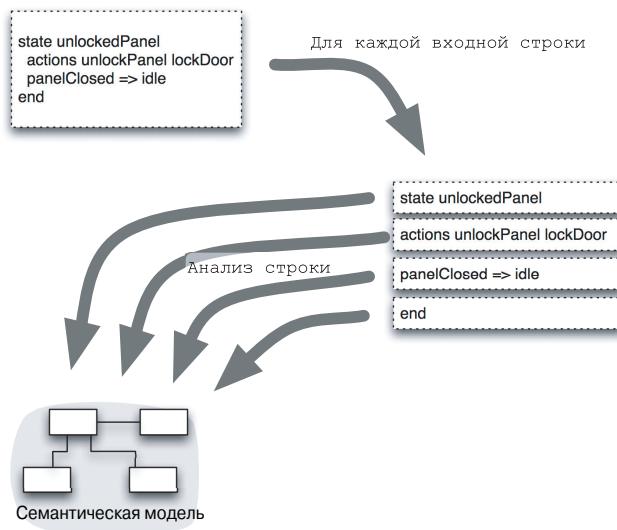
Вопросы создания внешних DSL

Глава 17

Трансляция, управляемая разделителями

Delimiter-Directed Translation

Трансляция исходного текста путем его разбиения на части (обычно на строки) с последующим анализом каждой части



17.1. Как это работает

Управляемая разделителями трансляция получает входной текст и разбивает его на фрагменты в соответствии с некоторым символом-разделителем. Вы можете использовать любые разделители, которые вам нравятся. Самым распространенным разделителем является символ конца строки, поэтому я буду использовать его.

Разбить сценарий на строки, как правило, довольно просто, поскольку в большинстве сред программирования имеются библиотечные функции, которые построчночитыва-

ют входной поток. Сложность, с которой вы можете столкнуться, — длинные строки, которые требуется физически разбить в вашем редакторе. Во многих средах есть способ сделать это, например в Unix это делается с помощью обратной косой черты в качестве последнего символа в строке.

Это выглядит довольно уродливо, и, кроме того, данный метод не работает, если в конце строки оказывается пробел (после специального символа). Поэтому лучше использовать некоторый символ продолжения строки. Выберите символ, который, будучи последним непробельным символом строки, указывает, что следующая строка является продолжением данной. Считывая вход, обратите внимание на символ продолжения строки и, если вы его видите, добавьте следующую строку к той, которую только что прочитали. При этом помните, что таких продолжений у одной строки может быть несколько.

Как вы обрабатываете строки, зависит от природы языка, с которым вы имеете дело. В простейшем случае каждая строка автономна и имеет один и тот же вид. Рассмотрим простой список правил для расчета бонусов для постоянных клиентов за проживание в гостинице.

```
score 300 for 3 nights at Bree
score 200 for 2 nights at Dol Amroth
score 150 for 2 nights at Orthanc
```

Я называю строки автономными, поскольку ни одна из них не влияет на другие. Я мог бы безопасно изменять порядок строк или удалять строки без изменения толкования любой конкретной строки. Они имеют один и тот же вид, поскольку каждая из них кодирует однотипную информацию. Поэтому обрабатывать строки довольно просто; к каждой из них я применяю одну и ту же функцию, которая извлекает нужную информацию и транслирует ее в нужное представление. Если я использую встроенную трансляцию (*Embedded Translation* (305)), это означает помещение информации в семантическую модель (*Semantic Model* (171)). Если я применяю построение дерева (*Tree Construction* (289)), то это означает создание абстрактного синтаксического дерева. Я очень редко встречал построение дерева в случае управляемой разделителями трансляции, поэтому я буду полагать, что здесь используется встроенная трансляция (достаточно часто встречается также *Embedded Interpretation* (311)).

Как выбрать интересующую вас информацию, зависит от возможностей обработки строк в вашем языке и от сложности строк, с которыми предстоит иметь дело. Если это возможно, то самый простой способ — разложить входную строку на части с помощью соответствующей функции. В большинстве библиотек, работающих со строками, имеется такая функция, разбивающая строку на элементы, разделенные специальными подстроками. В нашем случае в качестве разделителя выступают пробелы, так что количество бонусов можно получить как второй элемент разложения строки.

Иногда разбить строку не так просто, как в приведенном примере. В таком случае зачастую рекомендуется применять регулярные выражения. Для извлечения частей строки можно использовать группы регулярных выражений. Регулярное выражение обладает гораздо большими выразительными возможностями, чем разбиение строки, и, кроме того, это хороший способ проверить синтаксическую корректность строки. Следует заметить, что регулярные выражения являются более сложными, и многие программисты считают их слишком запутанными и неуклюжими. Часто в работе помогает следующий метод: разбить большое регулярное выражение на подвыражения, определить каждое из них отдельно, а затем объединить их (этот метод я называю составными регулярными выражениями).

Теперь рассмотрим ситуацию, когда у вас имеются строки различного вида. Это может быть код DSL, который описывает раздел содержимого начальной страницы местной газеты.

```
border grey
headline "Musical Cambridge"
```

```
filter by date in this week  
show concerts in Cambridge
```

В этом случае каждая строка автономна, но все они должны обрабатываться по-разному. Можно воспользоваться условным выражением, которое проверяет различные виды строк и вызывает соответствующие подпрограммы обработки.

```
if (isBorder()) parseBorder();  
else if (isHeadline()) parseHeadline();  
else if (isFilter()) parseFilter();  
else if (isShow()) parseShow();  
else throw new RecognitionException(input);
```

Проверки в условных выражениях могут использовать регулярные выражения или другие операции со строками. Иногда в условных выражениях рекомендуется применять регулярные выражения непосредственно, но я предпочитаю вызовы методов.

Помимо чисто изоморфных или полиморфных строк, можно получить гибрид, в котором каждая строка имеет одну и ту же глобальную структуру разделения на отдельные предложения, но каждое из них имеет свою форму. Вот еще одна версия кода DSL.

```
300 for stay 3 nights at Bree  
150 per day for stay 2 nights at Bree  
50 for spa treatment at Dol Amroth  
60 for stay 1 night at Orthanc or Helm's Deep or Dunharrow  
1 per dollar for dinner at Bree
```

Здесь мы имеем широкую изоморфную структуру. Стока начинаются с предложения о сумме, за которым следуют слово `for`, предложение действия, слово `at` и предложение местоположения. На такую структуру можно ответить одной процедурой обработки верхнего уровня, которая идентифицирует строку и вызывает свою подпрограмму обработки для каждого предложения. Процедуры обработки предложений следуют полиморфному шаблону.

Это описание можно отнести и к грамматикам, использующимся в синтаксически управляемой трансляции (Syntax-Directed Translation (229)). Полиморфные строки и предложения обрабатываются в грамматиках альтернативно, в то время как изоморфные строки обрабатываются правилами продукции без альтернатив. Использование методов для разбиения строк на предложения похоже на применение подправил.

Обработка неавтономных инструкций с помощью управляемой разделителями трансляции приводит к дополнительным сложностям, поскольку теперь требуется отслеживать определенную информацию о состоянии синтаксического анализа. В качестве примера может служить мой конечный автомат из начала книги, в котором были отдельные разделы для событий, команд и состояний. Стока `unlockPanel PNUL` должна обрабатываться по-разному в разделе событий и в разделе команд, даже если она имеет один и тот же синтаксический вид. Кроме того, появление этой строки в определении состояния является ошибкой.

Хороший способ справиться с такой ситуацией — иметь семейство различных синтаксических анализаторов для каждого состояния анализа. Так, анализатор конечного автомата мог бы иметь анализатор строки верхнего уровня и отдельные анализаторы для блока команды, блока события, блока сбрасывающего события и блока состояния. Когда анализатор строки верхнего уровня видит ключевое слово `events`, он передает анализ текущей строки анализатору строки события. Это, конечно, просто применение шаблона проектирования *State* [15].

Распространенный вопрос, который возникает при использовании управляемой разделителями трансляции, — обработка пробелов, в особенности вокруг операторов. Если

у вас есть строки вида `свойство = значение`, определитесь, будут ли пробелы вокруг знака равенства обязательными. Выбор необязательных пробелов может усложнить обработку строки, а требование их обязательности сделает DSL сложнее в использовании. Ситуация может быть еще хуже, если имеется различие между использованием одного и нескольких пробелов или между различными пробельными символами, такими как пробелы и символы табуляции.

У этого вида обработки есть определенная закономерность. Постоянно повторяется соответствие строки некоторому шаблону и выполняются правила обработки для этого шаблона. Это естественным образом приводит к мысли о некоторой схеме, в рамках которой имеется ряд объектов, каждый из которых содержит регулярное выражение для определенного вида строки, который обрабатывается этим объектом. Затем выполняется проход по всем объектам. Можно также добавить в схему некоторое указание общего состояния анализатора. Для упрощения настройки такой схемы можно добавить к ней DSL.

Конечно, не я первый об этом подумал. Именно такой стиль обработки используется генераторами лексических анализаторов, такими как Lex. Имеются как аргументы за использование инструментов такого рода, так и против. Если вы зайдете достаточно далеко, чтобы захотеть использовать схему, учтите: переход к синтаксически управляемой трансляции заведет вас ненамного дальше, но при этом вы получите доступ к большему количеству более мощных инструментов.

17.2. Когда это использовать

Большое преимущество управляемой разделителями трансляции в том, что она очень проста в использовании. Ее главная альтернатива, синтаксически управляемая трансляция (*Syntax-Directed Translation* (229)), требует долгого обучения работе с грамматиками. Управляемая разделителями трансляция опирается исключительно на хорошо знакомые большинству программистов методы и, таким образом, применима без дополнительных усилий.

Как это часто бывает, оборотной стороной доступности является трудность обработки более сложных языков. Управляемая разделителями трансляция очень хорошо работает с простыми языками, в особенности с теми, которые не требуют вложенного контекста. С ростом сложности языка такая трансляция быстро приводит к путанице.

В связи с этим я, как правило, выступаю за применение управляемой разделителями трансляции только тогда, когда есть автономные инструкции или, быть может, только один вложенный контекст. И даже в этом случае я предпочел бы использовать синтаксически управляемую трансляцию (кроме, конечно, ситуации, когда я работаю с командой, не готовой к изучению этой технологии).

17.3. Карты постоянных клиентов (C#)

Если вы имели несчастье быть разъездным консультантом, катающимся по всей стране, как я, значит, вы знакомы с различными бонусами туристических фирм, которые хотят вознаградить вас за путешествие возможностью путешествовать еще больше... Давайте представим набор правил для гостиничной сети в виде DSL следующим образом.

```
300 for stay 3 nights at Bree
150 per day for stay 2 nights at Bree
50 for spa treatment at Dol Amroth
60 for stay 1 night at Orthanc or Helm's Deep or Dunharrow
1 per dollar for dinner at Bree
```

17.3.1. Семантическая модель

Диаграмма классов соответствующей семантической модели представлена на рис. 17.1.

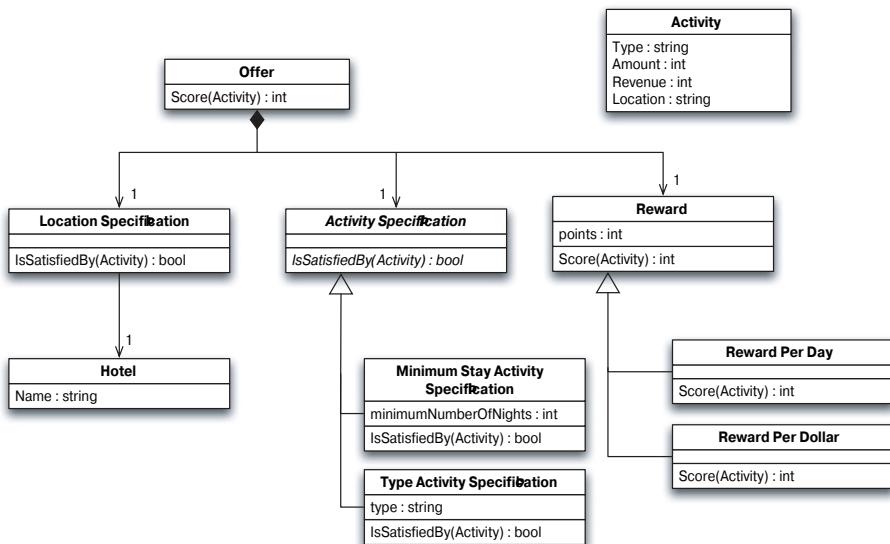


Рис. 17.1. Диаграмма классов семантической модели

Каждая строка сценария определяет одно предложение гостиничной сети. Основная задача предложения — указать количество очков за те или иные действия клиента. Действие представляет собой простое представление данных.

```

class Activity...
    public string Type { get; set; }
    public int Amount { get; set; }
    public int Revenue { get; set; }
    public string Location { get; set; }
  
```

Предложение состоит из трех компонентов. *Спецификация [7] местоположения* проверяет, выполняется ли действие в нужном месте, чтобы за него можно было начислить очки; спецификация действия проверяет, предусматривается ли за него начисление очков. Если обе спецификации удовлетворены, подсчитывается количество очков.

Самый простой из этих трех компонентов — спецификация местоположения. Он просто сверяет название отеля со списком названий.

```

class LocationSpecification...
    private readonly IList<Hotel> hotels = new List<Hotel>();

    public LocationSpecification(params String[] names) {
        foreach (string n in names)
            hotels.Add(Repository.HotelNamed(n));
    }

    public bool IsSatisfiedBy(Activity a) {
        Hotel hotel = Repository.HotelNamed(a.Location);
        return hotels.Contains(hotel);
    }
  
```

Мне нужны две спецификации действия двух видов. Одна проверяет, остановливался ли клиент в отеле указанное количество ночей.

```
abstract class ActivitySpecification {
    public abstract bool IsSatisfiedBy(Activity a);
}

class MinimumNightStayActivitySpec : ActivitySpecification {
    private readonly int minimumNumberOfNights;

    public MinimumNightStayActivitySpec(int numberOfNights) {
        this.minimumNumberOfNights = numberOfNights;
    }

    public override bool IsSatisfiedBy(Activity a) {
        return a.Type == "stay"
            ? a.Amount >= minimumNumberOfNights
            : false;
    }
}
```

Вторая проверяет корректность вида действия.

```
class TypeActivitySpec : ActivitySpecification {
    private readonly string type;

    public TypeActivitySpec(string type) {
        this.type = type;
    }

    public override bool IsSatisfiedBy(Activity a) {
        return a.Type == type;
    }
}
```

Классы вознаграждений подсчитывают размер вознаграждений.

```
class Reward {
    protected int points;

    public Reward(int points) { this.points = points; }
    virtual public int Score(Activity activity) {
        return points;
    }
}

class RewardPerDay : Reward {
    public RewardPerDay(int points) : base(points) {}

    public override int Score(Activity activity) {
        if (activity.Type != "stay")
            throw new ArgumentException(
                "can only use per day scores on stays");
        return activity.Amount * points;
    }
}

class RewardPerDollar : Reward {
    public RewardPerDollar(int points) : base(points) {}

    public override int Score(Activity activity) {
        return activity.Revenue * points;
    }
}
```

17.3.2. Синтаксический анализатор

Базовый алгоритм синтаксического анализатора заключается в чтении каждой входной строки и ее обработке.

```
class OfferScriptParser...
    readonly TextReader input;
    readonly List<Offer> result = new List<Offer>();
    public OfferScriptParser(TextReader input) {
        this.input = input;
    }
    public List<Offer> Run() {
        string line;
        while ((line = input.ReadLine()) != null) {
            line = appendContinuingLine(line);
            parseLine(line);
        }
        return result;
    }
```

В данном примере я хочу поддерживать "&" в качестве символа продолжения. Его работа обеспечивается простой рекурсивной функцией.

```
class OfferScriptParser...
    private string appendContinuingLine(string line) {
        if (IsContinuingLine(line)) {
            var first = Regex.Replace(line, @"\s*\"", "");
            var next = input.ReadLine();
            if (null == next) throw new RecognitionException(line);
            return first.Trim() + " " + appendContinuingLine(next);
        }
        else return line.Trim();
    }
    private bool IsContinuingLine(string line) {
        return Regex.IsMatch(line, @"\s*\"");
    }
```

Так все строки-продолжения будут собраны в единую строку.

Синтаксический анализ строки я начинаю с удаления комментариев и игнорирования пустых строк. После того как это сделано, я начинаю собственно анализ, делегируя его новому объекту.

```
class OfferScriptParser...
    private void parseLine(string line) {
        line = removeComment(line);
        if (IsEmpty(line)) return;
        result.Add(new OfferLineParser().Parse(line.Trim()));
    }
    private bool IsEmpty(string line) {
        return Regex.IsMatch(line, @"^\s*\"");
    }
    private string removeComment(string line) {
        return Regex.Replace(line, @"\#.*", "");
    }
```

Здесь для анализа каждой строки использован *объект метода* [3], поскольку, как мне кажется, остальная часть синтаксического анализа достаточно сложна, чтобы выполнять ее отдельно. Объект метода не сохраняет состояния, так что я мог бы повторно использовать его экземпляр, но я предпочитаю всякий раз создавать новый экземпляр, если нет уважительной причины не делать этого.

Базовый метод синтаксического анализа разбивает строку на предложения и вызывает отдельные методы анализа для каждого предложения. (Я мог бы попытаться сделать все в одном большом регулярном выражении, но, когда я представляю получившийся бы в результате код, у меня начинается головокружение...)

```
class OfferLineParser...
public Offer Parse(string line) {
    var result = new Offer();

    const string rewardRegexp = @"(?<reward>.*)";
    const string activityRegexp = @"(?<activity>.*)";
    const string locationRegexp = @"(?<location>.*)";

    var source = rewardRegexp + keywordToken("for") +
        activityRegexp + keywordToken("at") + locationRegexp;

    var m = new Regex(source).Match(line);
    if (!m.Success) throw new RecognitionException(line);

    result.Reward = parseReward(m.Groups["reward"].Value);
    result.Location =
        parseLocation(m.Groups["location"].Value);
    result.Activity =
        parseActivity(m.Groups["activity"].Value);
    return result;
}

private String keywordToken(String keyword) {
    return @"\s+" + keyword + @"\s+";
}
```

По моим стандартам, это довольно длинный метод. Я думал, как бы его разбить, но основное поведение метода заключается в разделении регулярных выражений на группы с последующим отображением результатов анализа групп на выходные данные. Между определением и использованием этих групп существуют сильные семантические связи, так что мне кажется, что лучше иметь большой метод, чем пытаться его разложить. Поскольку суть этого метода заключается в регулярном выражении, я поместил сборку этого регулярного выражения в отдельную строку, чтобы привлечь к нему внимание.

Я мог бы все сделать с помощью одного регулярного выражения, а не отдельных выражений (rewardRegexp, activityRegexp, locationRegexp). Всякий раз, столкнувшись со сложным регулярным выражением, таким как это, я предпочитаю разбивать его на более простые регулярные выражения, которые можно объединять. Этую методику я называю **составным регулярным выражением** [8] и считаю, что так гораздо легче понять, что происходит.

Когда строка разделена на фрагменты, можно поочередно выполнять синтаксический анализ каждого фрагмента. Начнем со спецификации местоположения, так как это самый простой момент. Здесь основная сложность в том, что может быть как одно местоположение, так и несколько, разделенных словом "or".

```
class OfferLineParser...
private LocationSpecification parseLocation(string input) {
    if (Regex.IsMatch(input, @"\bor\b"))
        return parseMultipleHotels(input);
    else
        return new LocationSpecification(input);
}
private
```

```
LocationSpecification parseMultipleHotels(string input) {
    String[] hotelNames = Regex.Split(input, @"\s+or\s+");
    return new LocationSpecification(hotelNames);
}
```

Что касается предложения действия, то я должен обрабатывать два вида действий. Простейшее действие — надо просто извлечь его тип, в отличие от действия, заключающегося в остановке в отеле, когда нужно найти минимальное количество ночей и выбрать другую спецификацию действия.

```
class OfferLineParser...
private ActivitySpecification parseActivity(string input) {
    if (input.StartsWith("stay"))
        return parseStayActivity(input);
    else return new TypeActivitySpec(input);
}

private ActivitySpecification
parseStayActivity(string input) {
const string stayKeyword = @"^stay\s+";
const string nightsKeyword = @"\s+nights?$/";
const string amount = @"(?<amount>\d+)";
const string source = stayKeyword + amount + nightsKeyword;

var m = Regex.Match(input, source);
if (!m.Success) throw new RecognitionException(input);
return new MinimumNightStayActivitySpec(
    Int32.Parse(m.Groups["amount"].Value));
}
```

Последнее предложение — предложение вознаграждения. Здесь следует просто идентифицировать базис для вознаграждения и вернуть подходящий подкласс класса вознаграждения.

```
class OfferLineParser...
private Reward parseReward(string input) {
    if (Regex.IsMatch(input, @"\d+$"))
        return new Reward(Int32.Parse(input));
    else if (Regex.IsMatch(input, @"\d+ per day$"))
        return new RewardPerDay(
            Int32.Parse(extractDigits(input)));
    else if (Regex.IsMatch(input, @"\d+ per dollar$"))
        return new RewardPerDollar(
            Int32.Parse(extractDigits(input)));
    else throw new RecognitionException(input);
}
private string extractDigits(string input) {
    return Regex.Match(input, @"\d+").Value;
}
```

17.4. Синтаксический анализ неавтономных инструкций контроллера мисс Грант (Java)

В качестве примера я воспользуюсь знакомым нам конечным автоматом.

```
events
doorClosed D1CL
drawerOpened D2OP
```

```

lightOn      L1ON
doorOpened   D1OP
panelClosed  PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel PNUL
  lockPanel   PNLK
  lockDoor    D1LK
  unlockDoor  D1UL
end

state idle
  actions unlockDoor lockPanel
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions unlockPanel lockDoor
  panelClosed => idle
end

```

Код разделен на несколько блоков: список команд, список событий, список сбрасывающих событий и по блоку для каждого состояния. Каждый блок имеет собственный синтаксис, так что можно рассматривать синтаксический анализатор, находящийся в различных состояниях при работе с каждым из блоков. Каждое состояние синтаксического анализатора распознает свой вид входной информации. В результате я решил использовать шаблон *State* [15], в котором главный анализатор конечного автомата использует различные анализаторы для обработки различных типов входных строк. (Это решение можно рассматривать и как шаблон *Strategy* [15]; разницу между ними часто сложно определить.)

Я начинаю чтение файла с помощью статического метода загрузки.

```

class StateMachineParser...
public static StateMachine loadFile(String fileName) {
  try {
    StateMachineParser loader =
      new StateMachineParser(new FileReader(fileName));
    loader.run();
    return loader.machine;
  } catch (FileNotFoundException e) {
    throw new RuntimeException(e);
  }
}

```

```

    }

public StateMachineParser(Reader reader) {
    input = new BufferedReader(reader);
}

private final BufferedReader input;

```

Метод `run` разбивает вход на строки и передает строку текущему синтаксическому анализатору строки, начиная с верхнего уровня.

```

class StateMachineParser...
void run() {
    String line;
    setLineParser(new TopLevelLineParser(this));
    try {
        while ((line = input.readLine()) != null)
            lineParser.parse(line);
        input.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    finishMachine();
}

private LineParser lineParser;
void setLineParser(LineParser lineParser) {
    this.lineParser = lineParser;
}

```

Синтаксические анализаторы строк образуют простую иерархию.

```

abstract class LineParser {
    protected final StateMachineParser context;

    protected LineParser(StateMachineParser context) {
        this.context = context;
    }
}

class TopLevelLineParser extends LineParser {
    TopLevelLineParser(StateMachineParser parser) {
        super(parser);
    }
}

```

Сначала суперкласс синтаксического анализатора строк удаляет комментарии и излишние пробельные символы. После того как это сделано, управление передается подклассу.

```

class LineParser...
void parse(String s) {
    line = s;
    line = removeComment(line);
    line = line.trim();
    if (isBlankLine()) return;
    doParse();
}

protected String line;

private boolean isBlankLine() {
    return line.matches("^\\s*$");
}

```

```

private String removeComment(String line) {
    return line.replaceFirst("#.*", "");
}

abstract void doParse();

```

Разбирая строку, я следую тому же базовому плану во всех синтаксических анализаторах строк. Метод `doParse` представляет собой условную инструкцию, в которой каждое условие проверяет соответствие шаблону строки. Если имеется соответствие шаблону, я вызываю код обработки этой строки.

Бот как выглядит условие для верхнего уровня.

```

class TopLevelLineParser...
void doParse() {
    if
        (hasOnlyWord("commands"))
            context.setLineParser(
                new CommandLineParser(context));
    else if
        (hasOnlyWord("events"))
            context.setLineParser(
                new EventLineParser(context));
    else if
        (hasOnlyWord("resetEvents"))
            context.setLineParser(
                new ResetEventLineParser(context));
    else if
        (hasKeyword("state")) processState();
        else failToRecognizeLine();
}

```

Здесь используются некоторые общие проверки, которые я поместил в суперкласс.

```

class LineParser...
protected boolean hasOnlyWord(String word) {
    if (words(0).equals(word)) {
        if (words().length != 1) failToRecognizeLine();
        return true;
    }
    else return false;
}

protected boolean hasKeyword(String keyword) {
    return keyword.equals(words(0));
}

protected String[] words() {
    return line.split("\s+");
}

protected String words(int index) {
    return words()[index];
}

protected void failToRecognizeLine() {
    throw new RecognitionException(line);
}

```

В большинстве случаев на верхнем уровне выполняется только просмотр открывающей блок команды, а затем происходит переключение синтаксического анализатора

строки на новый, необходимый для этого блока. Случай состояния более сложен — я вернусь к нему позже.

Я мог бы в условных выражениях вместо вызова методов использовать регулярные выражения и вместо `hasOnlyWord("commands")` записать `line.matches("commands\\s*")`. Регулярные выражения — это очень мощный инструмент, но я не без оснований предпочитаю вызовы методов. В первую очередь, это ясность: я считаю, что `hasKeyword` проще понять, чем соответствующее регулярное выражение. Как и любой другой код, регулярные выражения часто выигрывают от обворачивания в методы с понятными именами, чтобы сделать их проще для понимания. Конечно, я мог бы реализовать метод `hasKeyword` с помощью регулярного выражения, а не разбивать строку на слова и тестировать первое из них. Но раз уж во многих тестах в этом синтаксическом анализе используется разбиение на слова, представляется более простым решением применять его везде, где это возможно.

Метод также позволяет мне сделать больше: в данном случае — убедиться в том, что после слова "commands" в строке нет другого текста. Если бы я использовал "голые" регулярные выражения в условных конструкциях, для такой проверки пришлось бы прибегнуть к дополнительным регулярным выражениям.

В качестве следующего шага давайте рассмотрим строку в блоке команд. Это может быть либо строка определения, либо слово `end`.

```
class CommandLineParser...
void doParse() {
    if (hasOnlyWord("end")) returnToTopLevel();
    else if (words().length == 2)
        context.registerCommand(new Command(words(0),
                                             words(1)));
    else failToRecognizeLine();
}

class LineParser...
protected void returnToTopLevel() {
    context.setLineParser(new TopLevelLineParser(context));
}

class StateMachineParser...
void registerCommand(Command c) {
    commands.put(c.getName(), c);
}
private Map<String, Command> commands =
    new HashMap<String, Command>();
Command getCommand(String word) {
    return commands.get(word);
}
```

В дополнение к управлению общим синтаксическим анализом у меня имеется анализатор конечного автомата, работающий как таблица символов (*Symbol Table* (177)).

Коды обработки событий и сбрасывающих событий очень похожи, поэтому перейду к рассмотрению обработки состояний. Первое, чем отличается обработка состояний, — это более сложный код синтаксического анализатора верхнего уровня.

```
class TopLevelLineParser...
private void processState() {
    State state = context.obtainState(words(1));
    context.primeMachine(state);
    context.setLineParser(new StateLineParser(context, state));
}
```

```

class StateMachineParser...
State obtainState(String name) {
    if (!states.containsKey(name))
        states.put(name, new State(name));
    return states.get(name);
}
void primeMachine(State state) {
    if (machine == null) machine = new StateMachine(state);
}
private StateMachine machine;

```

Первое упомянутое состояние становится стартовым (начальным), — отсюда и метод `primeState`. При первом упоминании состояния я помещаю его в таблицу символов и, таким образом, использую метод `obtain` (в соответствии с моим соглашением об именовании это означает “получить, если существует, или создать в противном случае”).

Синтаксический анализатор строк для блока состояния более сложен, так как имеется больше видов строк, которые он должен обрабатывать.

```

class StateLineParser...
void doParse() {
    if (hasOnlyWord("end")) returnToTopLevel();
    else if (isTransition()) processTransition();
    else if (hasKeyword("actions")) processActions();
    else failToRecognizeLine();
}

```

Действия я просто добавляю к состоянию.

```

class StateLineParser...
private void processActions() {
    for (String s : wordsStartingWith(1))
        state.addAction(context.getCommand(s));
}

class LineParser...
protected String[] wordsStartingWith(int start) {
    return Arrays.copyOfRange(words(), start, words().length);
}

```

В этом случае я мог бы просто использовать цикл наподобие следующего.

```

for (int i = 1; i < words().length; i++)
    state.addAction(context.getCommand(words(i)));

```

Однако я думаю, что использование в качестве инициализатора цикла 1 вместо обычного 0 — слишком тонкое изменение, чтобы эффективно пояснить читателю кода, что именно я делаю.

Для случая перехода требуется большее количество кода как в условии, так и в действии.

```

class StateLineParser...
private boolean isTransition() {
    return line.matches(".*=>.*");
}
private void processTransition() {
    String[] tokens = line.split("=>");
    Event trigger = context.getEvent(tokens[0].trim());
    State target = context.obtainState(tokens[1].trim());
    state.addTransition(trigger, target);
}

```

Я не использую разбиение на слова, как ранее, поскольку хочу, чтобы были корректны выражения вида `drawerOpened=>waitingForLight` (без пробелов слева и справа от оператора).

Как только входной файл обработан, все, что остается — убедиться в том, что сбрасывающие события добавляются в конечный автомат. Я делаю это в последнюю очередь, поскольку сбрасывающие события могут быть указаны перед объявлением первого состояния.

```
class StateMachineParser...
private void finishMachine() {
    machine.addResetEvents(resetEvents.toArray(
        new Event[resetEvents.size()]));
```

Важным вопросом является разделение обязанностей между синтаксическим анализатором конечного автомата и синтаксическими анализаторами строк. Это также классический вопрос шаблона *State*: в каком соотношении разделяется поведение в общем контексте в различных объектах состояний? В этом примере я показал децентрализованный подход, при котором я стараюсь делать как можно больше работы в различных синтаксических анализаторах строк. В качестве альтернативы можно поместить это поведение в синтаксический анализатор конечного автомата, используя синтаксические анализаторы строк просто для извлечения нужной информации из текста.

Я проиллюстрирую сказанное, сравнив два способа обработки блока команд. Вот децентрализованный способ, реализованный ранее.

```
class CommandLineParser...
void doParse() {
    if (hasOnlyWord("end")) returnToTopLevel();
    else if (words().length == 2)
        context.registerCommand(new Command(words(0),
            words(1)));
    else failToRecognizeLine();
}

class LineParser...
protected void returnToTopLevel() {
    context.setLineParser(new TopLevelLineParser(context));
}

class StateMachineParser...
void registerCommand(Command c) {
    commands.put(c.getName(), c);
}
private Map<String, Command> commands =
    new HashMap<String, Command>();
Command getCommand(String word) {
    return commands.get(word);
}
```

А вот так выглядит централизованный подход, когда все поведение сосредоточивается в синтаксическом анализаторе конечного автомата.

```
class CommandLineParser...
void doParse() {
    if (hasOnlyWord("end"))
        context.handleEndCommand();
    else if (words().length == 2)
        context.handleCommand(words(0), words(1));
    else failToRecognizeLine();
```

```
}

class StateMachineParser...
void handleCommand(String name, String code) {
    Command command = new Command(name, code);
    commands.put(command.getName(), command);
}
public void handleEndCommand() {
    lineParser = new TopLevelLineParser(this);
}
```

Недостатком децентрализованного подхода является то, что, поскольку синтаксический анализатор конечного автомата выступает в качестве таблицы символов, он постоянно используется синтаксическими анализаторами строк для доступа к данным. Постоянное получение данных из объекта, как правило, дурно пахнет. При централизованном подходе о таблице символов не нужно знать никаким другим объектам, так что нет необходимости выставлять состояние напоказ. Недостатком централизованного подхода, однако, является то, что большое количество логики переносится в синтаксический анализатор конечного автомата, а это может сделать его слишком сложным. Чем больше DSL, тем большей проблемой это может оказаться.

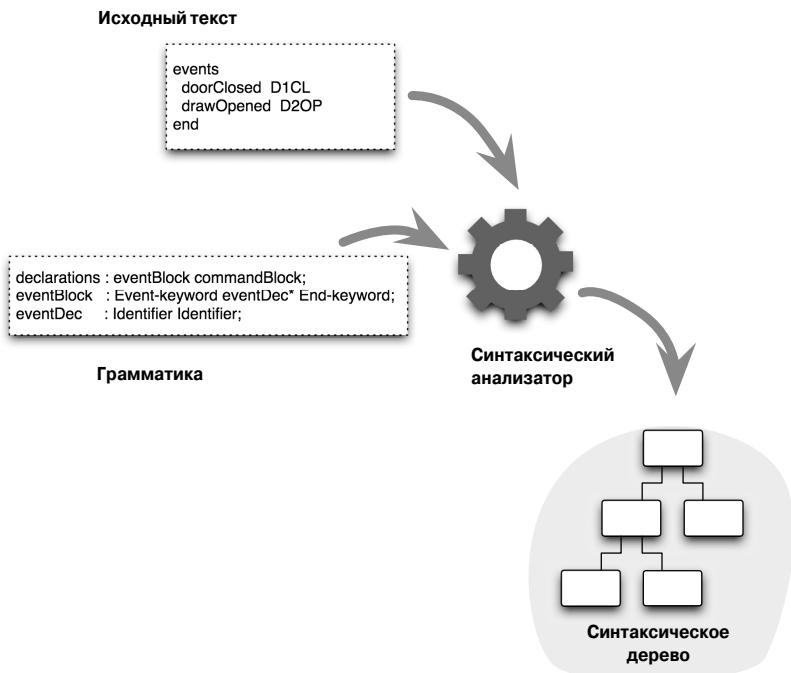
У обоих вариантов имеются свои проблемы, и, признаюсь, в данном случае у меня нет какого-либо четкого предпочтения.

Глава 18

Синтаксически управляемая трансляция

Syntax-Directed Translation

Транслирует исходный текст путем определения грамматики и ее применения для трансляции структуры



Компьютерные языки имеют естественную тенденцию следовать иерархической структуре с множественными уровнями контекста. Корректный синтаксис такого языка можно определить, создав грамматику, которая описывает, как элементы языка разбиваются на подэлементы.

При синтаксически управляемой трансляции эта грамматика используется для определения синтаксического анализатора, который может превратить исходный текст в синтаксическое дерево, имитирующее структуру правил грамматики.

18.1. Как это работает

Если вы читали книги о языках программирования, то наверняка встречались с понятием “грамматика”. Грамматика представляет собой способ определения корректного синтаксиса языка программирования. Рассмотрим часть моего вводного примера конечного автомата, которая объявляет события и команды.

```
events
  doorClosed D1CL
  drawerOpened D2OP
# ...
end

commands
  unlockPanel PNUL
  lockPanel PNLK
# ...
end
```

Синтаксический вид этих объявлений можно определить с помощью следующей грамматики.

```
declarations : eventBlock commandBlock;
eventBlock   : Event-keyword eventDec* End-keyword;
eventDec    : Identifier Identifier;
commandBlock : Command-keyword commandDec* End-keyword;
commandDec  : Identifier Identifier;
```

Такая грамматика предоставляет удобочитаемое для человека определение языка. Грамматики обычно записываются в форме Бэкуса–Наура (BNF (237)). Грамматика упрощает для людей понимание того, что является корректным синтаксисом языка программирования. При использовании синтаксически управляемой трансляции грамматику можно применять в качестве основы для разработки программы для обработки этого языка.

Такая обработка может быть получена из грамматики несколькими путями. Один подход заключается в использовании грамматики как спецификации и руководства для создания рукописного синтаксического анализатора. Двумя вариантами решения этой задачи являются создание синтаксического анализатора рекурсивного спуска (Recursive Descent Parser (253)) и применение шаблона Parser Combinator (263). Альтернативой является использование грамматики в качестве DSL и применение генератора синтаксического анализатора (Parser Generator (277)) для автоматического создания синтаксического анализатора на основе самой грамматики. В этом случае вам не придется писать код ядра синтаксического анализатора самостоятельно — он будет сгенерирован из грамматики.

При всей своей полезности грамматика справляется только с частью проблемы. Она может указать, как превратить исходный текст в структуру данных синтаксического дерева. Но почти всегда с исходным текстом необходимо сделать нечто большее. Поэтому генераторы синтаксических анализаторов предоставляют способы встраивания дальнейшего поведения в анализатор, так что вы можете сделать что-то вроде наполнения семантической модели (Semantic Model (171)). Таким образом, хотя генератор синтаксических анализаторов и делает для вас большую работу, вам все равно нужно внести свой вклад для создания чего-то действительно полезного. Таким образом, генератор синтаксиче-

ских анализаторов является отличным примером практического применения DSL. Он не решает всей проблемы, но делает значительную ее часть гораздо проще. Это также DSL с долгой историей.

18.1.1. Лексический анализатор

Почти всегда при синтаксически управляемой трансляции происходит разделение на лексический и синтаксический анализаторы. Лексический анализатор, который также называется **токенизатором** или **сканером**, выполняет первый этап обработки исходного текста. Лексический анализатор разбивает символы исходного текста на токены, которые представляют собой более осмыслиенные части входной информации.

Токены, как правило, определяются с помощью регулярных выражений. Вот как могут выглядеть правила лексического анализа для команд и событий в приведенном выше примере.

```
event-keyword : 'events';
command-keyword: 'commands';
end-keyword    : 'end';
identifier      : [a-z A-Z 0-9]*;
```

Вот очень маленький фрагмент исходного текста.

```
events
  doorClosed D1CL
  drawOpened D2OP
end
```

Правила лексического анализа превратят его в последовательность токенов.

```
[Event-keyword: "events"]
[Identifier: "doorClosed"]
[Identifier: "D1CL"]
[Identifier: "drawOpened"]
[Identifier: "D2OP"]
[End-keyword: "end"]
```

Каждый токен представляет собой объект с, по сути, двумя свойствами: типом и содержимым. Тип — это вид нашего токена, например `Event-keyword` или `Identifier`. Содержимое — это текст, которому лексическим анализатором было найдено соответствие: `events` или `doorClosed`. Для ключевых слов содержимое практически не имеет значения — достаточно знать тип. Для идентификаторов содержимое очень важно, так как это данные, которые будут использоваться в дальнейшем синтаксическом анализе.

Лексический анализ отделяется по нескольким причинам. Во-первых, это упрощает синтаксический анализатор, потому что теперь он может работать в терминах токенов, а не просто символов. Во-вторых, это повышает его эффективность: код реализации, необходимый для сборки набора символов в токены, отличается от кода, применяемого в синтаксическом анализаторе. (В теории автоматов лексический анализатор, как правило, представляет собой конечный автомат, в то время как синтаксический анализатор обычно является стековой машиной.) Поэтому такое разделение — традиционный подход, хотя и оспариваемый некоторыми современными разработчиками. (ANTLR использует стековую машину в своем лексическом анализаторе, а некоторые из современных синтаксических анализаторов комбинируют лексический и синтаксический анализ в синтаксические анализаторы без сканирования.)

Правила лексического анализа проверяются по порядку до первого успешного совпадения. Таким образом, вы не можете использовать строку `events` как идентификатор,

потому что лексический анализатор всегда будет распознавать ее как ключевое слово. Это обычно хорошо, потому что позволяет избежать путаницы, такой как пресловутые выражения PL/1 вида `if if = then then then = if ;`. Однако иногда такое ограничение приходится обходить с помощью той или иной формы альтернативной токенизации (Alternative Tokenization (325)).

Если вы были достаточно внимательны при сравнении токенов с исходным текстом, то заметили, что в списке токенов кое-что недостает. Это “кое-что” — пробельные символы: пробелы, символы табуляции и символы новой строки. Во многих языках лексический анализатор вырезает пробельные символы, чтобы синтаксическому анализатору не надо было иметь с ними дело. В этом заключается большое отличие от трансляции, управляемой разделителями (Delimiter-Directed Translation (213)), в котором пробелы обычно играют ключевую роль в структурировании.

Если пробелы синтаксически значимы — например, символы новой строки в качестве разделителей инструкций или отступы, указывающие блочную структуру, — то лексический анализатор не может просто их игнорировать. Вместо этого он должен генерировать токен некоторого вида, например токен новой строки при применении Newline Separators (339). Однако чаще всего языки, предназначенные для обработки с помощью синтаксически управляемой трансляции, игнорируют пробельные символы. Фактически многие DSL можно сделать вообще без разделителей инструкций. Так, в случае рассматриваемого языка конечного автомата можно смело отбрасывать все пробелы в лексическом анализаторе.

Еще одна вещь, отбрасываемая лексическим анализатором, — комментарии. Комментарии всегда полезно иметь, даже в самом маленьком DSL, и лексический анализатор может легко избавиться от них. Иногда удалять комментарии не нужно, например они могут быть полезны для отладки (в частности, в генерированном коде). В этом случае стоит подумать о том, как присоединить их к элементам семантической модели (Semantic Model (171)).

Я уже говорил, что токены обладают типом и содержимым. На практике они могут включать больше информации. Часто такая информация полезна для диагностики ошибок, например номер строки и позиция символа в строке.

В ходе принятия решений о токенах часто возникает искушение обеспечить тонкую настройку проверки совпадений. При рассмотрении примера контроллера я говорил, что коды событий представляют собой четырехсимвольные последовательности из прописных букв и цифр. Я мог бы рассмотреть возможность использования для них конкретного типа токена, что-то вроде следующего.

```
code: [A-Z 0-9] {4}
```

Проблема в том, что при этом лексический анализатор будет давать неверные токены в случаях наподобие такого.

```
events
  FAIL FZ17
end
```

В этом исходном тексте FAIL будет воспринято лексическим анализатором как код, а не как идентификатор, потому что анализ выполняется с использованием символов, а не общего контекста выражения. Это отличие лучше оставить для рассмотрения синтаксическому анализатору, который имеет достаточно информации, чтобы различать имя и код. Это означает, что проверки соответствия правилу четырех символов должны выполняться позже, в синтаксическом анализаторе. В общем случае лексический анализатор должен быть как можно проще.

Большую часть времени я предпочитаю рассматривать лексический анализатор как работающий с тремя разновидностями токенов.

- *Пунктуация*. Ключевые слова, операторы или другие организующие конструкции (скобки, разделители инструкций). В пунктуации важен тип токена, а не его содержимое. Указанные конструкции также являются фиксированными элементами языка.
- *Предметный текст*. Названия элементов, литературные значения. Для них тип токена, как правило, весьма обобщенный — наподобие “число” или “идентификатор”. Это переменные; каждый сценарий DSL будет иметь свой предметный текст.
- *Игнорируемый текст*. Элементы, которые, как правило, удаляются лексическим анализатором, такие как пробелы и комментарии.

Большинство генераторов синтаксических анализаторов (*Parser Generator* (277)) предоставляют генераторы лексических анализаторов, используя правила с регулярными выражениями наподобие показанных ранее. Тем не менее многие люди предпочитают писать собственные лексические анализаторы. Они достаточно просты, чтобы можно было воспользоваться шаблоном *Regex Table Lexer* (247). В случае рукописных лексических анализаторов вы получаете большую гибкость для более сложных взаимодействий между синтаксическим анализатором и лексическим анализатором, что часто может оказаться полезным.

Одно из взаимодействий синтаксического и лексического анализаторов, которое может оказаться полезным, заключается в поддержке нескольких режимов лексического анализа с возможностью их переключения из синтаксического анализатора. Это позволяет синтаксическому анализатору изменять поведение лексического анализа в определенных точках языка, что способствует применению альтернативной токенизации.

18.1.2. Синтаксический анализатор

После того как вы получили поток токенов, следующей частью синтаксически управляемой трансляции является собственно синтаксический анализ. Поведение анализатора можно разбить на два раздела, которые я буду называть синтаксическим анализом и действиями. Синтаксический анализ получает поток токенов и преобразует их в дерево разбора. Эта работа может быть выполнена полностью на основе одной лишь грамматики, и генератор синтаксических анализаторов (*Parser Generator* (277)) автоматически создает соответствующий инструмент. Раздел действий получает это синтаксическое дерево и выполняет с ним некоторые действия, такие как наполнение семантической модели (*Semantic Model* (171)).

Действия не могут быть получены из грамматики и, как правило, выполняются во время построения синтаксического дерева. Обычно файл грамматики генератора синтаксического анализатора сочетает в себе определение грамматики с дополнительным кодом для указания действий. Часто эти действия выражаются на языке программирования общего назначения, хотя некоторые из них могут быть выражены и на дополнительных DSL.

Здесь я буду игнорировать действия и рассматривать просто синтаксический анализ. Если мы строим синтаксический анализатор с использованием только грамматики и, следовательно, выполняем только синтаксический анализ, то результат анализа будет либо успешным, либо нет. Это указывает, соответствует ли введенный исходный текст грамматике. Часто об этом говорят как о том, *распознает* ли синтаксический анализатор исходный текст.

Вот грамматика для использовавшегося до сих пор исходного текста.

```

declarations : eventBlock commandBlock;
eventBlock   : Event-keyword eventDec* End-keyword;
eventDec     : Identifier Identifier;
commandBlock : Command-keyword commandDec* End-keyword;
commandDec   : Identifier Identifier;

```

А вот сам исходный текст.

```

events
  doorClosed D1CL
  drawOpened D2OP
end

```

Выше я уже показывал, как лексический анализатор разбивает исходный текст на токены.

```

[Event-keyword: "events"]
[Identifier   : "doorClosed"]
[Identifier   : "D1CL"]
[Identifier   : "drawOpened"]
[Identifier   : "D2OP"]
[End-keyword  : "end"]

```

Затем синтаксический анализ получает указанные токены и грамматику и преобразует их в древовидную структуру, показанную на рис. 18.1.

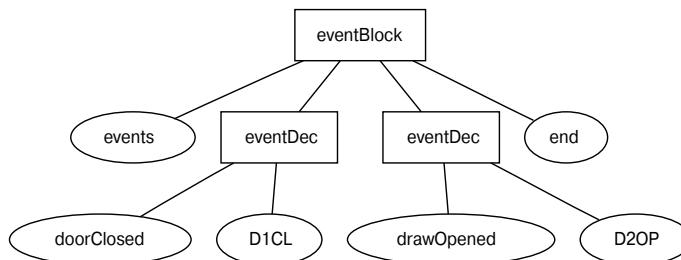


Рис. 18.1. Синтаксическое дерево, полученное в результате анализа блока событий

Как можно видеть, для того чтобы сформировать синтаксическое дерево, синтаксический анализ вводит дополнительные узлы (показанные в виде прямоугольников). Эти узлы определяются грамматикой.

Важно понимать, что любой язык может соответствовать многим грамматикам. Так, в нашем случае можно было бы использовать такую грамматику.

```

eventBlock : Event-keyword eventList End-keyword;
eventList  : eventDec*
eventDec   : Identifier Identifier;

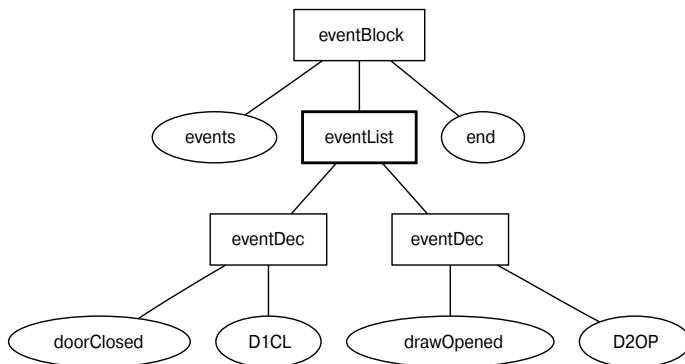
```

Эта грамматика распознает все исходные тексты, распознаваемые предыдущей грамматикой; однако она дает другое синтаксическое дерево, показанное на рис. 18.2.

Итак, в синтаксически управляемой трансляции грамматика определяет, как исходный текст превращается в синтаксическое дерево, и часто можно выбирать различные грамматики в зависимости от того, как мы хотим управлять синтаксическим анализом. Различные грамматики появляются также из-за различий в генераторах синтаксических анализаторов.

До сих пор я говорил о синтаксическом дереве так, как будто это что-то явно производимое синтаксическим анализатором в результате его работы. Однако обычно это не так.

В большинстве случаев вы никогда не получите доступ непосредственно к дереву разбора. Анализатор будет строить части синтаксического дерева и выполнять действия по ходу анализа. Как только это будет сделано для некоторой части синтаксического дерева, эта часть будет уничтожена (исторически это было важно для уменьшения потребления памяти). Если вы строите дерево (*Tree Construction* (289)), то, конечно, в этом случае вы получите полное синтаксическое дерево. Однако при этом, как правило, строится не полное синтаксическое дерево, а его упрощенная версия, именуемая абстрактным синтаксическим деревом.



Rис. 18.2. Альтернативное синтаксическое дерево блока событий

Здесь можно легко запутаться в терминологии. В академической литературе часто используется термин “разбор” (*parsing*) как синоним для синтаксического анализа, называя весь процесс чем-то типа трансляции, интерпретации или компиляции. Генераторы синтаксических анализаторов, как правило, рассматривают синтаксический анализатор как потребитель потока токенов — они говорят о лексическом и синтаксическом анализаторах как об отдельных инструментах. Поскольку такая практика весьма распространена, я точно так же поступаю и в этом разделе.

Еще одна терминологическая путаница связана с терминами “дерево разбора”, “синтаксическое дерево” и “абстрактное синтаксическое дерево”. Я предпочитаю говорить о **синтаксическом дереве**, которое точно отражает анализ с имеющейся грамматикой — по сути, просто “сыре” дерево, получаемое в процессе синтаксического анализа. Термин **абстрактное синтаксическое дерево** я употребляю для обозначения упрощенного синтаксического дерева, получающегося путем отбрасывая ненужных токенов и реорганизации дерева для последующей обработки.

18.1.3. Генерация вывода

Хотя грамматики и достаточно для описания синтаксического анализа, ее достаточно только для распознавания некоторого входа синтаксическим анализатором. Но, как правило, этого мало; мы хотим получить еще и некоторый выход синтаксического анализа. Для получения выхода обычно используется один из трех способов — встроенная трансляция (*Embedded Translation* (305)), построение дерева (*Tree Construction* (289)) и встроенная интерпретация (*Embedded Interpretation* (311)). Все перечисленное для работы требует нечто большее, чем просто грамматику, так что обычно приходится писать дополнительный код для получения вывода.

То, как вы “вплетаете” этот код в анализатор, зависит от того, как вы пишете последний. Если это синтаксический анализатор рекурсивного спуска (*Recursive Descent Parser*

(253)), то действия добавляются в рукописный код. В случае комбинатора синтаксического анализатора (*Parser Combinator* (263)) объекты действий передаются комбинаторам с помощью возможностей вашего языка программирования. При использовании генераторов синтаксических анализаторов (*Parser Generator* (277)) для добавления кода действий в текст файла грамматики используется внешний код (*Foreign Code* (315)).

18.1.4. Семантические предикаты

Синтаксические анализаторы, как написанные вручную, так и сгенерированные, следуют основному алгоритму, который позволяет им распознавать вход на основе грамматики. Однако иногда правила распознавания не могут быть полностью выражены в грамматике. Это наиболее заметно в генераторах синтаксических анализаторов (*Parser Generator* (277)).

Для того чтобы справиться с этими ситуациями, некоторые генераторы синтаксических анализаторов поддерживают семантические предикаты. Семантический предикат представляет собой фрагмент кода общего назначения, который предоставляет логический ответ на вопрос, принимается ли данная продукция грамматики, эффективно переопределя при этом выраженное правилом. Это позволяет синтаксическому анализатору делать то, что невозможно выразить одной лишь грамматикой.

Классическим примером необходимости семантического предиката является синтаксический анализ выражения типа $T(6)$ в C++. В зависимости от контекста это может быть либо вызов функции, либо преобразование типа в стиле конструктора. Чтобы отличить их друг от друга, требуется знать, как было определено T . Вы не можете указать это в контекстно-свободной грамматике, и поэтому для разрешения неоднозначности требуется семантический предикат.

У вас не должна возникнуть необходимость в семантических предикатах в DSL, так как вы должны быть в состоянии определить язык таким образом, чтобы ее избежать. Если вы все же нуждаетесь в них, обратитесь за дополнительной информацией к [23].

18.2. Когда это использовать

Синтаксически управляемая трансляция представляет собой подход, альтернативный трансляции, управляемой разделителями (*Delimiter-Directed Translation* (213)). Основным недостатком синтаксически управляемой трансляции является необходимость изучения анализа в соответствии с грамматикой, в то время как разбиение с помощью разделителей — обычно хорошо знакомый программистам подход. Впрочем, изучение не отнимет много времени, зато вы получите в свое распоряжение технологию, которую гораздо проще использовать с более сложными DSL.

В частности, файл грамматики — сам по себе являющийся DSL — обеспечивает четкую документацию синтаксической структуры обрабатываемого предметно-ориентированного языка, что упрощает развитие синтаксиса DSL с течением времени.

18.3. Дополнительная информация

Синтаксически управляемая трансляция на протяжении десятилетий была важным направлением научных исследований. Стандартная стартовая точка — знаменитая “книга дракона” [1]. Альтернативный маршрут, отклоняющийся от традиционного подхода к преподаванию этого материала, содержится в [23].

Глава 19

Форма Бэкуса-Наура

BNF

Формальное определение синтаксиса языка программирования

```
grammarDef : rule+;
rule       : id ':' altList ';';
altList    : element+ ( '|' element+ )*;
element   : id ebnfSuffix?
          | '(' altList ')'
          ;
ebnfSuffix : '?' | '*' | '+';
id        : 'a'...'z' ('a'...'z' | 'A'...'Z' | '_' | '0'...'9')* ;
```

19.1. Как это работает

Форма Бэкуса-Наура (БНФ (BNF)) (а также расширенная форма Бэкуса-Наура (РБНФ (EBNF))) представляет собой способ написания грамматик для определения синтаксиса языка. БНФ была изобретена в 1960-е годы для описания языка программирования Algol. С тех пор БНФ-грамматики широко использовались как для пояснения, так и для управления синтаксически управляемой трансляцией (Syntax-Directed Translation (229)).

Вы почти наверняка сталкивались с БНФ при изучении нового языка, или, вернее, сталкивались с чем-то в этом роде. По иронии судьбы БНФ, язык для определения синтаксиса, сам по себе стандартного синтаксиса не имеет. Практически всегда, когда вы сталкиваетесь с грамматикой БНФ, в ней есть как очевидные, так и тонкие отличия от всех других грамматик БНФ, которые вы видели раньше. В результате не совсем корректно называть БНФ языком; я склонен рассматривать ее как семейство языков.

Несмотря на то что и синтаксис, и семантика БНФ варьируются в широких пределах, есть и общие элементы. В первую очередь, это концепция описания языка с помощью последовательности правил вывода. В качестве примера рассмотрим такое описание контакта.

```
contact mowler {
  email: fowler@acm.org
}
```

Грамматика описания контактов может выглядеть следующим образом.

```
contact      : 'contact' Identifier '{' 'email:' emailAddress '}' ;
emailAddress : localPart '@' domain ;
```

Здесь грамматика состоит из двух правил. Каждое правило имеет имя и тело. Тело описывает, как можно разложить правило на последовательность элементов. Эти элементы могут быть другими правилами или терминалами. Терминал — это нечто, не являющееся правилом, как, например, литералы `contact` и `}`. Если вы используете грамматику БНФ с синтаксически управляемой трансляцией, ваши терминалы обычно представляют собой типы токенов, получаемых от лексического анализатора. (Я не выполнял дальнейшую декомпозицию правил. В частности, адреса электронной почты могут быть удивительно сложными [25].)

Я уже упоминал, что БНФ проявляется во множестве синтаксических форм. Одна из них — та, которая используется генератором синтаксических анализаторов ANTLR (*Parser Generator* (277)). Вот та же грамматика в форме, существенно более близкой к оригинальной БНФ языка программирования Algol.

```
<contact>      ::= contact <Identifier> { email: <emailAddress> }
<emailAddress> ::= <localPart> @ <domain>
```

В этом случае правила указаны в угловых скобках, литералы — без кавычек, правила завершаются символом новой строки, а не точкой с запятой, и в качестве разделителя между именем правила и телом используется "`: :=`". Вы увидите, что все эти элементы варьируются в разных БНФ, так что не зацикливаитесь на синтаксисе. В этой книге я обычно применяю синтаксис БНФ, который используется в ANTLR (так как я использую ANTLR для всех примеров с генераторами синтаксических анализаторов). Генераторы синтаксических анализаторов обычно применяют именно такой стиль, а не стиль Algol.

Теперь я расширю проблему, рассматривая контакты, которые могут иметь либо адрес электронной почты, либо номер телефона. Так, в дополнение к моему первоначальному примеру мы могли бы добавить следующее.

```
contact rparsons {
  tel: 312-373-1000
}
```

Я могу расширить свою грамматику для распознавания таких конструкций, добавляя в нее альтернативу.

```
contact : 'contact' Identifier '{' line '}';
line   : email | tel ;
email  : 'email:' emailAddress ;
tel    : 'tel:' TelephoneNumber ;
```

Здесь **альтернатива** представлена символом "`|`" в строке правила. Она гласит, что строка может быть разложена либо как `email`, либо как `tel`.

Теперь я извлеку идентификатор, добавив еще одно правило `username`.

```
contact : 'contact' username '{' line '}';
username: Identifier;
line   : email | tel ;
email  : 'email:' emailAddress ;
tel    : 'tel:' TelephoneNumber ;
```

Правило `username` разрешает имя пользователя в единственный идентификатор, но его стоит добавить, чтобы более четко показать намерения грамматики — аналогично извлечению простого метода в императивном коде.

Использование альтернативы в данном контексте достаточно ограниченное — оно позволяет мне иметь только один адрес электронной почты или только один телефон. Но хотя на самом деле альтернативы обладают огромной выразительной силой, в этой книге я все же буду предпочитать выразительности краткость.

19.1.1. Символы множественности (операторы Клини)

Серьезная программа для управления контактами не позволит опуститься до единственного адреса электронной почты или номера телефона. Хотя я и не собираюсь слишком приближаться к реальным приложениям для управления контактами, я все же делаю один шаг в этом направлении. Итак, мои требования таковы: контакт должен иметь имя пользователя, может содержать полное имя, должен включать по крайней мере один адрес электронной почты и, возможно, несколько номеров телефонов. Вот как выглядит соответствующая грамматика.

```
contact : 'contact' username '{' fullname? email+ tel* '}';
username : Identifier;
fullname : QuotedString;
email    : 'email:' emailAddress ;
tel      : 'tel:' TelephoneNumber ;
```

Вы, вероятно, узнали символы множественности, которые используются в регулярных выражениях (их часто называют **операторами Клини**). Они существенно облегчают понимание грамматики.

Рядом с символами множественности, скорее всего, вы увидите и конструкцию группировки, которая позволяет объединить несколько элементов, к которым применяются правила множественности. Так, я мог бы записать приведенную выше грамматику следующим образом.

```
contact : 'contact' Identifier '{'
QuotedString?
('email:' emailAddress) +
('tel:' TelephoneNumber)*
}'
;
```

Я бы не советовал так поступать, потому что подправила лучше отражают намерения автора и гораздо удобнее для чтения. Но иногда подправила добавляют беспорядок, и лучше применять операторы группировки.

Этот пример также показывает, как обычно форматируются большие правила БНФ. Большинство БНФ игнорируют завершения строк, так что размещение каждой логической части правила в собственной строке может сделать сложные правила яснее. В таком случае лучше поставить точку с запятой в отдельной строке, чтобы обозначить окончание правила. Этот вид форматирования я встречал чаще всего, и сам предпочитаю этот стиль, когда правило становится слишком сложным, чтобы легко помещаться в одной строке.

Добавление символов множественности — это обычно именно то, что отличает РБНФ (расширенную БНФ) и базовую БНФ. Однако, как обычно, и здесь налицоует терминологическая неразбериха. Когда люди говорят “БНФ”, то это может означать как базовые БНФ (т. е. не РБНФ), так и более широко понимаемые БНФ-образности (в том числе РБНФ). В этой книге, когда я имею в виду БНФ без символов множественности, я говорю о базовых БНФ, а когда я говорю БНФ, то подразумеваю любой БНФ-подобный язык, в том числе РБНФ.

Показанные здесь символы множественности наиболее распространены, в частности, в генераторах синтаксических анализаторов (Parser Generator (277)). Однако есть и другой вид группировки, который использует фигурные и квадратные скобки.

```
contact : 'contact' username '{' [fullname] email {email} {tel} '}';
username : Identifier;
fullname : QuotedString;
email    : 'email:' emailAddress ;
tel      : 'tel:' TelephoneNumber ;
```

При использовании фигурных и квадратных скобок `? заменяется на [..], а * на { .. }.` Здесь нет замены для `+`, так что можно заменить `foo+` на `foo { foo }`. Этот стиль довольно распространен в грамматиках, ориентированных на чтение человеком, и используется в стандарте ISO для РБНФ (ISO/IEC 14977). Однако большинство генераторов синтаксических анализаторов предпочитают форму с применением регулярных выражений. В своих примерах я буду использовать стиль регулярных выражений.

19.1.2. Другие полезные операторы

Есть несколько других операторов, о которых я должен упомянуть, поскольку я использую их в примерах в этой книге, и с которыми вы, возможно, сталкивались ранее.

Так как я активно использую в этой книге для своих грамматик ANTLR, я использую его оператор `~`, который я называю “оператор до”. **Оператор до** соответствует всему, находящемуся до элемента, следующего за `~`. Так что, если вы хотите выразить соответствие всем символам до фигурной закрывающей скобки, не включая ее, можете использовать шаблон `~' } '`. Если у вас нет такого оператора, эквивалентное регулярное выражение имеет вид наподобие `[^}]*`.

Большинство подходов к синтаксически управляемой трансляции (Syntax-Directed Translation (229)) отделяют лексический анализ от синтаксического. Лексический анализ также можно определить в стиле рассмотренных правил, но здесь обычно имеются тонкие, но важные различия, как, например, какие виды операторов и их комбинации допустимы. Лексические правила, скорее всего, будут близки к регулярным выражениям, хотя бы потому, что регулярные выражения часто применяются для лексического анализа в силу использования ими конечных автоматов, в отличие от стековой машины синтаксического анализатора (см. раздел “Регулярные, контекстно-свободные и контекстно-зависимые грамматики” на с. 112).

Важным оператором в лексическом анализе является оператор диапазона `" .. "`, который используется для определения диапазона символов, таких как буквы в нижнем регистре `'a'..'z'`. Распространенным правилом для идентификаторов является следующее.

```
Identifier:
  ('a'..'z' | 'A'..'Z')
  ('a'..'z' | 'A'..'Z' | '0'..9' | '_')*
;
```

Так определяются идентификаторы, которые начинаются с буквы — маленькой или заглавной, а затем могут содержать буквы, цифры или символ подчеркивания. Диапазоны имеют смысл только в лексических правилах, но не в синтаксических. Они традиционно ориентированы на набор символов ASCII, что затрудняет поддержку идентификаторов на языках, отличных от английского.

19.1.3. Грамматики, разбирающие выражения

Большинство БНФ-грамматик, с которыми приходится иметь дело — контекстно-свободные. Однако имеется еще один стиль грамматик, получивший распространение в последнее время: грамматики, разбирающие выражения (parsing expression grammar). Самая большая разница между грамматиками, разбирающими выражения, и контекстно-свободными грамматиками в том, что в первых имеются упорядоченные альтернативы. В контекстно-свободной грамматике вы пишете следующее.

```
contact : email | tel;
```

Это означает, что контакт может быть электронной почтой или номером телефона. Порядок, в котором вы записываете эти две альтернативы, не влияет на интерпретацию. В большинстве случаев это хорошо, но иногда неупорядоченные альтернативы приводят к двусмысленности.

Рассмотрим случай, когда вы хотите распознавать последовательность из десяти цифр как номер телефона в США, но при этом прочие последовательности рассматривать как просто неструктурированный номер телефона. Попробуйте написать грамматику наподобие такой.

```
tel : us_number | raw_number ;
raw_number
  : (DIGIT | SEP)+;
us_number
  : (us_area_code | '(' us_area_code ')') SEP? us_local;
us_area_code
  : DIGIT DIGIT DIGIT;
us_local
  : DIGIT DIGIT DIGIT SEP? DIGIT DIGIT DIGIT DIGIT;
DIGIT : '0'..'9';
SEP   : ('-' | '.') ;
```

Эта грамматика неоднозначна; когда она получает на вход что-то вроде “312-373”1000, то и `us_number`, и `raw_number` могут соответствовать этому входу. **Упорядоченные альтернативы** обеспечивают упорядоченное рассмотрение правил, и при этом используется то правило, которое подошло первым. Распространенный синтаксис для упорядоченных альтернатив использует символ `/`, так что поэтому правило `tel` должно гласить следующее.

```
tel: us_number / raw_number ;
```

(Нужно отметить, что хотя ANTLR и использует неупорядоченные альтернативы, они действуют подобно упорядоченным. При такого рода неоднозначности ANTLR выдаст предупреждение и будет работать с первой подошедшей альтернативой.)

Символ	Значение	Пример
	Альтернатива	email tel
*	Нуль или более повторений	tel*
+	Одно или более повторений	email+
?	Необязательно	fullname?
~	До	~ '}'
..	Диапазон	'0'..'9'
/	Упорядоченная альтернатива	us_tel / raw_tel

19.1.4. Преобразование РБНФ в БНФ

Символы множественности упрощают следование БНФ. Однако они не увеличивают выразительную силу БНФ. Грамматика РБНФ, использующая символы множественности, может быть заменена эквивалентной грамматикой базовой БНФ. Время от

времени это важно, так как некоторые генераторы синтаксических анализаторов (Parser Generator (277)) используют для своих грамматик базовые БНФ.

В качестве примера я вновь воспользуюсь грамматикой контактов.

```
contact : 'contact' username '{' fullname? email+ tel* '}';
username : Identifier;
fullname : QuotedString;
email    : 'email:' emailAddress ;
tel      : 'tel:' TelephoneNumber ;
```

Ключом к преобразованию являются альтернативы. Начнем со спецификатора “необязательности”. Так, `foo?` ; можно заменить на `foo | ;` (т.е. `foo` или ничего).

```
contact : 'contact' username '{' fullname email+ tel* '}';
username : Identifier;
fullname : /* необязательно */ | QuotedString ;
email    : 'email:' emailAddress ;
tel      : 'tel:' TelephoneNumber ;
```

Здесь добавлен комментарий, чтобы было понятнее, что я делаю. Конечно, различные инструменты используют различные синтаксисы комментариев, но я буду следовать соглашениям языка программирования C. Я не люблю использовать комментарии, если их можно заменить чем-то иным из применяемого языка программирования, но если это невозможно, я пишу их без колебаний, как в этом случае.

Если родительское предложение простое, можно разместить альтернативы в нем. Так `a : b?` с превратится в `a: c | b c`. Однако, если есть несколько необязательных элементов, можно получить комбинаторный взрыв, в эпицентре которого, как и в эпицентре большинства взрывов, лучше не оказываться...

Для преобразования повторения можно снова воспользоваться альтернативами, в данном случае — с рекурсией. Это весьма характерно для правил — применение рекурсии, т.е. самого правила в своем теле. Итак, `x : y*`; можно заменить на `x : y x | ;`. Применив этот подход к номеру телефона, получим следующее.

```
contact : 'contact' username '{' fullname email+ tel '}' ;
username : Identifier;
fullname : /* необязательно */ | QuotedString ;
email    : 'email:' emailAddress ;
tel      : /* несколько */ | 'tel:' TelephoneNumber tel;
```

Это основной способ обработки рекурсии. Работая с рекурсивным алгоритмом, необходимо рассмотреть два случая: случай терминала и случай, который включает рекурсивный вызов. В данной ситуации есть альтернатива для каждого из случаев: случай терминала пуст, а рекурсивный случай добавляет один элемент.

После ввода рекурсии, как в данном случае, часто приходится принимать решение о том, какой делать рекурсию — правой или левой, т.е. заменять `x : y*`; на `x : y x |` или на `x : x y |`. Обычно в документации к синтаксическому анализатору говорится, какой тип рекурсии предпочтителен в силу применяемого для синтаксического анализа алгоритма. Например, нисходящий синтаксический анализатор не в состоянии обрабатывать левую рекурсию вообще, а Yacc, хотя и работает с любой рекурсией, предпочитает правую.

Последний спецификатор множественности — `"+"`. Он похож на `*`, но терминальный случай не пустой, а содержит один элемент, поэтому `x : y+` можно заменить на `x : y | x y` (или `x: y | y x`, чтобы избежать левой рекурсии). Это приводит к следующему коду.

```

contact      : 'contact' username '{' fullname email tel '}';
username     : Identifier;
fullname     : /* необязательно */ | QuotedString ;
email        : singleEmail | email singleEmail;
singleEmail   : 'email:' emailAddress ;
tel          : /* несколько */ | 'tel:' TelephoneNumber tel;

```

Поскольку выражение для электронной почты в правиле `email` используется два раза, я выделил его в отдельное правило. Такое введение промежуточных правил часто необходимо при преобразовании в базовую БНФ; это также необходимо делать при наличии групп.

Теперь у меня есть грамматика контакта в базовой БНФ. Она работает так же хорошо, но следовать ей гораздо сложнее. Я не только теряю спецификаторы множественности, но и ввожу дополнительные подправила только для того, чтобы обеспечить корректную работу рекурсии. Поэтому при прочих равных условиях я предпоютаю РБНФ, но если необходима базовая БНФ, я без колебаний применяю все описанные методики.

РБНФ	$x : y?$	$x: y^*$	$x: y^+$
Базовая БНФ	$x: /* \text{необязательно} */$ y	$x: /* \text{несколько} */$ $y x$	$x: y$ $y x$

19.1.5. Действия

БНФ предоставляет способ определения синтаксической структуры языка, и генераторы синтаксических анализаторов (Parser Generator (277)) обычно используют БНФ для управления работой анализатора. Однако одной БНФ недостаточно. Она дает достаточно информации для создания синтаксического дерева, но не столько, чтобы достичь чего-то полезного с помощью абстрактного синтаксического дерева или решать такие задачи, как встроенная трансляция (Embedded Translation (305)) или встроенная интерпретация (Embedded Interpretation (311)). Поэтому обычно используется подход с размещением в БНФ кода действий.

Не все генераторы синтаксических анализаторов используют код действий. Еще один подход заключается в предоставлении отдельного DSL для чего-то наподобие Tree Construction (289).

Суть кода действий заключается в размещении фрагментов внешнего кода (Foreign Code (315)) в определенных местах грамматики. Эти фрагменты выполняются при распознавании синтаксическим анализатором соответствующих частей грамматики. Рассмотрим такую грамматику.

```

contact : 'contact' username '{' email? tel? '}';
username: ID;
email   : 'email:' EmailAddress {log("got email")};;
tel     : 'tel:' TelephoneNumber;

```

В этом случае, как только соответствующее предложение распознается в процессе синтаксического анализа, выполняется регистрация электронной почты в журнале. Подобный механизм может использоваться для отслеживания моментов, когда мы имеем дело с электронной почтой. В коде действия можно делать все что угодно, так что мы можем также добавить информацию в структуры данных.

Коду действий часто приходится обращаться к элементам, распознаваемым в процессе синтаксического анализа. Приятно записать в журнал тот факт, что обнаружено письмо, но было бы приятнее записать еще и адрес электронной почты. Чтобы сделать это,

нужно обратиться к токену распознанного адреса электронной почты. Генераторы синтаксических анализаторов делают это по-разному. Классический Yacc обращается к токенам посредством специальных переменных, проиндексированных положением элемента. Так что к токену адреса электронной почты можно обращаться как к \$2 (\$1 относится к токену 'email:'). Позиционные ссылки очень чувствительны к изменениям в грамматике, так что более распространенным подходом в современных генераторах синтаксических анализаторов являются метки элементов. Вот как это делается в ANTLR.

```
contact : 'contact' username '{' email? tel? '}';
username: ID;
email   : 'email:' e=EmailAddress {log("got email " + $e.text);};
tel     : 'tel:' TelephoneNumber;
```

В ANTLR ссылка \$e относится к элементу, помеченному в грамматике как e=. Так как этот элемент является токеном, я использую атрибут text, чтобы получить соответствующий текст. (Я могу также получить такую информацию, как тип токена, номер строки и т.д.)

Для разрешения таких ссылок генераторы синтаксических анализаторов пропускают код действий через шаблонную систему, которая заменяет выражения типа \$e соответствующими значениями. Фактически ANTLR идет еще дальше. Атрибуты наподобие text не ссылаются на поля или методы непосредственно — ANTLR выполняет дальнейшие подстановки для получения правильной информации.

Аналогично ссылке на токен можно сослаться и на правило.

```
contact : 'contact' username '{' e=email? tel? '}';
      {log("email " + $e.text); }
      ;
username: ID;
email   : 'email:' EmailAddress ;
tel     : 'tel:' TelephoneNumber;
```

Здесь в журнал будет записываться полный текст, соответствующий правилу ("email: fowler@acm.org"). Часто подобный возврат объекта некоторого правила не слишком полезен, в особенности в случае больших правил. В результате генераторы синтаксических анализаторов обычно позволяют определить, что возвращается правилом при его распознавании. В ANTLR это делается путем определения типа возвращаемого значения и переменной для правила, которая затем и возвращается.

```
contact : 'contact' username '{' e=email? tel? '}';
      {log("email " + $e.result); }
      ;
username: ID;
email   returns [EmailAddress result]
      : 'email:' e=EmailAddress
      {$result = new EmailAddress($e.text); }
      ;
tel     : 'tel:' TelephoneNumber;
```

И правило можно вернуть все что угодно, а затем обратиться к тому, что было возвращено в родительском правиле. (ANTLR позволяет определить несколько возвращаемых значений.) Эта возможность в сочетании с кодом действия является исключительно важной. Часто правило, которое дает лучшую информацию о значении, не является лучшим с точки зрения решения, что же делать с этими данными. Передав данные по стеку правил, можно захватывать информацию на низком уровне синтаксического анализа, а работать с ней — на более высоком уровне. Без этого вам придется использовать много

переменных контекста (Context Variable (187)), что приведет к быстрому существенному загрязнению кода.

Коды действий могут использоваться во всех трех стилях — во встроенной трансляции, встроенной интерпретации и в построении дерева. Однако особый стиль кода в построении дерева годится для подхода, использующего для описания формирования результирующего синтаксического дерева другой DSL (см. раздел “Использование синтаксиса построения дерева ANTLR” на с. 292).

Положение кода действия в грамматике определяет, когда он будет выполняться. Так, правило `parent : first {log("hello");} second` приведет к вызову метода `log` после того, как будет распознано подправило `first`, но до того, как будет распознано подправило `second`. Чаще всего проще помещать код действия в конец правила, но в отдельных случаях нужно располагать его в средине. Иногда трудно понять последовательность выполнения действий, потому что она зависит от алгоритма синтаксического анализатора. Синтаксические анализаторы рекурсивного спуска, как правило, довольно легко понять; работа восходящих синтаксических анализаторов часто сбивает с толку. Возможно, вам придется детально изучить работу своего синтаксического анализатора, чтобы понять, когда именно будут выполнены те или иные действия.

Одной из опасностей кодов действий является то, что в конечном итоге в них можно разместить слишком много кода. Если вы так поступите, грамматика станет трудной для понимания, и вы потеряете большую часть преимуществ ее возможностей самодокументирования. Поэтому я настоятельно рекомендую вам применять при работе с кодами действий шаблон `Embedment Helper` (537).

19.2. Когда это использовать

Используйте БНФ всякий раз, когда работаете с генераторами синтаксических анализаторов (Parser Generator (277)), так как эти инструменты применяют БНФ-грамматики для определения того, как выполнять синтаксический анализ. Она также очень полезна в качестве неформального инструмента, помогающего визуализировать структуру DSL или передавать синтаксические правила языка другим людям.

Глава 20

Лексический анализатор на основе таблицы регулярных выражений

Автор — Ребекка Парсонс

Regex Table Lexer

*Реализация лексического анализатора с помощью списка
регулярных выражений*

Шаблон	Тип токена
<code>^events</code>	K_EVENT
<code>^end</code>	K_END
<code>^(\\w)+</code>	IDENTIFIER
<code>^(\\s)+</code>	WHITESPACE

Синтаксические анализаторы работают, в первую очередь, со структурой языка, в частности со способом объединения компонентов языка. Большинство фундаментальных компонентов языка, таких как ключевые слова, числа и имена, могут быть легко распознаны синтаксическим анализатором. Однако в общем случае мы выделяем этот этап в лексический анализатор. Используя отдельный проход для распознавания этих терминальных символов, можно существенно упростить построение синтаксического анализатора.

Непосредственная реализация лексического анализатора относительно проста. Лексические анализаторы прочно обосновались в пространстве регулярных языков, что означает возможность использования для их реализаций стандартных API регулярных выражений. В лексическом анализаторе на основе таблицы регулярных выражений мы используем список регулярных выражений, каждое из которых связано с определенным

терминальным символом. Мы сканируем входной поток, сопоставляем отдельные его части с соответствующими регулярными выражениями и генерируем поток токенов. Именно этот поток токенов и подается на вход синтаксического анализатора.

20.1. Как это работает

При использовании синтаксически управляемой трансляции (Syntax-Directed Translation (229)) распространено выделение лексического анализа в собственную студию. Взгляните на этот шаблон еще раз, чтобы понять, почему мы отделяем лексический анализ, вспомнить некоторые концептуальные вопросы, связанные с лексическим анализом, и то, как лексический и синтаксический анализ вписывается в общую, более широкую картину. При рассмотрении же данного шаблона мы сосредоточимся на реализации простого лексического анализатора.

Базовый алгоритм достаточно прост. Лексический анализатор сканирует входную строку с самого начала, по ходу сканирования выполняя сопоставление токенам и изымая из входного потока соответствующие символы. Давайте начнем с очень простого и немного смешного примера. Есть два символа, которые нужно распознавать, — строки Hello и Goodbye. Регулярными выражениями для этих символов являются соответственно `^Hello` и `^Goodbye`. Оператор `^` необходим для того, чтобы привязать регулярное выражение к началу строки. Назовем наши токены HOWDY и BYEBYE, просто чтобы название токена отличалось от лексемы. Давайте рассмотрим, как основной алгоритм работает с входной строкой

```
HelloGoodbyeHelloHelloGoodbye
```

Регулярное выражение для Hello соответствует началу строки, поэтому мы генерируем токен HOWDY и продвигаем указатель в строке к первому символу G. Алгоритм возвращается в начало списка регулярных выражений, так как имеет значение его упорядоченность. Таким образом, первой будет выполняться проверка строки, начинающейся с буквы G, на соответствие регулярному выражению Hello. Эта проверка, конечно, неудачна. Таким образом, мы испытываем следующее регулярное выражение из списка, а именно — Goodbye, и это испытание успешно. Мы добавляем токен BYEBYE к нашему выходному потоку токенов, переносим указатель строки ко второму H и продолжаем работу. Окончательный выходной поток для приведенного выше предложения — HOWDY, BYEBYE, HOWDY, HOWDY, BYEBYE.

Порядок проверки шаблонов важен для правильной обработки таких вещей, как ключевые слова. В грамматике конечного автомата, например, ключевые слова соответствуют правилам для идентификаторов. Поэтому, чтобы токены для ключевых слов были правильно определены, они должны проверяться первыми. Выбор соответствующих токенов представляет собой для лексического анализатора проектное решение. В грамматике конечного автомата мы не пытаемся проводить различие между кодами и именами, используя единый токен для обоих случаев. Этот выбор связан с тем, что лексический анализатор не имеет знаний о контексте для того, чтобы суметь определить, что четырехсимвольное слово является токеном, если код в этой позиции недопустим. Часто, однако, в набор токенов входят ключевые слова, имена, числа, знаки препинания и операторы.

Мы инстанцируем конкретный лексический анализатор, указывая “распознаватели”, т.е. данные для выполнения распознавания, и используем таблицу или список, чтобы их упорядочить. Каждый распознаватель содержит тип токена, регулярное выражение для его распознавания и логическое значение, указывающее, должен ли этот токен быть по-

мещен в выходной поток. Тип токена используется просто для идентификации класса токена синтаксического анализатора. Логическое значение позволяет обрабатывать, например, семантически незначащие пробелы и комментарии. Хотя эти строки находятся во входном потоке и должны быть обработаны лексическим анализатором, мы не передаем соответствующие токены синтаксическому анализатору. Повторное последовательное сканирование таблицы обеспечивает упорядочение проверок на соответствие, и, кроме того, таблица распознавателей позволяет легко вводить дополнительные типы токенов.

Процесс распознавания отдельных токенов проходит по таблице распознавателей; если найдено соответствие, из входной строки удаляются соответствующие символы, а токен — в предположении, что установлен логический флаг вывода — отправляется в выходной поток. Мы заполняем поле `tokenValue` выходного токена независимо от того, является ли это необходимым. Как правило, значения токенов нужны только для идентификаторов, чисел, а иногда и операторов, но такой глобальный подход избавляет нас от второго логического флага и упрощает код. Основной метод сканера постоянно вызывает единственный метод сопоставления регулярному выражению, чтобы убедиться в успешности распознавания. После успешного распознавания входной строки буфер токенов передается для обработки синтаксическому анализатору.

Чтобы помочь синтаксическому анализатору диагностировать ошибки, можно добавить в токен информацию о том, где он находился во входном потоке символов, например, указав номер строки и столбца.

20.2. Когда это использовать

Хотя имеются генераторы лексических анализаторов, такие как Lex, использовать их с учетом распространенности API регулярных выражений нет нужды. Единственным исключением является использование в качестве генератора синтаксических анализаторов (Parser Generator (277)) ANTLR, так как лексический и синтаксический анализ более тесно интегрирован в этот инструмент.

Описанная здесь реализация очевидна для лексического анализа. Ее производительность напрямую зависит от специфики используемого API регулярных выражений. Не использовать лексический анализатор на основе таблицы регулярных выражений я бы предложила только в том случае, если недоступны приемлемые API регулярных выражений.

Учитывая простой синтаксис многих DSL, возможны ситуации, когда такой подход будет использоваться для распознавания языка полностью. Если язык является регулярным, этот подход применим и для его синтаксического анализатора.

20.3. Лексический анализатор контроллера мисс Грант (Java)

Лексический анализатор для грамматики конечного автомата достаточно типичен. Мы выявляем токены для ключевых слов, знаков препинания и идентификаторов. У нас также есть тип токена для комментариев и пробелов, с которыми имеет дело только лексический анализатор. Мы используем для указания шаблонов и проверки соответствий API `java.util.regex`. На вход лексического анализатора поступает анализируемый сценарий DSL, а выход представляет собой буфер токенов, включающий типы токенов и связанные с ними значения. Этот буфер токенов становится входом для синтаксического анализатора.

Реализация разделяется на спецификацию распознаваемых токенов и алгоритм самого лексического анализа. Такой подход позволяет легко добавлять дополнительные

типы токенов в лексический анализатор. Мы используем для указания типа токена перечисление с атрибутами, указывающими соответствующее регулярное выражение и логическое значение, управляющее выводом токена. Это подходит Java, но с тем же успехом можно использовать и более традиционные объекты. Главное, что типы токенов должны быть легко доступны для использования в самом анализаторе.

```
class ScannerPatterns...
public enum TokenType {
    TT_EVENT("^events", true),
    TT_RESET("^resetEvents", true),
    TT_COMMANDS("^commands", true),
    TT_END("^end", true),
    TT_STATE("^state", true),
    TT_ACTIONS("^actions", true),
    TT_LEFT("^\\{", true),
    TT_RIGHT("^\\}", true),
    TT_TRANSITION("^=>", true),
    TT_IDENTIFIER("^([\\w]+", true),
    TT_WHITESPACE("^([\\s]+", false),
    TT_COMMENT("^\\\\\\\\(.)*$", false),
    TT_EOF("^EOF", false);
}

private final String regexPattern;
private final Boolean outputToken;

TokenType(String regexPattern, Boolean output) {
    this.regexPattern = regexPattern;
    this.outputToken = output;
}
}
```

В лексическом анализаторе мы инстанцируем таблицу объектов распознавания со скомпилированными распознавателями, объединенными с их типами токенов и логическими значениями.

```
class ScannerPatterns...
public static ArrayList<ScanRecognizer> LoadPatterns() {
    Pattern pattern;
    for (TokenType t : TokenType.values()) {
        pattern = Pattern.compile(t.regexPattern);
        patternMatchers.add(new ScanRecognizer(t, pattern, t.outputToken));
    }
    return(patternMatchers);
}
```

Для нашего лексического анализатора мы определяем класс. Переменные экземпляра лексического анализатора включают распознаватели, входную строку и список выходных токенов.

```
class StateMachineTokenizer...
private String scannerBuffer;
private ArrayList<Token> tokenList;
private ArrayList<ScanRecognizer> recognizerPatterns;
```

Основной цикл обработки в лексическом анализаторе представляет собой цикл `while`.

```
class StateMachineTokenizer...
while (parseInProgress) {
    Iterator<ScanRecognizer> patternIterator =
        recognizerPatterns.iterator();
    parseInProgress = matchToken(patternIterator);
}
```

Этот цикл постоянно выполняет сопоставление, пока не будет исчерпан буфер или для оставшегося буфера не останется возможных вариантов.

Метод `matchToken` по порядку проходит по распознавателям и пытается сопоставлять токены по одному.

```
private boolean matchToken(Iterator<ScanRecognizer>
                           patternIterator) {
    boolean tokenMatch;
    ScanRecognizer recognizer;
    Pattern pattern;
    Matcher matcher;
    boolean result;
    tokenMatch = false;
    result = true;

    do {
        recognizer = patternIterator.next();
        pattern = recognizer.tokenPattern;
        matcher = pattern.matcher(scannerBuffer);
        if (matcher.find()) {
            if (recognizer.outputToken) {
                tokenList.add(new Token(recognizer.token,
                                         matcher.group()));
            }
            tokenMatch = true;
            scannerBuffer = scannerBuffer.substring(matcher.end());
        }
    } while (patternIterator.hasNext() && (!tokenMatch));

    if ((!tokenMatch) ||
        (matcher.end() == scannerBuffer.length())) {
        result = false;
    }
    return result;
}
```

Если сопоставление завершается успешно, мы проходим во входном буфере в точку, следующую за сопоставленным текстом, в данном случае получая конечную точку с помощью вызова `matcher.end()`. Мы проверяем значение логического флага для распознанного токена и, если он позволяет, генерируем этот токен. Если совпадение не найдено, объявляется о неудачном завершении операции. Метод `find` из API `regex` выполняет поиск соответствия путем сканирования до конца строки. Если при сканировании оставшейся части строки соответствия не выявлены, то весь лексический анализ считается неудачным.

Внешний цикл работает до тех пор, пока внутренний цикл находит токены, проходя по входной строке до ее конца. Значение итератора для каждой итерации внутреннего цикла сбрасывается, гарантируя тем самым проверку всех шаблонов токенов при каждом проходе. Результатом анализа является буфер токенов, в котором каждый токен имеет свой тип и строковое значение, для которого лексическим анализатором было выявлено соответствие данному токену.

Глава 21

Синтаксический анализатор на основе рекурсивного спуска

Автор — Ребекка Парсонс

Recursive Descent Parser

*Создание исходящего синтаксического анализатора
с использованием управления потоком для операторов грамматики
и рекурсивных функций для распознавания нетерминалов*

```
boolean eventBlock() {
    boolean parseSuccess = false;
    Token t = tokenBuffer.nextToken();
    if (t.isTokenType(ScannerPatterns.TokenTypes.TT_EVENT)) {
        tokenBuffer.popToken();
        parseSuccess = eventDecList();
    }
    if (parseSuccess) {
        t = tokenBuffer.nextToken();
        if (t.isTokenType(ScannerPatterns.TokenTypes.TT_END)) {
            tokenBuffer.popToken();
        }
        else {
            parseSuccess = false;
        }
    }
    return parseSuccess;
}
```

Многие DSL как языки довольно просты. Хотя гибкость внешних языков является привлекательным фактором, применение генератора синтаксических анализаторов (Parser Generator (277)) добавляет в проект новые инструменты и языки, усложняя тем самым процесс сборки.

Синтаксический анализатор на основе рекурсивного спуска поддерживает гибкость внешнего DSL без участия генератора синтаксических анализаторов. Синтаксический анализатор на основе рекурсивного спуска может быть реализован на любом выбранном языке программирования общего назначения. Он использует операторы управления по-

током для реализации различных операторов грамматики. Отдельные методы или функции реализуют правила синтаксического анализа для различных нетерминальных символов грамматики.

21.1. Как это работает

Как и в других реализациях, мы вновь разделяем лексический и синтаксический анализ. Синтаксический анализатор на основе рекурсивного спуска получает поток токенов от лексического анализатора, такого как лексический анализатор на основе таблицы регулярных выражений (Regex Table Lexer (247)).

Базовая структура синтаксического анализатора на основе рекурсивного спуска довольно проста. Для каждого нетерминального символа грамматики имеется свой метод. Этот метод реализует различные правила вывода, связанные с соответствующим нетерминалом. Метод возвращает логическое значение, которое представляет собой результат сопоставления правилу. Неудача на любом уровне передается обратно по стеку вызовов. Каждый метод работает с буфером токенов, перемещая указатель текущей позиции по мере выявления соответствия некоторой части предложения правилу.

Поскольку в грамматике относительно мало операторов грамматики (последовательности, альтернативы и повторения), эти методы реализации используют небольшое количество шаблонов. Давайте начнем с обработки альтернатив, которая использует условные конструкции. Так, для фрагмента грамматики

```
Файл грамматики...
С : А | В
```

соответствующая функция будет иметь следующий вид.

```
boolean С ()
if (А())
then true
else if (В())
then true
else false
```

Такая реализация просто проверяет одну альтернативу за другой, работая скорее как упорядоченные альтернативы (с. 240). Если вам действительно необходимо допустить неопределенность, вносимую неупорядоченными альтернативами, возможно, вам следует прибегнуть к генератору синтаксических анализаторов (Parser Generator (277)).

После успешного вызова А() указатель буфера токенов будет передвинут на первый токен, следующий за теми, которые соответствовали А. Если же вызов А() терпит неудачу, буфер токенов остается неизменным.

Оператор последовательности грамматики реализуется с помощью вложенных инструкций if, так как, если любой из методов неудачен, обработка завершается. Таким образом, реализация

```
Файл грамматики...
С : А В
```

будет иметь следующий вид.

```
boolean С ()
if (А())
then if (В())
then true
else false
else false
```

Оператор необязательности реализуется немного иначе.

Файл грамматики...

C : A?

Мы должны попытаться распознать токены, соответствующие нетерминалу A, но глобально здесь нас не может постичь неудача. Если мы находим соответствие A, мы возвращаем значение `true`. Если мы не находим соответствие A, мы все равно возвращаем значение `true`, ведь наличие A необязательно. Таким образом, реализация имеет следующий вид.

```
boolean C ()  
A()  
true
```

Если проверка соответствия правилу A завершилась неудачей, буфер токенов остается в том же состоянии, что и на входе в C. Если же соответствие найдено, указатель буфера соответствующим образом перемещается. Вызов C успешен в любом случае.

Оператор повторения имеет две основные формы: нуль или более экземпляров ("*") и один или более экземпляров ("+"). Реализация последней формы

Файл грамматики...

C : A+

использует следующий шаблон.

```
boolean C ()  
if (A())  
then while (A())  
{ }  
true  
else  
false
```

Этот код сначала проверяет, есть ли в наличии хотя бы один нетерминал A. Если это так, функция пытается искать другие вхождения A, но, сколько бы их ни было найдено, она всегда будет возвращать значение `true`, потому что обязательное условие — наличие по крайней мере одного вхождения — выполнено. Код для списка, в котором допускается наличие нуля экземпляров, получается из приведенного выше простым удалением внешнего if и возвратом `true` в любом случае.

```
boolean C ()  
while (A())  
{ }  
true
```

В следующей таблице резюмированы рассмотренные выше реализации операторов грамматики (с использованием псевдокода).

Правило грамматики	Реализация
A B	<pre>if (A()) then true else if (B()) then true else false</pre>

Окончание таблицы

Правило грамматики	Реализация
A B	<pre>if (A()) then if (B()) then true else false else false</pre>
A?	<pre>A(); true</pre>
A*	<pre>while A(); true</pre>
A+	<pre>if (A()) then while (A()); else false</pre>

Мы используем вспомогательные функции в том же стиле, что и в других разделах, чтобы отделять действия от синтаксического анализа. В рекурсивном спуске возможны как построение дерева (Tree Construction (289)), так и встраиваемая трансляция (Embedded Translation (305)).

Чтобы сделать данный подход максимально ясным, методы, реализующие правила вывода, должны иметь согласующееся поведение. Наиболее важное правило касается управления буфером токенов. Если метод обнаруживает искомое соответствие, текущее положение в строке токенов перемещается в точку, следующую непосредственно за распознанной последовательностью токенов. Например, если распознано одно ключевое слово, указатель текущей позиции перемещается на один токен. Если же соответствие не обнаружено, текущая позиция остается той же, что и до вызова метода. Это очень важно! В начале функции мы должны сохранить текущую позицию во входном буфере на тот случай, если первая часть последовательности (A в приведенном выше примере) соответствует правилу, а соответствие второй части — B — не выполняется. Такое управление буфером токенов позволяет корректно обработать альтернативы в грамматике.

Другое важное правило касается наполнения семантической модели или синтаксического дерева. Насколько это возможно, каждый метод должен управлять собственной частью модели или создавать собственные элементы синтаксического дерева. Естественно, любые действия должны предприниматься только в том случае, когда обнаружено и подтверждено полное соответствие правилу. Как и управление буфером токенов, действия должны быть отложены до полного завершения всей последовательности токенов.

Одна из жалоб на генераторы синтаксических анализаторов заключается в том, что они требуют от разработчиков знания грамматик языков. Хотя это и правда, что синтаксис операторов грамматики не проявляется в реализации рекурсивного спуска, грамматика сосредоточена в методах. Изменение методов приводит к изменению грамматики. Разница заключается не в наличии или отсутствии грамматики, а в том, как она выражается.

21.2. Когда это использовать

Самой сильной стороной рекурсивного спуска является его простота. Как только вы поймете базовый алгоритм и научитесь обращаться с различными операторами грамматики, написание синтаксического анализатора на основе рекурсивного спуска станет для вас очень простой программной задачей. После этого синтаксический анализатор будет представлять собой обычный класс вашей системы. Тестирование в этом случае выполняется точно так, как и для других систем; в частности, модульное тестирование имеет больше смысла, когда единицей тестирования является метод. Наконец, так как анализатор — это просто программа, достаточно легко разобраться в его поведении и выполнить отладку. Синтаксический анализатор на основе рекурсивного спуска является прямой реализацией алгоритма синтаксического анализа, что облегчает отслеживание потока через синтаксический анализатор.

Наиболее серьезным недостатком синтаксического анализатора на основе рекурсивного спуска является отсутствие явного представления грамматики. При кодировании алгоритма рекурсивного спуска вы теряете ясное представление грамматики, которая остается только в документации или комментариях. И комбинатор синтаксических анализаторов (*Parser Combinator* (263)), и генератор синтаксических анализаторов (*Parser Generator* (277)) включают явные инструкции грамматики, что облегчает ее понимание и развитие.

Другая проблема синтаксических анализаторов на основе рекурсивного спуска состоит в использовании нисходящего алгоритма, не способного справиться с левой рекурсией, что в результате делает его более запутанным при работе сложенными операторными выражениями (*Nested Operator Expression* (333)). Реальная производительность также обычно уступает генератору синтаксических анализаторов. Но на практике для DSL эти недостатки не являются такими уж важными.

Синтаксический анализатор на основе рекурсивного спуска прост в реализации до тех пор, пока относительно проста грамматика. Одним из факторов, которые могут облегчить его написание, является ограничение предпросмотра, т.е. количества токенов, которые должны быть считаны после текущего для принятия однозначного решения о том, что следует делать дальше. В общем случае я бы не использовала синтаксические анализаторы рекурсивного спуска для грамматик, требующих более одного символа предпросмотра; для таких грамматик лучше подходят генераторы синтаксических анализаторов.

21.3. Дополнительная информация

Для получения дополнительной информации в контексте менее традиционных языков программирования можно воспользоваться [23]. И, конечно же, книгой всех времен и народов по данной тематике остается “Книга дракона” [1].

21.4. Рекурсивный спуск и контроллер мисс Грант (Java)

Начнем с класса синтаксического анализатора, который включает переменные экземпляра для входного буфера, выхода конечного автомата и различных данных синтаксического анализа. Эта реализация использует шаблон (*Regex Table Lexer* (247)) для заполнения входного буфера токенов из исходной строки.

```
class StateMachineParser...
    private TokenBuffer tokenBuffer;
    private StateMachine machineResult;
    private ArrayList<Event> machineEvents;
    private ArrayList<Command> machineCommands;
    private ArrayList<Event> resetEvents;
    private Map<String, State> machineStates;
    private State partialState;
```

Конструктор класса конечного автомата принимает входной буфер токенов и настраивает структуры данных. Существует очень простой метод запуска синтаксического анализатора, который вызывает функцию, представляющую весь наш конечный автомат в целом.

```
class StateMachineParser...
    public StateMachine startParser() {
        if (stateMachine()) /* Продукция основного уровня */
            loadResetEvents();
        }
        return machineResult;
    }
```

Метод `startParser` отвечает также за построение конечного автомата в случае успеха. Единственное оставшееся действие, не охваченное остальными методами, — заполнение сбрасывающего события конечного автомата.

Правило грамматики для нашего конечного автомата представляет собой простую последовательность различных блоков.

Файл грамматики...

```
stateMachine: eventBlock optionalResetBlock
               optionalCommandBlock stateList
```

Функция верхнего уровня представляет собой просто последовательность различных компонентов конечного автомата.

```
class StateMachineParser...
    private boolean stateMachine() {
        boolean parseSuccess = false;
        if (eventBlock())
            if (optionalResetBlock())
                if (optionalCommandBlock())
                    if (stateList())
                        parseSuccess = true;
        }
    }
    return parseSuccess;
}
```

Воспользуемся объявлением события, чтобы показать, как работает вместе большинство функций. Первая продукция, которая строит блок события, представляет собой последовательность.

Файл грамматики...

```
eventBlock: eventKeyword eventDecList endKeyword
```

Код для этой последовательности соответствует описанному выше шаблону. Обратите внимание на сохранение начальной позиции в буфере и восстановление в случае, когда полная последовательность не распознана.

```

class StateMachineParser...
private boolean eventBlock() {
    Token t;
    boolean parseSuccess = false;
    int save = tokenBuffer.getCurrentPosition();
    t = tokenBuffer.nextToken();
    if (t.isTokenType(ScannerPatterns.TokenTypes.TT_EVENT)) {
        tokenBuffer.popToken();
        parseSuccess = eventDecList();
    }
    if (parseSuccess) {
        t = tokenBuffer.nextToken();
        if (t.isTokenType(ScannerPatterns.TokenTypes.TT_END)) {
            tokenBuffer.popToken();
        }
        else {
            parseSuccess=false;
        }
    }
    if (!parseSuccess) {
        tokenBuffer.resetCurrentPosition(save);
    }
    return parseSuccess;
}

```

Правило грамматики для списка событий очень простое.

Файл грамматики...

```
eventDecList: eventDec+
```

Функция eventDecList точно следует шаблону. Все действия выполняются в функции eventDec.

```

class StateMachineParser...
private boolean eventDecList() {
    int save = tokenBuffer.getCurrentPosition();
    boolean parseSuccess = false;
    if (eventDec()) {
        parseSuccess = true;
        while (parseSuccess) {
            parseSuccess = eventDec();
        }
        parseSuccess = true;
    }
    else {
        tokenBuffer.resetCurrentPosition(save);
    }
    return parseSuccess;
}

```

Реальная работа начинается, когда выявляется соответствие самому событию. Грамматика очень проста.

Файл грамматики...

```
eventDec: identifier identifier
```

Код для этой последовательности также сохраняет позицию в буфере токенов. Код также выполняет наполнение модели конечного автомата при успешном распознавании.

```

class StateMachineParser...
private boolean eventDec() {
    Token t;
```

```

boolean parseSuccess = false;
int save = tokenBuffer.getCurrentPosition();
t = tokenBuffer.nextToken();
String elementLeft = "";
String elementRight = "";

if (t.isTokenType(
    ScannerPatterns.TokenTypes.TT_IDENTIFIER)) {
    elementLeft = consumeIdentifier(t);
    t = tokenBuffer.nextToken();
    if (t.isTokenType(
        ScannerPatterns.TokenTypes.TT_IDENTIFIER)) {
        elementRight = consumeIdentifier(t);
        parseSuccess = true;
    }
}

if (parseSuccess) {
    makeEventDec(elementLeft, elementRight);
} else {
    tokenBuffer.resetCurrentPosition(save);
}
return parseSuccess;
}

```

Две вызываемые здесь вспомогательные функции требуют дальнейшего пояснения. Первая, `consumeIdentifier`, выполняет перемещение в буфере токенов и возвращает значение токена для идентификатора, которое затем может использоваться для наполнения объявления события.

```

class StateMachineParser...
private String consumeIdentifier(Token t) {
    String identName = t.tokenValue;
    tokenBuffer.popToken();
    return identName;
}

```

Вспомогательная функция `makeEventDec` использует имя и код события для фактического наполнения объявления события.

```

class StateMachineParser...
private void makeEventDec(String left, String right) {
    machineEvents.add(new Event(left, right));
}

```

С точки зрения действий реальные трудности имеются только в обрабатываемых состояниях. Поскольку переходы могут ссылаться на еще несуществующие состояния, наши вспомогательные функции должны допускать ссылки на состояния, которые еще не определены. Это свойство выполняется для всех реализаций, которые не используют построение дерева (Tree Construction (289)).

Последним методом, достойным описания, является `optionalResetBlock`, реализующий следующее правило грамматики.

```

Файл грамматики...
optionalResetBlock: (resetBlock)?
resetBlock: resetKeyword (resetEvent)* endKeyword
resetEvent: identifier

```

Приведенная далее реализация включает различные шаблоны операторов грамматики, встречающихся в правиле.

```
class StateMachineParser...
private boolean optionalResetBlock() {
    int save = tokenBuffer.getCurrentPosition();
    boolean parseSuccess = true;
    Token t = tokenBuffer.nextToken();
    if (t.isTokenType(ScannerPatterns.TokenTypes.TT_RESET)) {
        tokenBuffer.popToken();
        t = tokenBuffer.nextToken();
        parseSuccess = true;
        while ((! (t.isTokenType(
            ScannerPatterns.TokenTypes.TT_END))) &
            (parseSuccess)) {
            parseSuccess = resetEvent();
            t = tokenBuffer.nextToken();
        }
        if (parseSuccess) {
            tokenBuffer.popToken();
        } else {
            tokenBuffer.resetCurrentPosition(save);
        }
    }
    return parseSuccess;
}

private boolean resetEvent() {
    Token t;
    boolean parseSuccess = false;

    t = tokenBuffer.nextToken();
    if (t.isTokenType(
        ScannerPatterns.TokenTypes.TT_IDENTIFIER)) {
        resetEvents.add(findEventFromName(t.tokenValue));
        parseSuccess = true;
        tokenBuffer.popToken();
    }
    return parseSuccess;
}
```

В этом методе при отсутствии ключевого слова `reset` возвращается значение `true`, так как блок в целом не является обязательным. Если же указанное ключевое слово существует, то мы должны иметь нуль или более объявлений сбрасывающего события, за которыми следует ключевое слово `end`. Если это не так, то проверка соответствия данного блока правилу терпит неудачу, и возвращается значение `false`.

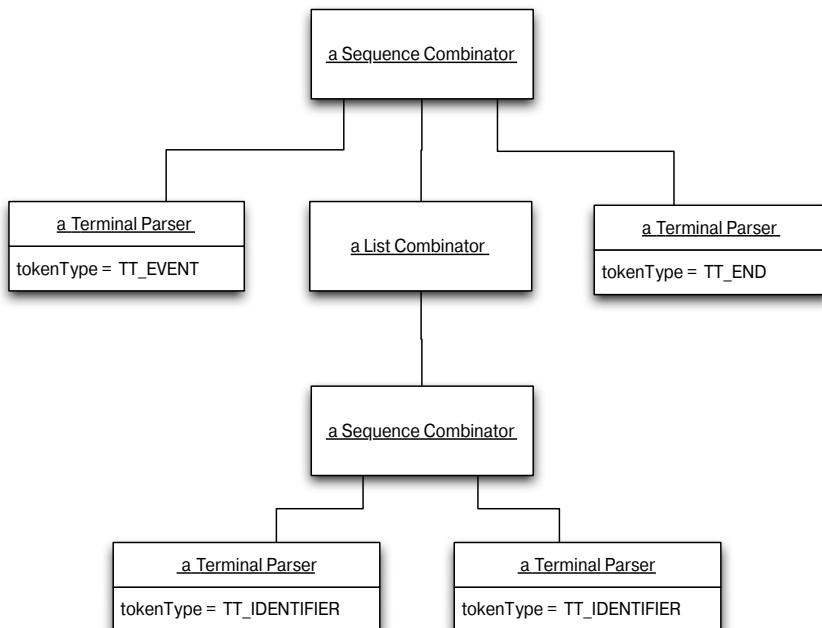
Глава 22

Комбинатор синтаксических анализаторов

Автор — Ребекка Парсонс

Parser Combinator

Создание исходящего синтаксического анализатора путем композиции объектов синтаксических анализаторов



Несмотря на наши уверения в том, что генераторы синтаксических анализаторов (Parser Generator (277)) далеко не так страшны, как их малютят, все же имеются веские причины по возможности их избегать. Наиболее очевидная проблема заключается в дополнительных шагах в процессе сборки, требующихся сначала для генерации синтаксического анализатора, а затем и для его построения. Хотя генераторы синтаксических анализаторов все еще остаются правильным выбором для более сложных контекстно-свободных грамматик (особенно если грамматика неоднозначна или решающее значение имеет производительность), непосредственная реализация синтаксического анализатора на языке программирования общего назначения представляет собой вполне жизнеспособный вариант.

Комбинатор синтаксических анализаторов реализует грамматику с использованием структуры объектов синтаксических анализаторов. Распознаватели символов в правилах вывода объединяются с использованием шаблона *Composites* [15]. Фактически комбинаторы синтаксического анализа представляют семантическую модель (*Semantic Model* (171)) грамматики.

22.1. Как это работает

Как и в случае синтаксического анализатора на основе рекурсивного спуска (Recursive Descent Parser (253)), для лексического анализа входной строки мы используем лексический анализатор, например, на базе таблицы регулярных выражений (Regex Table Lexer (247)). Комбинатор синтаксических анализаторов работает с полученной в результате строкой токенов.

Основная идея комбинаторов синтаксических анализаторов проста. Термин “комбинатор” берет свое начало из функциональных языков. Комбинаторы создаются для получения путем объединения более сложных операций, выход которых имеет тот же тип, что и их вход. Так, комбинаторы синтаксических анализаторов объединяются для получения более сложных комбинаторов. В функциональных языках эти комбинаторы представляет собой функции, но в объектно-ориентированной среде мы можем получить то же самое с объектами. Мы начнем с базовых случаев, которыми являются распознаватели терминалных символов нашей грамматики. Затем мы используем комбинаторы, которые реализуют различные операторы грамматики (такие, как операторы последовательности, списка и т.д.), для реализации правил продукции грамматики. По сути, для каждого нетерминала грамматики имеется свой комбинатор, так же как в синтаксическом анализаторе рекурсивного спуска для каждого нетерминала имеется рекурсивная функция.

Каждый комбинатор отвечает за распознавание какой-то части языка, определение соответствия, отбор токенов из входного буфера и выполнение прочих необходимых действий. Это те же операции, что и выполняемые функциями в синтаксическом анализаторе на основе рекурсивного спуска. Для реализации распознавания различных операторов грамматики применима та же логика, что и использовавшаяся в реализации синтаксических анализаторов на основе рекурсивного спуска. Чем же в действительности отличается данный шаблон? Главным в нем является то, что мы абстрагируемся от фрагментов логики, связанной с обработкой операторов грамматики для нисходящего синтаксического анализа, и создаем для ее хранения комбинаторы. В то время как синтаксический анализатор на основе рекурсивного спуска объединяет эти фрагменты с помощью вызовов функций в своем коде, комбинатор синтаксических анализаторов выполняет объединение, связывая объекты в адаптивную модель (*Adaptive Model* (478)).

Индивидуальные комбинаторы синтаксических анализаторов принимают в качестве входных данных текущее состояние сопоставления, текущий буфер токенов и, возможно,

множество накопленных результатов действий. Комбинаторы синтаксических анализаторов возвращают состояние сопоставления, буфер токенов (возможно, измененный) и множество результатов действий. Для упрощения описания будем временно считать, что буфер токенов и множество результатов сопоставлений хранятся где-то в виде состояния. Это предположение будет изменено позже, а пока что такое допущение упрощает следование логике комбинатора. Кроме того, сначала мы сосредоточимся на описании логики распознавания, а уже затем перейдем к действиям.

Рассмотрим базовый случай — распознаватель для терминального символа. Фактическое распознавание терминального символа выполняется легко: мы просто сравниваем токен в текущей позиции во входном буфере с токеном, для которого создан наш распознаватель. В случае соответствия мы перемещаем текущую позицию в буфере токенов.

Теперь рассмотрим комбинаторы для различных операторов грамматики. Начнем с оператора альтернативы.

Файл грамматики...
С : А | В

Комбинатор альтернативы для С сначала испытывает один комбинатор, скажем, В. Если состояние сопоставления — true, то возвращаемое С значение то же, что и возвращаемое В. Мы поочередно испытываем все альтернативы. Если все испытания терпят неудачу, возвращаемое значение представляет собой неудачное состояние сопоставления, и буфер токенов остается неизменным. В псевдокоде этот комбинатор выглядит следующим образом.

```
CombinatorResult C ()  
  
if (A())  
    then return true  
else  
    if (B())  
        then true  
    else return false
```

Как видите, логика здесь абсолютно та же, что и в алгоритме рекурсивного спуска.
Оператор последовательности несколько сложнее.

Файл грамматики...
С : А В

Для его реализации необходимо пройти по всем компонентам последовательности. В случае неудачи любого из сопоставлений необходимо вернуть буфер токенов в состояние, в котором он находился до выполнения. Для приведенного выше правила грамматики комбинатор выглядит следующим образом.

```
CombinatorResult C ()  
  
saveTokenBuffer()  
if (A())  
    then  
    if (B())  
        then  
            return true  
        else  
            restoreTokenBuffer  
            return false  
    else return false
```

Эта реализация, как и другие, приведенные ниже, основана на характеристическом поведении комбинаторов. В случае успешного сопоставления относящиеся к нему токены потребляются из входного буфера (выполняется перемещение указателя). В случае неудачного сопоставления буфер остается в неизменном состоянии.

Оператор необязательности достаточно прост.

Файл грамматики...

```
C : A?
```

Этот оператор либо возвращает исходные токены, либо модифицирует их, основываясь на сопоставлении с A. Для приведенного выше правила комбинатор имеет следующий вид.

```
CombinatorResult C ()
```

```
A()
return true
```

Далее рассмотрим оператор повторения с одним или более экземплярами.

Файл грамматики...

```
C : A+
```

Этот комбинатор сначала проверяет наличие одного A. Если A есть в наличии, выполняется цикл сопоставлений A до тех пор, пока очередное сопоставление не будет неудачным, и возвращается обновленный буфер токенов. Если же первоначальное сопоставление неудачно, возвращаются значение `false` и неизмененный буфер токенов.

```
CombinatorResult C ()
```

```
if (A())
then
  while (A())
    return true
else
  return false
```

Понятно, что для оператора повторения с нулем или несколькими экземплярами следует убрать первоначальную проверку наличия A и всегда возвращать значение `true`. Буфер обрабатывается в соответствии с результатами сопоставления. Код в этом случае принимает такой простой вид.

```
CombinatorResult C ()
```

```
while (A())
return true
```

Показанные здесь реализации комбинаторов являются прямыми реализациями определенных правил. Мощь комбинаторов синтаксических анализаторов проистекает из того факта, что мы можем создавать составные комбинаторы из компонентов. Таким образом, код для оператора последовательности

Файл грамматики...

```
C : A B
```

на самом деле будет больше походить на объявление вида

```
C = new SequenceCombinator(A, B)
```

Здесь логика, реализующая последовательности, общая для всех правил грамматики такого типа.

22.1.1. Выполнение действий

Теперь, когда мы знаем, как работает распознавание, пора перейти к действиям. Сейчас мы вновь предполагаем, что у нас имеется некоторое состояние (значение сопоставления), с которым мы работаем в действиях. Действия могут принимать различные формы. При построении дерева (*Tree Construction* (289)) в процессе синтаксического анализа действия будут строить абстрактное синтаксическое дерево. При применении встроенной трансляции (*Embedded Translation* (305)) действия будут наполнять нашу семантическую модель.

Давайте вновь начнем с базового случая — комбинатора терминального символа. При успешном сопоставлении мы соответствующим образом заполняем его результирующее значение и выполняем над ним определенные действия. Например, в случае распознавания идентификатора он может быть записан в таблицу символов (*Symbol Table* (177)). Для терминальных символов, таких как идентификаторы и числа, действия зачастую просто записывают значения токена для дальнейшего использования.

Операция последовательности становится более интересной, когда дело доходит до действий. Концептуально, когда мы распознали все компоненты последовательности, мы должны вызвать действия на основе списка значений сопоставления отдельных компонентов. Давайте модифицируем распознаватель для вызова действий указанным способом.

```
CombinatorResult C ()  
  
    saveTokenBuffer()  
    saveActions()  
    if (A())  
        then  
            if (B())  
                then  
                    executeActions (aResult, bResult)  
                    return true  
                else  
                    restoreTokenBuffer()  
                    restoreActions()  
                    return false  
            else return false
```

Основная работа скрыта в методе `executeActions`. Значения сопоставлений для A и B должны быть сохранены, чтобы позже их можно было использовать в действиях.

Подход для других операторов грамматики аналогичен. Оператор альтернативы выполняет действия только для выбранного варианта. Оператор списка, как и оператор последовательности, должен работать со всеми значениями сопоставления. Очевидно, что оператор необязательности выполняет действие только при успешном сопоставлении.

Вызовы действий относительно просты. Основная проблема — в получении надлежащих методов действий, связанных с комбинатором. В языках с замыканиями или другими способами передачи функций в качестве параметров можно просто передавать детальную информацию о методе действия в конструктор в виде функции. В языках без замыканий, таких как Java, приходится поступать немного более хитро. Один из подходов состоит в расширении классов операторов классами, специфичными для того или иного конкретного правила, и перекрывать метод действия для получения требуемого поведения.

Как упоминалось выше, действия могут быть использованы для построения абстрактного синтаксического дерева. В этом случае значения сопоставлений, передаваемые функции действия, будут представлять собой деревья, построенные для различных компонентов, а действие будет комбинировать эти синтаксические деревья так, как следует

из рассматриваемого правила грамматики. Например, оператору списка обычно соответствует некоторый представляющий список тип узла в синтаксическом дереве. Действие оператора списка будет в таком случае создавать новое поддерево с этим узлом списка в качестве корня. Для компонентов, соответствующих дочерним узлам этого корня, будут строиться свои поддеревья.

22.1.2. Функциональный стиль комбинаторов

Пришло время ослабить предположение о работе с результатами действий и буфером токенов с помощью состояния. Мысля в функциональном стиле, комбинатор представляет собой функцию, которая отображает входное результирующее значение комбинатора на выходное результирующее значение комбинатора. Компонентами этого значения являются текущее состояние буфера токенов, текущее состояние сопоставления и текущее состояние выполненных к этому моменту кумулятивных действий. В этом стиле реализация оператора последовательности с действиями будет выглядеть следующим образом.

```
CombinatorResult C (in)
    aResult = A(in)
    if (aResult.matchSuccess)
        then
            bResult = B(aResult)
            if (bResult.matchSuccess)
                then
                    cResult.value =
                        executeActions (aResult.value, bResult.value)
                    return (true, bResult.tokens, cResult.value)
                else
                    return (false, in.tokens, in.value)
            else
                return (false, in.tokens, in.value)
```

В этой версии выполнять сохранения необязательно, поскольку значение входного параметра остается корректным. Эта версия также делает более явными обработку буфера токенов и происхождение значений действия.

22.2. Когда это использовать

Этот подход занимает нишу между синтаксическими анализаторами на основе рекурсивного спуска (Recursive Descent Parser (253)) и генераторами синтаксических анализаторов (Parser Generator (277)). Важным преимуществом использования генератора синтаксических анализаторов является явная спецификация грамматики языка. Грамматика в синтаксическом анализаторе рекурсивного спуска неявно выражена в функциях, но ее трудно воспринимать как грамматику. При использовании подхода комбинаторов синтаксических анализаторов эти комбинаторы могут быть определены декларативно, как было показано в примере выше. Хотя здесь и не используется синтаксис БНФ (BNF (237)), грамматика достаточно четко определена в терминах комбинаторов компонентов и операторов. Так что, применяя комбинаторы синтаксических анализаторов, вы получаете достаточно явные грамматики без сложностей построения, характерных для генераторов синтаксических анализаторов.

Существуют библиотеки, которые реализуют различные операторы грамматики на разных языках программирования. Очевидным выбором для реализации комбинаторов синтаксических анализаторов являются функциональные языки программирования, учитывая их первоклассную поддержку функций в качестве объектов, позволяющую пе-

редавать функцию действия конструктору комбинатора в качестве параметра. Однако вполне возможна реализация этого шаблона и на других языках программирования.

Как и синтаксический анализатор на основе рекурсивного спуска, комбинатор синтаксических анализаторов дает нисходящий синтаксический анализатор, так что к этому шаблону применимы те же ограничения, что и к синтаксическому анализатору на основе рекурсивного спуска. Точно так же к комбинатору синтаксических анализаторов приложимы и многие из преимуществ синтаксического анализатора на основе рекурсивного спуска, в частности простота понимания времени выполнения действий. Хотя комбинатор синтаксических анализаторов — очень своеобразная реализация анализатора, управляющий алгоритм синтаксического анализа можно отслеживать с помощью тех же инструментов, которые применяются для отладки других программ. По сути, подход комбинатора синтаксических анализаторов в сочетании с операторной библиотекой или протестированными реализациями операторов позволяет создателю языка сосредоточиться на действиях, а не на синтаксическом анализе.

Самым большим недостатком комбинатора синтаксических анализаторов является то, что строить его приходится самостоятельно. Кроме того, вы не получите более интеллектуальный анализ и возможности обработки ошибок, которые при использовании мощного генератора синтаксических анализаторов вы получаете “из коробки”.

22.3. Комбинаторы синтаксических анализаторов и контроллер мисс Грант (Java)

Для реализации синтаксического анализатора конечного автомата на Java с помощью комбинаторов синтаксических анализаторов нужно осуществить пару проектных решений. В данном примере мы будем применять более функциональный подход с использованием объекта результата комбинатора и встраиваемой трансляции (Embedded Translation (305)) для наполнения объекта конечного автомата по мере выполнения синтаксического анализа.

Начнем с повторения полной грамматики конечного автомата. Здесь для соответствия используемой стратегии реализации продукции перечислены в обратном порядке.

Файл грамматики...

```
eventDec      : IDENTIFIER IDENTIFIER
eventDecList  : (eventDec) *
eventBlock    : EVENTS eventDecList END
eventList     : (IDENTIFIER) *
resetBlock   : (RESET eventList END)?
commandDec   : IDENTIFIER IDENTIFIER
commandDecList: (commandDec) *
commandBlock : (COMMAND commandDecList END)?
transition   : IDENTIFIER TRANSITION IDENTIFIER
transitionList: (transition) *
actionDec    : IDENTIFIER
actionList   : (actionDec) *
actionBlock  : (ACTIONS LEFT actionList RIGHT)?
stateDec     : STATE IDENTIFIER actionBlock transitionList END
stateList    : (stateDec)*
stateMachine : eventBlock resetBlock commandBlock stateList
```

Для построения комбинатора синтаксических анализаторов полного конечного автомата следует построить комбинаторы для каждого из компонентов терминалов и нетерминалов в грамматике. Вот полный набор комбинаторов для данной грамматики на Java.

```

Файл грамматики...
// Терминальные символы
private Combinator matchEndKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_END);
private Combinator matchCommandKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_COMMANDS);
private Combinator matchEventsKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_EVENT);
private Combinator matchResetKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_RESET);
private Combinator matchStateKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_STATE);
private Combinator matchActionsKeyword =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_ACTIONS);
private Combinator matchTransitionOperator =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_TRANSITION);
private Combinator matchLeftOperator =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_LEFT);
private Combinator matchRightOperator =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_RIGHT);
private Combinator matchIdentifier =
    new TerminalParser(
        ScannerPatterns.TokenTypes.TT_IDENTIFIER);
// Правила нетерминальных продукции
private Combinator matchEventDec =
    new EventDec(matchIdentifier, matchIdentifier);
private Combinator matchEventDecList =
    new ListCombinator(matchEventDec);
private Combinator matchEventBlock =
    new SequenceCombinator(
        matchEventsKeyword, matchEventDecList,
        matchEndKeyword
    );
private Combinator matchEventList =
    new ResetEventsList(matchIdentifier);
private Combinator matchResetBlock =
    new OptionalSequenceCombinator (
        matchResetKeyword, matchEventList, matchEndKeyword
    );
private Combinator matchCommandDec =
    new CommandDec(matchIdentifier, matchIdentifier);
private Combinator matchCommandList =
    new ListCombinator(matchCommandDec);
private Combinator matchCommandBlock =
    new OptionalSequenceCombinator(
        matchCommandKeyword, matchCommandList,
        matchEndKeyword
    );
private Combinator matchTransition =
    new TransitionDec(
        matchIdentifier, matchTransitionOperator,
        matchIdentifier);

```

```

private Combinator matchTransitionList =
    new ListCombinator(matchTransition) ;
private Combinator matchActionDec =
    new ActionDec(
        ScannerPatterns.TokenTypes.TT_IDENTIFIER
    ) ;
private Combinator matchActionList =
    new ListCombinator(matchActionDec) ;
private Combinator matchActionBlock =
    new OptionalSequenceCombinator(
        matchActionsKeyword, matchLeftOperator,
        matchActionList, matchRightOperator) ;
private Combinator matchStateName =
    new StateName(
        ScannerPatterns.TokenTypes.TT_IDENTIFIER
    ) ;
private Combinator matchStateDec =
    new StateDec(
        matchStateKeyword, matchStateName, matchActionBlock,
        matchTransitionList, matchEndKeyword
    ) ;
private Combinator matchStateList =
    new ListCombinator(matchStateDec) ;
private Combinator matchStateMachine =
    new StateMachineDec(
        matchEventBlock, matchResetBlock,
        matchCommandBlock, matchStateList
    ) ;

```

Комбинаторы терминальных символов не имеют прямых аналогов в файле грамматики, так как для поиска этих символов мы использовали лексический анализатор. Однако далее объявления комбинаторов используют предварительно определенные комбинаторы, а те реализуют различные операторы грамматики для создания составных комбинаторов. Мы рассмотрим каждый из них отдельно.

Описание реализации комбинатора начнем с простых случаев и постепенно построим распознаватель для конечного автомата. Начнем с базового класса `Combinator`, от которого будут наследованы все прочие комбинаторы.

```

class Combinator...
public Combinator() {}
public abstract CombinatorResult
    recognizer(CombinatorResult inbound);
public void action(StateMachineMatchValue... results) {}

```

Все комбинаторы имеют две функции. Распознаватель `recognizer` отображает входной `CombinatorResult` на результирующее значение того же типа.

```

class CombinatorResult...
private TokenBuffer tokens;
private Boolean matchStatus;
private StateMachineMatchValue matchValue;

```

Три компонента результата представляют собой состояние буфера токенов, успех или неудачу распознавания, и объект, который представляет результирующее значение действия в буфере токенов. В нашем случае мы используем его просто для хранения строки значения токена.

```

class StateMachineMatchValue...
private String matchString;
public StateMachineMatchValue (String value) {

```

```

        matchString = value;
    }
    public String getMatchString () {
        return matchString;
    }
}

```

Вторым методом комбинатора является метод, который выполняет некоторые действия, относящиеся к сопоставлению. Входом для функции действия являются объекты значений сопоставления, по одному объекту для каждого комбинатора компонента. Например, у комбинатора последовательности, представляющего правило

Файл грамматики...
С : А В

метод действия будет получать два значения сопоставления.

Начнем с распознавателя для терминальных символов. В качестве примера воспользуемся распознавателем идентификатора. Вот его объявление.

```

class StateMachineCombinatorParser...
private Combinator matchIdentifier = new TerminalParser(
    ScannerPatterns.TokenTypes.TT_IDENTIFIER
);

```

Класс терминального комбинатора имеет одну переменную экземпляра, идентифицирующую найденный токен.

```

class TerminalParser...
public class TerminalParser extends Combinator {
    private ScannerPatterns.TokenTypes tokenMatch;
    public TerminalParser(ScannerPatterns.TokenTypes match) {
        this.tokenMatch = match;
    }
}

```

Стандартная функция распознавания терминала достаточно проста.

```

class TerminalParser...
public CombinatorResult
recognizer(CombinatorResult inbound) {
    if (!inbound.matchSuccess()) return inbound;
    CombinatorResult result;
    TokenBuffer tokens = inbound.getTokenBuffer();
    Token t = tokens.nextToken();
    if (t.isTokenType(tokenMatch)) {
        TokenBuffer outTokens =
            new TokenBuffer(tokens.makePoppedTokenList());
        result = new CombinatorResult(outTokens, true,
            new StateMachineMatchValue(t.tokenValue()));
        action(result.getMatchValue());
    } else {
        result = new CombinatorResult(tokens, false,
            new StateMachineMatchValue(""));
    }
    return result;
}

```

Убедившись, что входное состояние сопоставления — `true`, мы проверяем текущую позицию в буфере токенов, чтобы увидеть, соответствует ли это значение значению нашей переменной экземпляра. Если соответствует, мы строим объект `CombinatorResult` для успешного сопоставления, с измененным буфером токенов и значением найденного токена,

записанными в результирующее значение. Для этого значения сопоставления вызывает-ся метод действия, который в данном случае просто ничего с ним не делает.

Теперь перейдем к чему-нибудь более интересному. Грамматическое правило блока объявления события выглядит следующим образом.

Файл грамматики...

```
eventBlock: eventKeyword eventDecList endKeyword
```

Объявление комбинатора для этого правила использует `SequenceCombinator`.

```
class StateMachineCombinatorParser...
private Combinator matchEventBlock = new SequenceCombinator(
    matchEventsKeyword, matchEventDecList, matchEndKeyword
);
```

В этом случае мы вновь используем пустой метод действия, так как реальная работа на самом деле выполняется в другом месте. Конструктор экземпляра `SequenceCombinator` принимает список комбинаторов, по одному для каждого символа в правиле продукции.

```
class SequenceCombinator...
public class SequenceCombinator
    extends AbstractSequenceCombinator {
    public SequenceCombinator (Combinator ... productions) {
        super(false, productions);
    }
}
```

В этой реализации мы решили сделать отдельные классы для обязательной и необязательной последовательностей с совместно используемой реализацией, а не вводить оператор необязательности и добавлять еще один уровень правил в грамматику. Мы расширяем базовый класс `AbstractSequenceCombinator`, создавая классы `SequenceCombinator` и `OptionalSequenceCombinator`. Общий класс имеет переменную экземпляра, представляющую список комбинаторов последовательности и логическое значение, указывающее, является ли это составное правило обязательным.

```
class AbstractSequenceCombinator...
public abstract class AbstractSequenceCombinator
    extends Combinator {
    private Combinator[] productions;
    private Boolean isOptional;

    public AbstractSequenceCombinator(Boolean optional,
                                       Combinator... productions) {
        this.productions = productions;
        this.isOptional = optional;
    }
}
```

Функция сопоставления для совместно используемой последовательности комбинаторов применяет логическое значение для определения поведения при неудачном сопоставлении.

```
class AbstractSequenceCombinator...
public CombinatorResult
    recognizer(CombinatorResult inbound) {
    if (!inbound.matchSuccess()) return inbound;
    StateMachineMatchValue[] componentResults =
        new StateMachineMatchValue[productions.length];
    CombinatorResult latestResult = inbound;
    int productionIndex = 0;
```

```

        while (latestResult.matchSuccess() &&
               productionIndex < productions.length) {
            Combinator p = productions[productionIndex];
            latestResult = p.recognizer(latestResult);
            componentResults[productionIndex] =
                latestResult.getMatchValue();
            productionIndex++;
        }
        if (latestResult.matchSuccess()) {
            action(componentResults);
        } else if (isOptional) {
            latestResult =
                new CombinatorResult(inbound.getTokenBuffer(),
                                     true, new StateMachineMatchValue(""));
        } else {
            latestResult =
                new CombinatorResult(inbound.getTokenBuffer(),
                                     false, new StateMachineMatchValue(""));
        }
        return (latestResult);
    }
}

```

Мы вновь используем конструкцию, которая обеспечивает немедленный выход из метода, если входное состояние сопоставления равно `false`. Функция сопоставления использует цикл `while` для прохода по различным комбинаторам, определяющим последовательность. Цикл завершается, когда состояние сопоставления равно `false` или когда проверены все комбинаторы. Если сопоставление прошло успешно, значит, все комбинаторы последовательности успешно сопоставлены. В этом случае мы вызываем метод действия для массива значений сопоставления компонентов. Если входное значение было успешным, но в цикле некоторое сопоставление оказалось неудачным, нужно проконсультироваться с флагом необязательности. В случае необязательной последовательности данное сопоставление рассматривается как успешное, со значениями и входными токенами, возвращенными в их состояния в начале сопоставления. Естественно, метод действия при этом не вызывается, поскольку успешное сопоставление не выполнено. В противном случае (при обязательной последовательности) комбинатор сообщает о неудаче сопоставления и восстанавливает входные значения токенов и сопоставлений.

Примером необязательной последовательности может служить блок сброса

```

class StateMachineCombinatorParser...
private Combinator matchResetBlock =
    new OptionalSequenceCombinator(matchResetKeyword,
                                    matchEventList,
                                    matchEndKeyword);

```

для правила грамматики

```

Файл грамматики...
optionalResetBlock: (resetBlock)?
ResetBlock      : resetKeyword (resetEvent)* endKeyword
resetEvent      : identifier

```

Перед тем как перейти к настройке действий, давайте закончим с операторами грамматики, разобравшись с оператором списка. Правило нашей грамматики с использованием оператора списка — это список объявлений событий.

```

Файл грамматики...
eventDecList: eventDec+

```

Соответствующее объявление в синтаксическом анализаторе имеет следующий вид.

```
class StateMachineCombinatorParser...
private Combinator matchEventDecList =
    new ListCombinator(matchEventDec);
```

Показанная далее реализация предназначена для необязательных списков. В этой реализации мы предполагаем, что оператор списка применен к одному нетерминалу. Понятно, что это ограничение можно легко ослабить. Однако в данной реализации конструктор принимает единственный комбинатор, который затем оказывается представленным единственной переменной экземпляра в классе.

```
class ListCombinator...
public class ListCombinator extends Combinator {
    private Combinator production;
    public ListCombinator(Combinator production) {
        this.production = production;
    }
}
```

Функция сопоставления достаточно проста.

```
class ListCombinator...
public CombinatorResult
recognizer(CombinatorResult inbound) {
    if (!inbound.matchSuccess()) return inbound;
    CombinatorResult latestResult = inbound;
    StateMachineMatchValue returnValues[];
    ArrayList<StateMachineMatchValue> results =
        new ArrayList<StateMachineMatchValue>();

    while (latestResult.matchSuccess()) {
        latestResult = production.recognizer(latestResult);
        if (latestResult.matchSuccess()) {
            results.add(latestResult.getMatchValue());
        }
    }

    if (results.size() > 0) { //Имеется соответствие
        returnValues = results.toArray(
            new StateMachineMatchValue[results.size()]);
        action(returnValues);
        latestResult =
            new CombinatorResult(latestResult.getTokenBuffer(),
                true,
                new StateMachineMatchValue(""));
    }
    return (latestResult);
}
```

Поскольку заранее количество успешных сопоставлений для списка неизвестно, мы используем коллекцию `ArrayList` для хранения значений сопоставлений. Чтобы “осчастливить” систему типов Java, коллекцию нужно преобразовать в массив элементов такого же типа для передачи методу действия в качестве аргумента.

Как мы уже упоминали, действия для всех рассмотренных выше комбинаторов, за исключением метода идентификатора, ничего не делают. Действия, наполняющие различные компоненты конечного автомата, на самом деле связаны с другими нетерминалами. В данной реализации мы использовали внутренние классы Java, расширение базового класса оператора грамматики и перекрытие в нем метода действия. Рассмотрим правило продукции объявления события

Файл грамматики...

```
eventDec: identifier identifier
```

с соответствующим объявлением в синтаксическом анализаторе.

```
class StateMachineCombinatorParser...
private Combinator matchEventDec =
    new EventDec(matchIdentifier, matchIdentifier);
```

Определение класса имеет следующий вид.

```
class StateMachineCombinatorParser...
private class EventDec extends SequenceCombinator {
    public EventDec(Combinator... productions) {
        super(productions);
    }
    public void action(StateMachineMatchValue... results) {
        assert results.length == 2;
        addMachineEvent(new Event(results[0].getMatchString(),
            results[1].getMatchString()));
    }
}
```

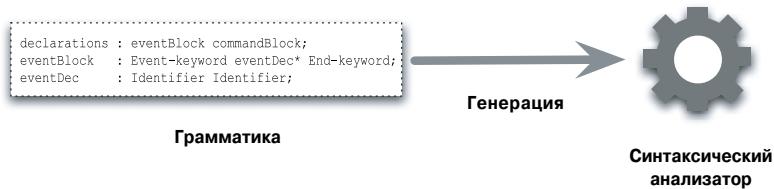
Этот класс расширяет класс `SequenceCombinator` и перекрывает метод действия. Как и для всех методов действий, входные данные являются просто списком результатов сопоставлений, представляющих имена идентификаторов для объявления события. Для загрузки события в конечный автомат мы используем те же вспомогательные функции, что и ранее, извлекая строки имен из соответствующих значений сопоставлений. Другая продукция следует той же схеме реализации.

Глава 23

Генератор синтаксических анализаторов

Parser Generator

Построение синтаксического анализатора на основе файла грамматики в качестве DSL



Файл грамматики представляет собой естественное средство описания синтаксической структуры DSL. Если у вас есть грамматика, превратить ее в написанный вручную синтаксический анализатор — громоздкая и утомительная работа, которая, естественно, должна быть поручена компьютеру.

Генератор синтаксических анализаторов использует упомянутый файл грамматики для генерации синтаксического анализатора. Синтаксический анализатор может быть обновлен путем простого обновления файла грамматики с последующей регенерацией. Сгенерированный синтаксический анализатор может использовать эффективные алгоритмы, которые сложно реализовывать и поддерживать вручную.

23.1. Как это работает

Создание собственного генератора синтаксических анализаторов — задача непростая, и любой, кто способен на это, вряд ли узнает что-то новое из этой книги. Так что здесь я буду говорить только об использовании генераторов синтаксических анализаторов. К счастью, такие генераторы являются инструментами в достаточной степени рас-

пространенными, при этом некоторые полезные их формы доступны на большинстве платформ программирования, часто с открытым исходным кодом.

Обычный способ работы с генератором синтаксических анализаторов — написать файл грамматики. Этот файл представляет собой особую форму БНФ (BNF (237)), которая используется данным генератором синтаксических анализаторов. Не ждите никакой стандартизации в этой области: если вы измените свой генератор синтаксических анализаторов, то вам придется писать и новый файл грамматики. Для того чтобы генерируемые синтаксические анализаторы были способны на что-то большее, чем простое распознавание грамматики, большинство генераторов синтаксических анализаторов используют в файле грамматики внешний код (*Foreign Code* (315)) для действий.

После того как вы создали файл грамматики, генератор синтаксических анализаторов на его основе создает анализатор. Большинство генераторов синтаксических анализаторов создают код, что может позволить вам генерировать синтаксические анализаторы для разных базовых языков программирования. Конечно, нет никаких причин, по которым генератор синтаксических анализаторов был бы не в состоянии прочесть файл грамматики и интерпретировать его во время выполнения приложения, возможно, путем создания комбинатора синтаксических анализаторов (*Parser Combinator* (263)). Генераторы синтаксических анализаторов создают код из-за сочетания традиций и соображений производительности, в частности потому, что они обычно ориентированы на языки программирования общего назначения.

Чаще всего сгенерированный код рассматривается как “черный ящик”, и вам не придется в него вникать. Однако иногда полезно разобраться в том, как работает полученный синтаксический анализатор, особенно если вы пытаетесь отлаживать грамматику. В этом случае преимущество получают генераторы синтаксических анализаторов, которые используют более простой для понимания алгоритм, например создающие синтаксические анализаторы рекурсивного спуска (*Recursive Descent Parser* (253)).

В этой книге многие шаблоны проиллюстрированы с помощью генератора синтаксических анализаторов ANTLR. Я обычно рекомендую ANTLR для новичков в этой области из-за доступности и хорошей документированности этого тщательно продуманного инструмента. Для него имеется также неплохая интегрированная среда разработки (*ANTLRWorks*), предоставляющая очень удобный пользовательский интерфейс для разработки грамматик.

23.1.1. Встроенные действия

Синтаксический анализ создает синтаксическое дерево; чтобы сделать с ним что-то полезное, нужно добавить в синтаксический анализатор дополнительный код. Мы размещаем его в грамматике с использованием шаблона *Foreign Code* (315). Местоположение фрагментов кода в грамматике указывает, когда этот код выполняется. Встроенный код помещается в выражения тех правил, вследствие распознавания которых он должен выполняться.

Давайте рассмотрим пример регистрации событий при работе с их объявлениями.

```
eventBlock : Event-keyword eventDec* End-keyword;
eventDec   : Identifier Identifier {registerEvent($1, $2);}
```

Здесь говорится, что после того, как синтаксический анализатор распознает второй идентификатор в *eventDec*, будет вызван метод *registerEvent*. Для того чтобы передать данные из дерева разбора в *registerEvent*, нужен способ обращения к токенам,

упоминающимся в правиле. В данном случае я использую \$1 и \$2 для указания идентификаторов по их позициям — в стиле генератора синтаксических анализаторов Yacc.

Действия обычно вставляются в создаваемые синтаксические анализаторы в процессе их генерации. В результате внедренный код, как правило, использует тот же язык программирования, что и созданный синтаксический анализатор.

Различные генераторы синтаксических анализаторов имеют различные возможности встраивания кода и связывания действий с правилами. Я не намерен рассматривать различные возможности множества инструментов, но упомяну о паре из них. Я уже говорил о связи встраиваемого кода с идентификаторами. Так как по своей природе синтаксический анализатор предназначается для создания синтаксического дерева, часто полезны перемещения данных по дереву. Поэтому распространена практика обеспечения подправилу возможности вернуть данные родительскому правилу. Для иллюстрации рассмотрим следующую грамматику, использующую генератор синтаксических анализаторов ANTLR.

```
eventBlock
: K_EVENT (e = eventDec {registerEvent($e.result);})* K_END
;
eventDec returns [Event result]
: name = ID code = ID {$result = createEvent($name, $code);}
;
```

Здесь правило eventDec описывается как возвращающее значение, которое может использовать правило более высокого уровня. (В ANTLR действия обращаются к элементам грамматики, что, как правило, лучше, чем обращение по местоположению.) Способность возвращать значения из правил может существенно облегчить написание синтаксических анализаторов, в частности так можно избежать многих переменных контекста (Context Variable (187)). Некоторые генераторы синтаксических анализаторов, в том числе ANTLR, имеют также возможность передавать данные вниз в качестве аргументов подправил, что обеспечивает большую гибкость в предоставлении им контекста.

Этот фрагмент также показывает, что размещение действий в грамматике определяет момент их вызова. Здесь действие расположено в средине правой части eventBlock, указывая, что оно должно быть вызвано после распознавания каждого под правила eventDec. Такое размещение действий — распространенная практика при использовании генераторов синтаксических анализаторов.

В ходе применения синтаксически управляемой трансляции (Syntax-Directed Translation (229)) возникает распространенная проблема, с которой мне приходилось сталкиваться, — слишком большой встроенный код в грамматике. При этом затрудняется понимание структуры грамматики, базовый код, исходный текст сложно редактировать, а для тестирования и отладки требуется регенерация синтаксического анализатора. Ключевой шаблон для решения этой проблемы — Embedment Helper (537): переместите как можно больше кода во вспомогательный объект. Код в грамматике должен представлять собой вызовы отдельных методов.

Действия определяют, что мы делаем с DSL, поэтому способ их написания зависит от общего подхода к синтаксическому анализу DSL: построение дерева (Tree Construction (289)), встроенная интерпретация (Embedded Interpretation (311)) или встроенная трансляция (Embedded Translation (305)). Сам по себе генератор синтаксических анализаторов без этих шаблонов мало интересен. Однако в данном обсуждении вы не найдете соответствующих примеров — вместо них давайте познакомимся с другими шаблонами.

Вот еще одно животное из этого зоопарка, очень похожее на действие. **Семантический предикат**, подобно действию, представляет собой блок внешнего кода, но он возвращает

логическое значение, указывающее, является ли синтаксический анализ правила успешным. Действия же, в отличие от семантических предикатов, на процесс синтаксического анализа не влияют. Обычно семантические предикаты используют, когда имеют дело с частями грамматики, которые не могут быть корректно описаны в грамматике языка. Они обычно появляются в более сложных языках, чаще всего — в языках общего назначения. Но если у вас возникли трудности при создании грамматики для работы с самой грамматикой DSL, то семантические предикаты открывают двери для более сложной обработки.

23.2. Когда это использовать

Для меня главное преимущество использования генератора синтаксических анализаторов состоит в том, что для определения синтаксической структуры обрабатываемого языка применяется явно выраженная грамматика. Это, конечно, ключевое преимущество использования DSL. Поскольку генераторы синтаксических анализаторов предназначены, в первую очередь, для обработки сложных языков, они предоставляют гораздо больше возможностей и мощи, чем вы получите, написав собственный синтаксический анализатор. Хотя эти возможности могут потребовать дополнительных усилий на их изучение, можно начать работу с простейшего их набора и постепенно усложнять свой синтаксический анализатор. Генераторы синтаксических анализаторов могут обеспечить хорошую обработку ошибок и диагностику, которые, хотя я и не говорю о них в книге, могут иметь большое значение при попытках разобраться, почему ваши грамматики делают не то, что, как вы думаете, должны.

Есть у генераторов синтаксических анализаторов и некоторые минусы. Вы можете находиться в языковой среде, где нет генератора синтаксических анализаторов, а это не та вещь, которую легко написать самому. Даже если он есть, вы можете задуматься, стоит ли добавлять в свой проект еще один новый инструмент. Поскольку генераторы синтаксических анализаторов обычно прибегают к созданию кода, они усложняют процесс сборки, что также может стать дополнительным раздражителем.

23.3. Hello World (Java и ANTLR)

В мире программ имеется традиция — всякий раз, начиная изучение нового языка программирования, первой писать программу “Hello World”. Это хорошая привычка, потому что, если вы не знакомы с новой средой программирования, обычно возникают вопросы, в которых нужно разобраться, прежде чем вы сможете запустить даже простейшую программу.

Генератор синтаксических анализаторов ANTLR почти такой же. Поэтому лучше начать с реально простой вещи, просто чтобы убедиться в том, что вы знаете, что и как работает. Так как в ряде примеров этой книги я использую ANTLR, я думаю, что здесь, где рассматриваются генераторы синтаксических анализаторов, также стоит поработать именно с ним, тем более что базовые шаги нужно знать в любом случае, для работы с любым генератором синтаксических анализаторов.

Основная рабочая модель генератора синтаксических анализаторов заключается в следующем. Вы пишете файл грамматики и запускаете генератор синтаксических анализаторов для получения исходного текста синтаксического анализатора этой грамматики. Затем выполняется компиляция полученного исходного текста вместе с другим кодом, который предназначен для совместной работы с синтаксическим анализатором. После этого вы можете анализировать файлы с исходными текстами на соответствующем языке.

23.3.1. Написание базовой грамматики

Поскольку мы хотим анализировать текст, нам понадобится некоторый действитель-но простой текст для анализа. Вот такой файл.

```
greetings.txt...
hello Rebecca
hello Neal
hello Ola
```

Я рассматриваю его как список приветствий, каждое из которых представляет собой ключевое слово (`hello`), за которым следует имя. Вот простая грамматика для распознавания такого текста.

```
Greetings.g...
grammar Greetings;

@header {
    package helloAntlr;
}

@lexer::header {
    package helloAntlr;
}

script    : greeting* EOF;
greeting : 'hello' Name;

Name      : ('a'...'z' | 'A'...'Z')+;

WS        : (' ' | '\t' | '\r' | '\n')+ {skip();} ;
COMMENT   : '#'(~'\n')* {skip();} ;
ILLEGAL   : .;
```

Файл грамматики прост, но не настолько, насколько мне хотелось бы.

В первой строке объявляется имя грамматики.

```
grammar Greetings;
```

Если только я не хочу разместить все в пакете по умолчанию (пустом), необходимо обеспечить, чтобы создаваемый синтаксический анализатор оказался в надлежащем пакете, в нашем случае — `helloAntlr`. Я делаю это с помощью атрибута грамматики `@header`, который обеспечивает вставку некоторого кода Java в заголовок создаваемого синтаксического анализатора. Если мне нужно добавить некоторые инструкции импорта, я также сделаю это здесь.

```
@header {
    package helloAntlr;
}
```

То же самое я делаю с кодом лексического анализатора.

```
@lexer::header {
    package helloAntlr;
}
```

Теперь перейдем к правилам, которые являются главными в файле. Хотя ANTLR, как и большинство генераторов синтаксических анализаторов, использует раздельные лексический и синтаксический анализаторы, оба они генерируются из одного файла. Первые две строки гласят, что сценарий представляет собой несколько приветствий, за которыми

следует конец файла, и что приветствие представляет собой токен ключевого слова hello, за которым следует токен Name.

```
script : greeting* EOF;
greeting : 'hello' Name;
```

ANTLR выделяет токены, начиная их с заглавной буквы. Имя — это просто строка из букв.

```
Name : ('a'..'z' | 'A'..'Z')+;
```

Я обычно нахожу полезным избавиться от пробелов и объявить комментарии. Когда что-то идет не так, комментарии становятся грубым, но надежным помощником в отладке.

```
WS : (' ' | '\t' | '\r' | '\n')+ {skip();} ;
COMMENT : '#'(~'\n')* {skip();} ;
ILLEGAL : .;
```

Последнее правило (ILLEGAL) заставляет лексический анализатор сообщить об ошибке, если он встречает токен, не вписывающийся ни в одно из его правил (в противном случае такие токены молча игнорируются).

В данный момент, если вы используете интегрированную среду разработки ANTLRWorks, у вас уже имеется достаточно, чтобы запустить интерпретатор ANTLR и убедиться в том, что он читает текст. Следующим шагом является генерация и запуск основного синтаксического анализатора.

(Мелочь, конечно, но... Если вы не поместите символ конца файла (EOF) в конце верхнего правила, ANTLR не будет сообщать об ошибке. Он просто остановит синтаксический анализ в первой же точке, где возникли неприятности, и даже не подумает, что что-то идет не так. Это особенно неприятно, потому что ANTLRWorks, когда такое случается, показывает ошибку в своем интерпретаторе, так что очень легко запутаться, разочароваться и совершить акт насилия по отношению к своему монитору (а как еще прикажете называть удар по нему чем-то тяжелым?).)

23.3.2. Построение синтаксического анализатора

Следующий шаг состоит в запуске генератора кода ANTLR для генерации исходных файлов. В этот момент наступает время для решения вопроса со средой построения. Стандартной системой сборки проектов Java является Ant, так что ее я и буду использовать в данном примере (хотя лично мне больше нравится Rake).

Для генерации исходных файлов я запускаю инструмент ANTLR, содержащийся в библиотечных JAR-файлах.

```
build.xml...
<property name="dir.src" value="src"/>
<property name="dir.gen" value="gen"/>
<property name="dir.lib" value="lib"/>
<path id="path.antlr">
    <fileset dir="${dir.lib}">
        <include name="antlr*.jar"/>
        <include name="stringtemplate*.jar"/>
    </fileset>
</path>
<target name="gen">
    <mkdir dir="${dir.gen}/helloAntlr"/>
    <java classname="org.antlr.Tool" classpathref="path.antlr"
          fork="true" failonerror="true">
        <arg value="-fo"/>
```

```

<arg value="\$\{dir.gen\}/helloAntlr"/>
<arg value="\$\{dir.src\}/helloAntlr/Greetings.g"/>
</java>
</target>

```

Эти настройки размещают различные генерируемые исходные файлы ANTLR в каталоге `gen`. Этот каталог я держу отдельно от моих основных исходных файлов. Так как эти файлы генерируются, я указываю своей системе контроля версий игнорировать их.

Генератор кода создает ряд исходных файлов. Для наших целей ключевыми являются исходные файлы Java лексического анализатора (`GreetingsLexer.java`) и синтаксического анализатора (`GreetingsParser.java`).

Это сгенерированные файлы. Следующим шагом будет их использование. Я предполагаю писать для этого собственный класс. Я называю его загрузчиком приветствий, так как ANTLR уже использует слово “анализатор”. Я снабжаю класс считывателем входного потока.

```

class GreetingsLoader...
private Reader input;
public GreetingsLoader(Reader input) {
    this.input = input;
}

```

Затем я пишу метод `run`, который управляет работой всех сгенерированных ANTLR файлов.

```

class GreetingsLoader...
public List<String> run() {
    try {
        GreetingsLexer lexer =
            new GreetingsLexer(new ANTLRReaderStream(input));
        GreetingsParser parser =
            new GreetingsParser(new CommonTokenStream(lexer));
        parser.script();
        return guests;
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (RecognitionException e) {
        throw new RuntimeException(e);
    }
}
private List<String> guests = new ArrayList<String>();

```

Основная идея в том, что сначала я создаю лексический анализатор на основе считывателя входного потока, а затем — синтаксический анализатор на основе лексического анализатора. После этого я вызываю метод синтаксического анализатора с тем же именем, что и у правила верхнего уровня моей грамматики. Это приводит к выполнению синтаксического анализа входного текста.

Я могу запустить получившийся синтаксический анализатор из простого теста.

```

@Test
public void readsValidFile() throws Exception {
    Reader input =
        new FileReader("src/helloAntlr/greetings.txt");
    GreetingsLoader loader =
        new GreetingsLoader(input);
    loader.run();
}

```

Все отлично работает, но от этой работы очень мало пользы. Все, что мы видим, — генерированный синтаксический анализатор без ругани работает со входным файлом. Но это не говорит даже о том, что входной файл считан без проблем. Так что было бы полезно передать синтаксическому анализатору некорректный входной файл.

```
invalid.txt...
hello Rebecca
XXhello Neal
hello Ola

test...
@Test
public void errorWhenKeywordIsMangled() throws Exception {
    Reader input =
        new FileReader("src/helloAntlr/invalid.txt");
    GreetingsLoader loader = new GreetingsLoader(input);
    try {
        loader.run();
        fail();
    } catch (Exception expected) {}
}
```

С таким кодом тест не будет выполнен. ANTLR выведет предупреждение о наличии неприятностей, но ANTLR ориентирован на продолжение синтаксического анализа и максимальное восстановление после ошибок. В общем случае это неплохо, но (особенно на раннем этапе) такая терпимость ANTLR к ошибкам может оказаться неприятным сюрпризом.

Таким образом, в данный момент у нас есть проблемы. Во-первых, все, что делает синтаксический анализатор, — это читает файл, но не производит никаких выходных данных. Во-вторых, трудно определить наличие ошибок. Я могу исправить обе эти проблемы путем добавления своего кода в файл грамматики.

23.3.3. Добавление кода действий в грамматику

В ходе применения синтаксически управляемой трансляции (Syntax-Directed Translation (229)) можно использовать три стратегии для получения некоторого выхода: построение дерева (Tree Construction (289)), встроенную интерпретацию (Embedded Interpretation (311)) и встроенную трансляцию (Embedded Translation (305)). Я обычно предпочитаю встроенную трансляцию — это дает мне простой способ увидеть, что происходит.

При использовании кода действия я обычно следую шаблону `Embedment Helper` (537). Простейший способ его применения с ANTLR — взять уже имеющийся у меня класс загрузчика и добавить его в грамматику. При этом я также могу сделать кое-что для лучшего уведомления пользователя об ошибках.

Первая стадия этого процесса заключается в модификации файла грамматики — вставке Java-кода в генерируемый синтаксический анализатор. Я использую атрибут `members` для объявления вспомогательного класса и перекрываю функцию обработки ошибок по умолчанию функцией отчета об ошибке.

```
Greetings.g...
@members {
    GreetingsLoader helper;
    public void reportError(RecognitionException e) {
        helper.reportError(e);
    }
}
```

В загрузчике я могу предоставить простую реализацию функции отчета об ошибках, которая записывает информацию о них.

```
class GreetingsLoader...
private List errors = new ArrayList();
void reportError(RecognitionException e) {
    errors.add(e);
}
public boolean hasErrors() {return !isOk();}
public boolean isOk() {return errors.isEmpty();}
private String errorReport() {
    if (isOk()) return "OK";
    StringBuffer result = new StringBuffer("");
    for (Object e : errors)
        result.append(e.toString()).append("\n");
    return result.toString();
}
```

Теперь все, что я должен сделать, — это добавить пару строк в метод `run` для настройки вспомогательного класса и генерации исключения, если синтаксический анализатор сообщает об ошибках.

```
class GreetingsLoader...
public void run() {
    try {
        GreetingsLexer lexer =
            new GreetingsLexer(new ANTLRReaderStream(input));
        GreetingsParser parser =
            new GreetingsParser(new CommonTokenStream(lexer));
        parser.helper = this;
        parser.script();
        if (hasErrors())
            throw new RuntimeException("it all went pear-shaped\n"
                + errorReport());
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (RecognitionException e) {
        throw new RuntimeException(e);
    }
}
```

При наличии вспомогательного класса можно легко добавить код действия для отчета о людях, которых я поприветствовал.

```
Greetings.g...
greeting : 'hello' n=Name {helper.recordGuest($n);}

class GreetingsLoader...
void recordGuest(Token t) {guests.add(t.getText());}
List<String> getGuests() {return guests;}
private List<String> guests = new ArrayList<String>();

test...
@Test
public void greetedCorrectPeople() throws Exception {
    Reader input =
        new FileReader("src/helloAntlr/greetings.txt");
    GreetingsLoader loader = new GreetingsLoader(input);
    loader.run();
    List<String> expectedPeople =
        Arrays.asList("Rebecca", "Neal", "Ola");
    assertEquals(expectedPeople, loader.getGuests());
}
```

Все это очень “сыро”, но этого достаточно, чтобы убедиться, что генератор синтаксических анализаторов создает нечто, что будет разбирать файл, находить ошибки и передавать информацию для вывода. Убедившись, что этот простой пример работает, я могу добавить в него другую полезную функциональность.

23.3.4. Применение шаблона Generation Gap

Еще один способ включения вспомогательных методов и обработки ошибок в синтаксический анализатор заключается в использовании шаблона **Generation Gap** (561). При этом я вручную создаю суперкласс для генерируемого ANTLR синтаксического анализатора. После этого сгенерированный синтаксический анализатор может использовать вспомогательные методы как обычные вызовы собственных методов.

Для этого требуется добавить соответствующее указание в файл грамматики. Теперь вся грамматика принимает следующий вид.

```
Greetings.g...
grammar Greetings;
options {superClass = BaseGreetingsParser; }

@header {
    package subclass;
}

@lexer::header {
    package subclass;
}

script      : greeting * EOF;
greeting   : 'hello' n=Name {recordGuest($n)} ;

Name       : ('a'..'z' | 'A'..'Z')+;
WS         : (' ' | '\t' | '\r' | '\n')+ {skip();} ;
COMMENT   : '#'(~'\n')* {skip();} ;
ILLEGAL   : . ;
```

Мне больше не нужно перекрывать `reportError`, так как я буду делать это в своем рукописном суперклассе. Вот этот суперкласс.

```
abstract public class BaseGreetingsParser extends Parser {
    public BaseGreetingsParser(TokenStream input) {
        super(input);
    }

    //----- Вспомогательные методы
    void recordGuest(Token t) {guests.add(t.getText());}
    List<String> getGuests() { return guests; }
    private List<String> guests = new ArrayList<String>();

    //----- Обработка ошибок
    private List errors = new ArrayList();

    public void reportError(RecognitionException e) {
        errors.add(e);
    }

    public boolean hasErrors() {return !isOk();}
    public boolean isOk() {return errors.isEmpty();}
```

Этот класс является подклассом базового класса анализатора ANTLR, поэтому он вносит сам себя в иерархию выше сгенерированного синтаксического анализатора. Рукописный класс содержит вспомогательный код и код отчета об ошибках, который раньше находился в отдельном классе загрузчика. Тем не менее лучше все же оставить класс-оболочку для координации работы анализатора.

```
class GreetingsLoader...
    private Reader input;
    private GreetingsParser parser;

    public GreetingsLoader(Reader input) {
        this.input = input;
    }

    public void run() {
        try {
            GreetingsLexer lexer =
                new GreetingsLexer(new ANTLRReaderStream(input));
            parser =
                new GreetingsParser(new CommonTokenStream(lexer));
            parser.script();
            if (parser.hasErrors())
                throw new RuntimeException("it all went pear-shaped");
        } catch (IOException e) {
            throw new RuntimeException(e);
        } catch (RecognitionException e) {
            throw new RuntimeException(e);
        }
    }

    public List<String> getGuests() {
        return parser.getGuests();
    }
```

Как отношение наследования, так и отношение делегирования имеют свои сильные стороны для применения в шаблоне **Embedment Helper** (537). У меня нет твердого мнения о том, какое из них лучше для использования, и в примерах из этой книги вы встретите оба варианта.

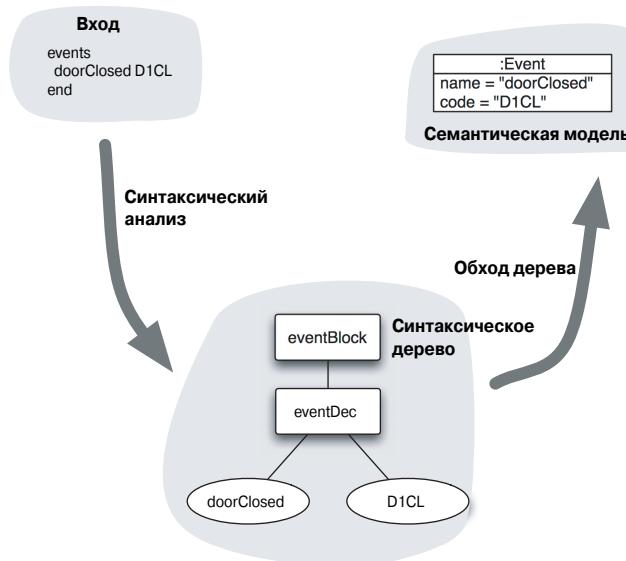
Для реальной работы предстоит сделать еще многое, но я думаю, что эти простые примеры послужат хорошей отправной точкой. Вы сможете найти больше примеров использования генераторов синтаксических анализаторов в остальных шаблонах из этой части книги.

Глава 24

Построение дерева

Tree Construction

Синтаксический анализатор создает и возвращает представление исходного текста в виде синтаксического дерева, с которым позже работают путем его обхода



24.1. Как это работает

Любой синтаксический анализатор с синтаксически управляемой трансляцией (Syntax-Directed Translation (229)) строит в процессе анализа синтаксическое дерево. Дерево строится на стеке, с обрезкой ветвей, когда работа с ними завершена. В случае же применения шаблона построения дерева мы создаем действия синтаксического анализатора, которые в процессе анализа строят синтаксическое дерево в памяти. После завер-

шения синтаксического анализа у нас получается полное синтаксическое дерево для сценария DSL. Затем мы можем выполнять с этим деревом дальнейшие операции. Если мы используем семантическую модель (*Semantic Model* (171)), то запускаем код, который обходит это дерево и наполняет семантическую модель.

Создаваемое в памяти синтаксическое дерево не обязано точно соответствовать фактическому синтаксическому дереву, которое в процессе работы создает синтаксический анализатор; на самом деле они обычно отличаются друг от друга. Вместо этого мы строим то, что называется **абстрактным синтаксическим деревом**. Абстрактное синтаксическое дерево представляет собой упрощение синтаксического дерева, которое обеспечивает лучшее древовидное представление входного языка.

Давайте для ясности рассмотрим небольшой пример. Я буду использовать объявление событий для моего примера конечного автомата.

```
events
  doorClosed D1CL
  drawOpened D2OP
end
```

Я анализирую этот текст с применением грамматики

```
declarations : eventBlock commandBlock;
eventBlock   : Event-keyword eventDec* End-keyword;
eventDec     : Identifier Identifier;
commandBlock : Command-keyword commandDec* End-keyword;
commandDec   : Identifier Identifier;
```

и получаю синтаксическое дерево, показанное на рис. 24.1.

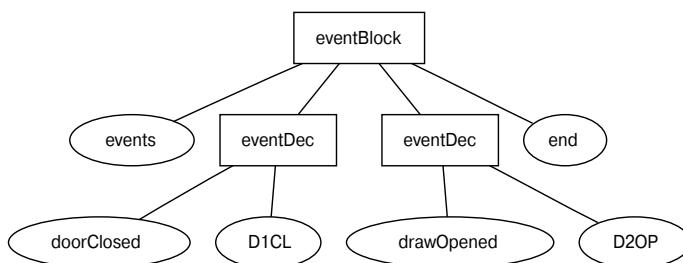


Рис. 24.1. Синтаксическое дерево для объявлений событий

Взглянув на это дерево, вы должны сообразить, что узлы `events` и `end` не являются необходимыми. Эти слова нужны во входном тексте для того, чтобы отметить границы объявлений событий, но как только в ходе анализа мы превращаем исходный текст в древовидную структуру, они больше не нужны и только загромождают структуру данных. Вместо этого мы могли бы представить исходный текст синтаксическим деревом, показанным на рис. 24.2.

Это дерево не является точным представлением исходного текста, но это именно то, что нам нужно для обработки событий. Это абстракция исходного текста, которая лучше всего подходит для наших целей. Очевидно, для разных целей могут потребоваться различные абстрактные синтаксические деревья; если все, что нам надо, — только список кодов событий, то можно отказаться от узлов названий и `eventDec` и хранить только коды.

Сейчас я должен уточнить терминологию. Я использую термин **синтаксическое дерево** для описания иерархической структуры данных, формируемой путем синтаксического анализа некоторых входных данных. Я часто использую термин “синтаксическое дерево”

как обобщенный термин — так, абстрактные синтаксические деревья являются подмножеством синтаксических деревьев.

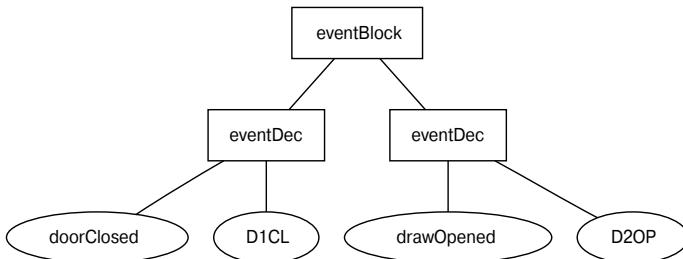


Рис. 24.2. Абстрактное синтаксическое дерево для объявлений событий

Для того чтобы построить синтаксическое дерево, можно использовать коды действий в БНФ (BNF (237)). При таком подходе очень удобной оказывается способность действий возвращать значения для узла — каждое действие добавляет представление своего узла в результирующее синтаксическое дерево.

Некоторые генераторы синтаксических анализаторов (Parser Generator (277)) идут еще дальше, предлагая DSL для определения синтаксического дерева. В ANTLR, например, можно создать показанное выше абстрактное синтаксическое дерево, используя следующее правило.

```
eventDec : name=ID code=ID -> ^ (EVENT_DEC $name $code) ;
```

Оператор `->` вводит правило построения дерева. Тело правила представляет собой список, в котором первым элементом является тип узла (`EVENT_DEC`), а за ним следуют дочерние узлы, которые в данном случае являются токенами имени и кода.

Использование DSL для построения дерева может значительно упростить создание абстрактного синтаксического дерева. Часто генераторы синтаксических анализаторов, поддерживающие эту возможность, дают вам по умолчанию (если вы не предоставите какие-либо правила построения дерева) полное и точное синтаксическое дерево, но оно почти никогда не нужно. Обычно предпочтительнее упростить его до абстрактного синтаксического дерева с помощью указания правил построения.

Построенное таким образом абстрактное синтаксическое дерево будет состоять из обобщенных объектов, в которых хранятся данные дерева. В приведенном выше примере `eventDec` представляет собой такой обобщенный узел дерева с именем и кодом в качестве дочерних узлов. И имя, и код являются обобщенными токенами. Если вы самостоятельно строите дерево с помощью действий, то можете создавать реальные объекты — такие, как истинный объект события с именем и кодом в виде полей. Я предпочитаю получить обобщенное абстрактное синтаксическое дерево, а затем на втором этапе обработки использовать его для преобразования в семантическую модель. Мне кажется, что легче выполнить два простых преобразования, чем одно сложное.

24.2. Когда это использовать

И построение дерева, и встраиваемая трансляция (Embedded Translation (305)) представляют собой полезные подходы к наполнению семантической модели (Semantic Model (171)) в процессе анализа. Встроенная трансляция делает преобразование за один шаг, в то время как построение дерева использует два этапа с абстрактным синтаксическим деревом

в качестве промежуточной модели. Аргументом в пользу построения дерева является то, что одно сложное преобразование разбивается на две попроще. Стоит ли такое разбиение усилий по борьбе с промежуточной моделью — во многом зависит от сложности преобразования. Чем сложнее преобразование, тем больше пользы от его разделения.

При особой сложности можно сделать несколько проходов по сценарию DSL. В процессе однопроходного анализа сложнее работать с такими вещами, как, например, опережающие ссылки. При построении синтаксического дерева его можно легко много-кратно обойти в качестве части последующей обработки.

Другим фактором, который говорит в пользу использования построения дерева, является наличие в вашем генераторе синтаксических анализаторов (Parser Generator (277)) инструментария, позволяющего легко создавать абстрактные синтаксические деревья. Некоторые генераторы синтаксических анализаторов просто не дают вам выбора — при их использовании вы вынуждены строить дерево. Большинство генераторов позволяют использовать встроенную трансляцию, но если генератор синтаксических анализаторов позволяет очень легко построить абстрактное синтаксическое дерево, то такое построение становится более привлекательным вариантом.

При построении дерева, вероятно, потребляется больше памяти, чем при альтернативных подходах, поскольку при этом необходимо хранить в памяти все дерево. В большинстве случаев, однако, это не является важным фактором (хотя, конечно же, было крайне важным в былые времена).

Одно и то же абстрактное синтаксическое дерево можно обработать различными способами для заполнения различных семантических моделей, т.е. повторно использовать результаты работы синтаксического анализатора. Это может оказаться удобным, но если построение дерева синтаксическим анализатором выполняется очень просто, то может оказаться эффективнее применять для разных целей разные абстрактные синтаксические деревья. Может также оказаться неплохим решением преобразование в одну семантическую модель, которая затем применяется как основа для преобразования в другие представления.

24.3. Использование синтаксиса построения дерева ANTLR (Java и ANTLR)

Я вновь воспользуюсь DSL любимого конечного автомата, а также контроллером мисс Грант. Вот конкретный исходный текст, который будет использован мною в данном примере.

```

events
  doorClosed    D1CL
  drawerOpened  D2OP
  lightOn       L1ON
  doorOpened    D1OP
  panelClosed   PNCL
end

resetEvents
  doorOpened
end

commands
  unlockPanel  PNUL
  lockPanel    PNLK
  lockDoor     D1LK

```

```

    unlockDoor D1UL
end

state idle
  actions {unlockDoor lockPanel}
  doorClosed => active
end

state active
  drawerOpened => waitingForLight
  lightOn      => waitingForDrawer
end

state waitingForLight
  lightOn => unlockedPanel
end

state waitingForDrawer
  drawerOpened => unlockedPanel
end

state unlockedPanel
  actions {unlockPanel lockDoor}
  panelClosed => idle
end

```

24.3.1. Токенизация

Токенизация в данном случае очень проста. У нас есть несколько ключевых слов (`events`, `end` и т.п.) и набор идентификаторов. ANTLR позволяет помещать ключевые слова в правила грамматики в виде литералов. Так что нам нужны правила лексического анализа только для идентификаторов.

```

fragment LETTER : ('a'..'z' | 'A'..'Z' | '_');
fragment DIGIT  : ('0'..'9');

ID           : LETTER (LETTER | DIGIT)* ;

```

Строго говоря, лексические правила для имен и кодов различны — имена могут быть любой длины, но код должен состоять из четырех заглавных букв. Таким образом, мы могли бы определить для них различные правила. Однако в этом случае возникают сложности. Стока `ABC1` является правильным кодом, но одновременно и правильным именем. Если мы видим `ABC1` в DSL-программе, по контексту мы можем определить, что это: `state ABC1` отличается от `event unlockDoor ABC1`. Синтаксический анализатор также имеет возможность использовать разницу в контексте, но лексический анализатор этого не умеет. Так что наилучший вариант заключается в использовании одного и того же токена в обоих случаях и в предоставлении возможности принять решение анализатору. Это означает, что синтаксический анализатор не будет генерировать ошибки для пятибуквенных кодов — мы должны сами проверять длину кода в процессе семантической обработки.

Необходимы также правила лексического анализа для удаления пробельных символов.

```

WHITE_SPACE : (' ' | '\t' | '\r' | '\n')+ {skip();} ;
COMMENT     : '#' ~'\n'* '\n' {skip();};

```

В данном случае пробельные символы включают в себя символы новой строки. Я форматировал текст DSL так, чтобы можно было предположить значимость отступов и построчного размещения инструкций, но, как видите, это вовсе не так. Все пробелы, в том числе окончания строк, удаляются. Это позволяет мне форматировать исходный текст DSL любым подходящим мне способом. В этом заключается заметное отличие от трансляции, управляемой разделителями (*Delimiter-Directed Translation* (213)). На самом деле следует отметить, что, в отличие от большинства языков общего назначения, здесь нет никаких разделителей инструкций вообще. Такое не редкость в DSL, так как обычно его инструкции весьма ограничены. Такие вещи, как инфиксные выражения, заставляют использовать разделители инструкций, но во многих DSL можно обойтись и без них. Здесь, как и в жизни, не надо тянуть за собой то, без чего можно обойтись.

В этом примере я полностью опускаю все пробельные символы, а это означает, что для синтаксического анализатора они полностью потеряны. Это разумно, так как синтаксическому анализатору они не нужны, все его потребности ограничиваются значимыми токенами. Однако есть случай, когда пробелы могут пригодиться — когда дела идут плохо. Чтобы дать хорошее информативное сообщение об ошибке, желательно указать номера строки и столбца, а для этого следует учитывать пробельные символы. ANTLR позволяет сделать это, посыпая пробельные токены в другой канал, с помощью синтаксиса наподобие `WS : ('\\r' | '\\n' | ' ' | '\\t')+ { $channel=HIDDEN }.` Так пробельные символы отправляются в скрытый канал, что позволяет использовать их при обработке ошибок, но не влияет на правила синтаксического анализа.

24.3.2. Синтаксический анализ

Правила лексического анализа работают в ANTLR одинаково, независимо от того, используете ли вы построение дерева. По-разному работает только синтаксический анализатор. Чтобы использовать построение дерева, нужно указать ANTLR о необходимости создания абстрактного синтаксического дерева.

```
options {
    output=AST;
    ASTLabelType = MfTree;
}
```

Приказав ANTLR строить абстрактное синтаксическое дерево, я также указываю, что оно должно быть заполнено узлами определенного типа — `MfTree`. Это подкласс обобщенного класса `ANTLR CommonTree`, класс, который позволяет добавить некоторое поведение, которое я хотел бы иметь у узлов дерева. Здесь есть некоторая путаница. Класс представляет узел и его дочерние узлы, так что вы можете рассматривать его как узел или дерево (поддерево). В ANTLR он называется деревом, и я следую этому в своем коде, хотя и рассматриваю его как узел в дереве.

Теперь перейдем к правилам грамматики. Начнем с правила верхнего уровня, которое определяет структуру всего DSL-файла.

```
machine : eventList resetEventList? commandList? state*;
```

Это правило перечисляет основные части в определенной последовательности. Если я не предоставлю ANTLR никаких правил построения дерева, он будет просто возвращать узлы последовательно для каждого члена из правой части правила. Обычно это не то, чего мы хотим, но в данном случае нас это устраивает.

Как видите, в исходном тексте первыми идут события.

```
eventList : 'events' event* 'end' -> ^(EVENT_LIST event*) ;
event      : n=ID c=ID -> ^(EVENT $n $c) ;
```

Есть несколько новых нюансов, о которых следует упомянуть, говоря об этих двух правилах. Один из нюансов заключается в том, что эти правила вводят синтаксис ANTLR для построения дерева, который представляет собой код после "`->`" в каждом правиле.

Правило `eventList` использует две строковые константы — так выглядит передача токенов ключевых слов непосредственно в правила синтаксического анализатора, без отдельных правил для лексического анализатора.

Правила построения дерева позволяют указать, что должно быть в абстрактном синтаксическом дереве. В обоих приведенных здесь случаях мы используем `^(list...)`, чтобы создать и вернуть новый узел в абстрактном синтаксическом дереве. Первым элементом в скобках является тип токена узла. В данном случае мы создали новый тип токена. Все элементы после типа токена являются другими узлами дерева. В случае списка событий мы просто помещаем все события в список в качестве "братьев". Токены событий мы именуем в БНФ и ссылаемся на них при построении дерева, чтобы показать их расположение.

Токены `EVENT_LIST` и `EVENT` являются специальными токенами, которые я создаю в рамках синтаксического анализа (эти токены лексическим анализатором не производятся). Создавая токены такого рода, их нужно объявить в файле грамматики.

```
tokens { EVENT_LIST; EVENT; COMMAND_LIST; COMMAND;
          STATE; TRANSITION_LIST; TRANSITION; ACTION_LIST;
          RESET_EVENT_LIST;
}
```

Команды рассматриваются аналогично, а сбрасывающие события представляют собой простой список.

```
commandList : 'commands' command*
              'end' -> ^(COMMAND_LIST command*) ;
command : ID ID -> ^(COMMAND ID+) ;

resetEventList : 'resetEvents' ID*
                  'end' -> ^RESET_EVENT_LIST ID*) ;
```

Состояния обрабатываются принципиально точно так же, хотя и включают немного больше сущностей.

```
state
  : 'state' ID actionList? transition* 'end'
  -> ^STATE ID ^ACTION_LIST actionList?
      ^TRANSITION_LIST transition* )
;

transition : ID '=>' ID -> ^TRANSITION ID+ ;
actionList : 'actions' '{' ID* '}' -> ID* ;
```

Каждый раз мои действия сводятся к накоплению и сбору соответствующих групп DSL и их размещению в узле, который описывает, что представляет собой эта группа. В результате получается абстрактное синтаксическое дерево, которое хотя и очень похоже на полное синтаксическое дерево, но все же отличается от него (рис. 24.3). Моя цель — сделать так, чтобы правила построения дерева оставались очень простыми, а синтаксическое дерево можно было легко обойти.

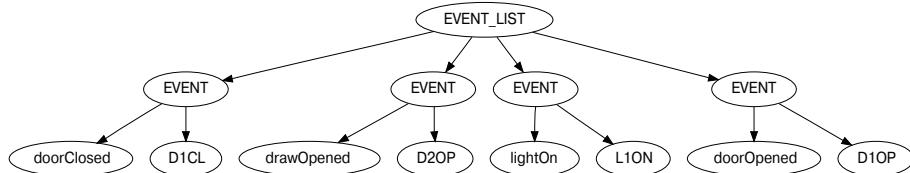


Рис. 24.3. Абстрактное синтаксическое дерево списка события контроллера мисс Грант

24.3.3. Наполнение семантической модели

Как только анализатор построил дерево, следующим шагом будет обход этого дерева и наполнение семантической модели (*Semantic Model (171)*). Используемая семантическая модель — это та же модель конечного автомата, которую я использовал во введении. Интерфейс построения довольно прост, поэтому я не буду вдаваться здесь в подробности.

Я создаю класс загрузчика для наполнения семантической модели.

```

class StateMachineLoader...
private Reader input;
private MfTree ast;
private StateMachine machine;

public StateMachineLoader(Reader input) {
    this.input = input;
}
  
```

Я использую загрузчик как командный класс. Вот его метод `run`, который указывает последовательность шагов, используемых для выполнения трансляции.

```

class StateMachineLoader...
public void run() {
    loadAST();
    loadSymbols();
    createMachine();
}
  
```

Объясню простым человеческим языком. Сначала я использую сгенерированный ANTLR синтаксический анализатор для разбора входного потока и создания абстрактного синтаксического дерева. Затем я обхожу дерево для построения таблицы символов (*Symbol Table (177)*). Наконец я собираю объекты в конечный автомат.

Первый шаг состоит в “уговаривании” ANTLR построить абстрактное синтаксическое дерево.

```

class StateMachineLoader...
private void loadAST() {
    try {
        StateMachineLexer lexer =
            new StateMachineLexer(new ANTLRReaderStream(input));
        StateMachineParser parser =
            new StateMachineParser(new CommonTokenStream(lexer));
        parser.helper = this;
        parser.setTreeAdaptor(new MyNodeAdaptor());
        ast = (MfTree) parser.machine().getTree();
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (RecognitionException e) {
        throw new RuntimeException(e);
    }
}
  
```

```
class MyNodeAdaptor extends CommonTreeAdaptor {
    public Object create(Token token) {
        return new MfTree(token);
    }
}
```

Вторым шагом в этом направлении является указание посредством `MyNodeAdaptor` строить абстрактное синтаксическое дерево с использованием `MfTree`, а не `CommonTree`.

Следующим шагом является создание таблицы символов. Это предполагает навигацию по абстрактному синтаксическому дереву, чтобы найти все события, команды и состояния и загрузить их в отображения для последующего быстрого поиска при построении конечного автомата.

```
class StateMachineLoader...
private void loadSymbols() {
    loadEvents();
    loadCommands();
    loadStateNames();
}
```

Бот код для событий.

```
class StateMachineLoader...
private Map<String, Event> events =
    new HashMap<String, Event>();

private void loadEvents() {
    MfTree eventList = ast.getSoleChild(EVENT_LIST);
    for (MfTree eventNode : eventList.getChildren()) {
        String name = eventNode.getText(0);
        String code = eventNode.getText(1);
        events.put(name, new Event(name, code));
    }
}

class MfTree...
List<MfTree> getChildren() {
    List<MfTree> result = new ArrayList<MfTree>();
    for (int i = 0; i < getChildCount(); i++)
        result.add((MfTree) getChild(i));
    return result;
}

MfTree getSoleChild(int nodeType) {
    List<MfTree> matchingChildren = getChildren(nodeType);
    assert 1 == matchingChildren.size();
    return matchingChildren.get(0);
}

List<MfTree> getChildren(int nodeType) {
    List<MfTree> result = new ArrayList<MfTree>();
    for (int i = 0; i < getChildCount(); i++)
        if (getChild(i).getType() == nodeType)
            result.add((MfTree) getChild(i));
    return result;
}

String getText(int i) {
    return getChild(i).getText();
}
```

Типы узлов определены в сгенерированном коде анализатора. Когда я использую их в загрузчике, чтобы упростить обращение, я могу применить статический импорт.

Команды загружаются аналогично. Я уверен, что вы сможете представить себе соответствующий код. Состояния также загружаются подобным образом, но в этот момент в объектах состояния у меня есть только имена.

```
class StateMachineLoader...
private void loadStateNames() {
    for (MfTree node : ast.getChildren(STATE))
        states.put(stateName(node),
                   new State(stateName(node)));
}
```

Я должен поступить подобным образом, потому что состояния используются в опежающих ссылках. В DSL я могу упомянуть состояние в переходе до того, как оно будет объявлено. Это именно тот случай, для которого очень хорошо подходит построение дерева — нет никакой проблемы в выполнении любого необходимого количества обходов абстрактного синтаксического дерева.

Последним шагом является фактическое создание конечного автомата.

```
class StateMachineLoader...
private void createMachine() {
    machine = new StateMachine(getStartState());
    for (MfTree node:
         ast.getChildren(StateMachineParser.STATE))
        loadState(node);
    loadResetEvents();
}
```

Начальное состояние представляет собой первое объявленное состояние.

```
class StateMachineLoader...
private State getStartState() {
    return states.get(getStartStateName());
}

private String getStartStateName() {
    return
        stateName((MfTree)ast.getFirstChildWithType(STATE));
}
```

Теперь можно связать переходы и действия для всех состояний.

```
class StateMachineLoader...
private void loadState(MfTree stateNode) {
    for (MfTree t : stateNode.getSoleChild(TRANSITION_LIST)
         .getChildren()) {
        getState(stateNode).addTransition(
            events.get(t.getText(0)),
            states.get(t.getText(1)));
    }
    for (MfTree t : stateNode.getSoleChild(ACTION_LIST)
         .getChildren())
        getState(stateNode).addAction(
            commands.get(t.getText()));
}

private State getState(MfTree stateNode) {
    return states.get(stateName(stateNode));
}
```

И наконец добавим сбрасывающие события, которых ожидает от нас API конечного автомата.

```
class StateMachineLoader...
private void loadResetEvents() {
    if (!ast.hasChild(RESET_EVENT_LIST)) return;
    MfTree resetEvents = ast.getsoleChild(RESET_EVENT_LIST);
    for (MfTree e : resetEvents.getChildren())
        machine.addResetEvents(events.get(e.getText()));
}

class MfTree...
boolean hasChild(int nodeType) {
    List<MfTree> matchingChildren = getChildren(nodeType);
    return matchingChildren.size() != 0;
}
```

24.4. Построение дерева с использованием кода действий (Java и ANTLR)

Использование синтаксиса ANTLR для построения дерева — самый простой способ создания последнего, но у многих генераторов синтаксических анализаторов (*Parser Generator (277)*) аналогичная функция отсутствует. В этих случаях вы все равно можете построить дерево, но самостоятельно с помощью кода действий. Ниже я приведу соответствующий пример. Я буду использовать в нем ANTLR, чтобы не вводить в книгу еще один генератор синтаксических анализаторов. Но следует подчеркнуть, что в реальности использовать этот способ в ANTLR нет смысла, так как проще получить тот же результат с помощью специального синтаксиса.

Первое, что нужно решить, — как представлять дерево. Я использую простой класс узла.

```
class Node...
private Token content;
private Enum type;
private List<Node> children = new ArrayList<Node>();

public Node(Enum type, Token content) {
    this.content = content;
    this.type = type;
}
public Node(Enum type) {
    this(type, null);
}
```

В данном случае мои узлы не являются статически типизированными — я использую один и тот же класс и для узлов состояний, и для узлов событий. Альтернативой является наличие различных видов узлов для различных классов.

У меня есть небольшой конструктор класса дерева, который представляет собой “обертку” для сгенерированного анализатора ANTLR для построения абстрактного синтаксического дерева.

```
class TreeConstructor...
private Reader input;

public TreeConstructor(Reader input) {
    this.input = input;
```

```

    }
    public Node run() {
        try {
            StateMachineLexer lexer =
                new StateMachineLexer(new ANTLRReaderStream(input));
            StateMachineParser parser =
                new StateMachineParser(new CommonTokenStream(lexer));
            parser.helper = this;
            return parser.machine();
        } catch (IOException e) {
            throw new RuntimeException(e);
        } catch (RecognitionException e) {
            throw new RuntimeException(e);
        }
    }
}

```

Мне требуется перечисление для объявления типов узлов. Я помещаю его в конструктор дерева и статически импортирую его для других классов, нуждающихся в нем.

```

class TreeConstructor...
public enum NodeType {STATE_MACHINE,
    EVENT_LIST, EVENT, RESET_EVENT_LIST,
    COMMAND_LIST, COMMAND,
    NAME, CODE,
    STATE, TRANSITION, TRIGGER, TARGET,
    ACTION_LIST, ACTION
}

```

Правила грамматики в синтаксическом анализаторе следуют одному и тому же базовому формату, который отлично иллюстрируется правилом для событий:

```

Файл грамматики...
event returns [Node result]
: {$result = new Node(EVENT);}
  name=ID {$result.add(NAME, $name);}
  code=ID {$result.add(CODE, $code);}
;

```

Каждое правило в качестве возвращаемого типа объявляет узел. Первая строка правила создает результирующий узел. При распознавании каждого токена, который является частью правила, я просто добавляю его в качестве дочернего узла.

Правила более высокого уровня повторяют этот шаблон.

```

Файл грамматики...
eventList returns [Node result]
: {$result = new Node(EVENT_LIST);}
  'events'
  // Добавление события:
  (e=event {$result.add($e.result);} )*
  'end'
;

```

Разница лишь в том, что я добавил узлы из подправил с помощью `$e.result`, так что ANTLR сможет корректно выбрать возвращаемый тип правила.

В строке за комментарием Добавление события имеется неочевидная идиома. Обратите внимание на то, как нужно разместить инструкцию события и код действия в скобках и применить оператор Клини к группе в скобках. Я делаю это, чтобы гарантировать, что код действия выполняется по одному разу для каждого события.

Я добавил к узлу методы для упрощения кода добавления дочерних узлов.

```
class Node...
    public void add(Node child) {
        children.add(child);
    }
    public void add(Enum nodeType, Token t) {
        add(new Node(nodeType, t));
    }
}
```

Как правило, я использую с файлом грамматики встроенный помощник (**Embedment Helper** (537)). Этот случай является исключением, так как код для построения абстрактного синтаксического дерева настолько прост, что вызовы методов помощника ничуть не облегчили бы работу с ним.

Правило верхнего уровня конечного автомата продолжает базовую структуру.

Файл грамматики...

```
machine returns [Node result]
    : {$result = new Node(STATE_MACHINE);}
      e=eventList {$result.add($e.result);}
      (r=resetEventList {$result.add($r.result);} )?
      (c=commandList {$result.add($c.result);} )?
      (s=state {$result.add($s.result);} )*
;
```

Команды и сбрасывающие события загружаются так же, как и обычные события.

Файл грамматики...

```
commandList returns [Node result]
    : {$result = new Node(COMMAND_LIST);}
      'commands'
      (c=command {$result.add($c.result);} )*
      'end'
;
command returns [Node result]
    : {$result = new Node(COMMAND);}
      name=ID {$result.add(NAME, $name);}
      code=ID {$result.add(CODE, $code);}
;
resetEventList returns [Node result]
    : {$result = new Node(RESET_EVENT_LIST);}
      'resetEvents'
      (e=ID {$result.add(NAME, $e);} )*
      'end'
;
```

Еще одним отличием является то, что я создал специальные типы узлов для имени и кода, которые позже сделают обход немного яснее.

Последний рассматриваемый код — синтаксический анализ состояний.

Файл грамматики...

```
state returns [Node result]
    : {$result = new Node(STATE);}
      'state' name = ID {$result.add(NAME, $name);}
      (a=actionList {$result.add($a.result);} )?
      (t=transition {$result.add($t.result);} )*
      'end'
;
transition returns [Node result]
    : {$result = new Node(TRANSITION);}
      trigger=ID {$result.add(TRIGGER, $trigger);}
      '>='
      target=ID {$result.add(TARGET, $target);}
;
```

```

actionList returns [Node result]
: {$result = new Node(ACTION_LIST);}
'actions' '{'
(action=ID {$result.add(ACTION, $action);}) *
}'
;

```

Код в файле грамматики совершенно обычный и довольно скучный. Это обычно означает, что вам нужна другая абстракция, — именно та, которую предоставляет специальный синтаксис построения дерева.

Вторая часть кода обходит дерево и создает конечный автомат. Это в значительной степени тот же код, что и примере выше, но с разницей, определяемой тем фактом, что здесь узлы немного отличаются от использовавшихся ранее.

```

class StateMachineLoader...
private Node ast;
private StateMachine machine;

public StateMachineLoader(Node ast) {
    this.ast = ast;
}
public StateMachine run() {
    loadSymbolTables();
    createMachine();
    return machine;
}

```

Я начинаю с загрузки таблицы символов.

```

class StateMachineLoader...
private void loadSymbolTables() {
loadStateNames();
loadCommands();
loadEvents();
}

private void loadEvents() {
for (Node n : ast.getDescendents(EVENT)) {
String name = n.getText(NAME);
String code = n.getText(CODE);
events.put(name, new Event(name, code));
}
}

class Node...
public List<Node> getDescendents(Enum requiredType) {
List<Node> result = new ArrayList<Node>();
collectDescendents(result, requiredType);
return result;
}
private void collectDescendents(List<Node> result, Enum requiredType) {
if (this.type == requiredType) result.add(this);
for (Node n : children) n.collectDescendents(result, requiredType);
}

```

Я показываю код загрузки событий; другие классы аналогичны.

```

class StateMachineLoader...
private void loadCommands() {
for (Node n : ast.getDescendents(COMMAND)) {
String name = n.getText(NAME);
String code = n.getText(CODE);

```

```

        commands.put(name, new Command(name, code));
    }
}
private void loadStateNames() {
    for (Node n : ast.getDescendents(STATE)) {
        String name = n.getText(NAME);
        states.put(name, new State(name));
    }
}

```

К классу узла я добавил метод получения текста для единственного узла типа. Это похоже на поиск в словаре, но используется та же самая структура данных дерева.

```

class Node...
public String getText(Enum nodeType) {
    return getSoleChild(nodeType).getText();
}
public String getText() {
    return content.getText();
}
public Node getSoleChild(Enum requiredType) {
    List<Node> children = getChildren(requiredType);
    assert children.size() == 1;
    return children.get(0);
}
public List<Node> getChildren(Enum requiredType) {
    List<Node> result = new ArrayList<Node>();
    for (Node n : children)
        if (n.getType() == requiredType) result.add(n);
    return result;
}

```

После организации символов я могу создать конечный автомат.

```

class StateMachineLoader...
private void loadState(Node stateNode) {
    loadActions(stateNode);
    loadTransitions(stateNode);
}
private void loadActions(Node stateNode) {
    for (Node action : stateNode.getDescendents(ACTION))
        states.get(stateNode.getText(NAME))
            .addAction(commands.get(action.getText())));
}
private void loadTransitions(Node stateNode) {
    for (Node transition :
        stateNode.getDescendents(TRANSITION)) {
        State source = states.get(stateNode.getText(NAME));
        Event trigger =
            events.get(transition.getText(TRIGGER));
        State target =
            states.get(transition.getText(TARGET));
        source.addTransition(trigger, target);
    }
}

```

Последний шаг — загрузка сбрасывающих событий.

```

class StateMachineLoader...
private void loadResetEvents() {
    if (!ast.hasChild(RESET_EVENT_LIST)) return;
    for (Node n : ast.getsoleDescendent(RESET_EVENT_LIST)

```

```
        .getChildren(NAME))
    machine.addResetEvents(events.get(n.getText())));
}

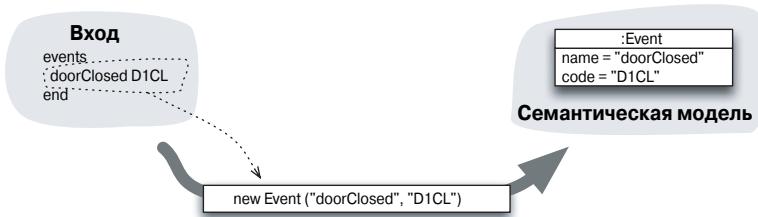
class Node...
public boolean hasChild(Enum nodeType) {
    return ! getChildren(nodeType).isEmpty();
}
```

Глава 25

Встроенная трансляция

Embedded Translation

Встраивает выходной код продукции в синтаксический анализатор, так что выход последнего создается постепенно по мере его работы



В синтаксически управляемой трансляции (Syntax-Directed Translation (229)) “чистый” синтаксический анализатор просто создает внутреннее синтаксическое дерево, так что для наполнения семантической модели (Semantic Model (171)) придется сделать нечто большее.

Встроенный перевод наполняет семантическую модель путем внедрения в синтаксический анализатор кода, который выполняет наполнение в соответствующих точках синтаксического анализа.

25.1. Как это работает

Синтаксические анализаторы предназначены для распознавания синтаксических структур. Используя встроенную трансляцию, мы помещаем код для наполнения семантической модели (Semantic Model (171)) в анализатор, так что по мере проведения синтаксического анализа постепенно выполняется и наполнение семантической модели. По большей части это означает, что код наполнения модели находится там, где распознается предложение входного языка, хотя на практике можно размещать части этого кода в различных точках.

При использовании встроенной трансляции с генераторами синтаксических анализаторов (Parser Generator (277)) для внедрения кода обычно применяется шаблон Foreign Code (315). Большинство генераторов синтаксических анализаторов предоставляют воз-

можность использовать внешний код; из использовавшихся мною генераторов только один не предназначался для работы с построением дерева (*Tree Construction* (289)).

Определенные проблемы при работе со встроенной трансляцией может вызвать то, что действия с побочными эффектами могут быть выполнены в самых неожиданных местах, в зависимости от того, как именно правила распознаются алгоритмом синтаксического анализа. Это не имеет значения в случае построения дерева, так как при этом просто создается поддерево возвращаемого значения. Так что, если вы обнаружили, что попали в запутанный клубок побочных эффектов, откажитесь от встроенной трансляции и займитесь построением дерева.

25.2. Когда это использовать

Наибольшая привлекательность встроенной трансляции заключается в том, что она обеспечивает простой способ за один проход выполнить как синтаксический анализ, так и наполнение модели. В случае построения дерева (*Tree Construction* (289)) вы должны предоставить код для построения и записи абстрактного синтаксического дерева и создать наполнитель модели, который выполняет обход этого дерева. Для простых случаев (которыми являются многие DSL), этот двухэтапный процесс может оказаться сложнее, чем следует.

На ваш выбор оказывают решающее влияние возможности генератора синтаксических анализаторов (*Parser Generator* (277)). Чем лучшие возможности по построению дерева имеет ваш генератор синтаксических анализаторов, тем более привлекательным становится применение соответствующей технологии.

Одной из самых больших проблем встроенной трансляции является то, что она может приводить к сложным файлам грамматики, обычно из-за плохого использования внешнего кода (*Foreign Code* (315)). Если вы дисциплинированы в его применении, то проблемы маловероятны; в этом отношении сильная сторона шаблона построения дерева в том, что он способствует выработке такой дисциплины.

Встроенная трансляция укладывается в один проход, так как вся работа выполняется одновременно с синтаксическим анализом. Это означает, что со всем, что вызывает сложности при одном проходе (например, опережающие ссылки), будут проблемы и при встроенной трансляции. Чтобы справиться с ними, часто приходится пользоваться переменной контекста (*Context Variable* (187)), что может еще больше усложнить синтаксический анализ.

Так что главный вывод можно сформулировать так: чем проще язык и синтаксический анализатор, тем привлекательнее применение встроенной трансляции.

25.3. Контроллер мисс Грант (Java и ANTLR)

Как всегда, я возьму тот же набивший оскомину пример, что и для построения дерева (*Tree Construction* (289)), с теми же инструментами (Java и ANTLR), но на этот раз будет использована встроенная трансляция. Прежде всего, все изменения относятся только к синтаксическому анализу — лексический анализ остается неизменным. Я не буду вновь рассматривать токенизацию (если хотите освежить память, обратитесь к разделу “Токенизация” на с. 293).

Еще одним сходством между этими двумя примерами является лежащая в основе грамматика БНФ. В основном при применении различных моделей синтаксического анализа БНФ-правила не меняются; изменения касаются кода поддержки, создаваемого

вокруг БНФ. Но в то время как при построении дерева применялись возможности ANTLR для объявления абстрактного синтаксического дерева, встроенная трансляция для размещения фрагментов Java, выполняющих непосредственное наполнение семантической модели (*Semantic Model* (171)), использует внешний код (*Foreign Code* (315)).

Встроенный перевод включает размещение произвольного кода общего назначения в файле грамматики. Как и в большинстве случаев, когда требуется вставка фрагментов на одном языке программирования в другой, я предпочитаю использовать шаблон *Embedment Helper* (537). Я работал с множеством файлов грамматик и считаю, что этот шаблон помогает поддерживать их ясность, не хороня их под горой внешнего кода. Я начинаю работу с объявления помощника в своей грамматике.

```
@members {
    StateMachineLoader helper;
//...
```

Верхний уровень грамматики определяет файл конечного автомата.

```
machine : eventList resetEventList commandList state*;
```

Как видите, это та же последовательность объявлений.

Чтобы увидеть реальную трансляцию, посмотрим на обработку списка событий.

```
eventList : 'events' event* 'end';
event      : name=ID code=ID {helper.addEvent($name, $code);};
```

Здесь мы видим типичное применение встроенной трансляции. Большая часть файла грамматики остается обычной БНФ, но в некоторых подходящих для этого точках в нее вносятся фрагменты кода общего назначения для выполнения трансляции. Так как я использую встраиваемый помощник, все, что я добавляю в грамматику, — это вызов одного метода помощника.

```
class StateMachineLoader...
void addEvent(Token name, Token code) {
    events.put(name.getText(),
               new Event(name.getText(), code.getText()));
}

private Map<String, Event> events =
    new HashMap<String, Event>();
private Map<String, Command> commands =
    new HashMap<String, Command>();
private Map<String, State> states =
    new HashMap<String, State>();
private List<Event> resetEvents =
    new ArrayList<Event>();
```

Этот вызов создает новый объект события и помещает его в таблицу символов, которая представляет собой коллекцию словарей в загрузчике. При вызове метода помощника ему передаются токены имени и кода. Чтобы встроенный код мог обращаться к элементам грамматики, ANTLR использует для их именования синтаксис присваивания. Размещение встроенного кода указывает, когда он выполняется. В данном случае встроенный код выполняется после того, как будут распознаны оба дочерних узла.

Команды обрабатываются точно так же. Однако при обработке состояний появляется пара интересных вопросов: иерархический контекст и опережающие ссылки.

Начнем с иерархического контекста. Проблема в том, что различные элементы состояния — действия и переходы — располагаются в его определении, так что, для того чтобы обработать действие, надо знать, в каком из состояний оно было объявлено.

Ранее я уже проводил аналогию между встроенной трансляцией и обработкой XML-текста с применением SAX. В определенной степени это корректная аналогия, так как встроенный код работает с правилами поочередно, по одному правилу за раз. Но эта аналогия несколько обманчива, поскольку генератор синтаксических анализаторов (*Parser Generator* (277)) может дать гораздо больше информации о контексте во время выполнения кода, поэтому хранить всю информацию о контексте не нужно.

В ANTLR вы можете передавать в правила информацию о контексте с помощью параметров.

```
state : 'state' name=ID {helper.addState($name);}
    actionList[$name]?
    transition[$name]*
    'end';

actionList [Token state]
    : 'actions' '{' actions+=ID* '}'
    {helper.addAction($state, $actions);}
;
```

Здесь токен состояния передается в правило для распознавания действия. Таким образом, код встроенной трансляции может получать как токен состояния, так и токены команд ("*" указывает, что они представляют собой список). Это обеспечивает корректный контекст для помощника.

```
class StateMachineLoader...
public void addAction(Token state, List actions) {
    for (Token action : (Iterable<Token>) actions)
        getState(state).addAction(getCommand(action));
}
private State getState(Token token) {
    return states.get(token.getText());
}
```

Второй вопрос состоит в том, что объявления переходов включают опережающие ссылки на еще необъявленные состояния. Во многих DSL можно переписать код так, чтобы ни один элемент не ссылался на идентификатор, который еще не объявлен. Однако наша модель этого не позволяет, и в результате мы сталкиваемся с проблемой опережающих ссылок. Шаблон построения дерева (*Tree Construction* (289)) позволяет обрабатывать абстрактное синтаксическое дерево в несколько проходов, так что при первом проходе можно собрать информацию из всех объявлений, а при втором заполнить состояния. В многопроходном режиме опережающие ссылки проблемой не являются, так как их можно разрешить при более позднем обходе дерева. Однако при встроенной трансляции такой возможности нет.

Наше решение заключается в использовании операции “получения” (термин, который я использую для операции “найти или создать”) как для ссылок, так и для объявлений. По существу это означает, что всякий раз, когда мы упоминаем состояние, мы неявно его объявляем (если оно еще не существует).

```
stateMachine.g...
transition [Token sourceState]
    : trigger = ID '=>' target = ID
    {helper.addTransition($sourceState,
        $trigger, $target);};

class StateMachineLoader...
public void addTransition(Token state, Token trigger,
```

```

        Token target) {
    getState(state).addTransition(getEvent(trigger),
                                  obtainState(target));
}
private State obtainState(Token token) {
    String name = token.getText();
    if (!states.containsKey(name))
        states.put(name, new State(name));
    return states.get(name);
}
}
```

Одним из следствий этого подхода является то, что опечатка при указании состояния в переходе приведет к пустому целевому состоянию. Если нас это устраивает, все можно оставить как есть. Но, как правило, следует выполнить проверку наличия объявлений, для чего нужно отследить все состояния, созданные не объявлением, а использованием, и убедиться, что для всех них имеются соответствующие объявления.

Наш язык определяет стартовое состояние, как первое состояние, упомянутое в программе. Этот вид контекста не слишком хорошо обрабатывается генератором синтаксических анализаторов, поэтому для решения данного вопроса требуется переменная контекста.

```

class StateMachineLoader...
public void addState(Token n) {
    obtainState(n);
    if (null == machine)
        machine = new StateMachine(getState(n));
}
```

Обработка сбрасывающих событий тривиальна — они просто вносятся в отдельный список.

```

stateMachine.g...
resetEventList : 'resetEvents' resetEvent* 'end' ;
resetEvent      : name=ID {helper.addResetEvent($name)} ;

class StateMachineLoader...
public void addResetEvent(Token name) {
    resetEvents.add(getEvent(name));
}
```

Однопроходная природа синтаксического анализатора привносит сложности и здесь. Сбрасывающие события могут быть определены прежде, чем мы получим первое состояние, и, таким образом, до того, как у нас появится конечный автомат, в который их следует поместить. Поэтому я храню их в поле для того, чтобы в конце добавить их к конечному автомату.

Метод `run` загрузчика указывает общую последовательность задач: лексический анализ, запуск генерированного синтаксического анализатора и завершение наполнения модели сбрасывающими событиями.

```

class StateMachineLoader...
public StateMachine run() {
    try {
        StateMachineLexer lexer =
            new StateMachineLexer(new ANTLRReaderStream(input));
        StateMachineParser parser =
            new StateMachineParser(new CommonTokenStream(lexer));
        parser.helper = this;
        parser.machine();
        machine.addResetEvents(
            resetEvents.toArray(new Event[0]));
    }
```

310 Часть III. Вопросы создания внешних DSL

```
    return machine;
} catch (IOException e) {
    throw new RuntimeException(e);
} catch (RecognitionException e) {
    throw new RuntimeException(e);
}
```

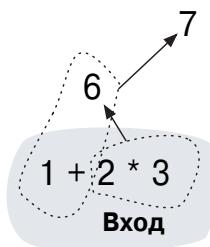
В коде, который следует за синтаксическим анализом, нет ничего необычного. Здесь же будет выполняться и весь семантический анализ.

Глава 26

Встроенная интерпретация

Embedded Interpretation

В грамматику встроены действия интерпретатора, так что синтаксический анализ интерпретирует текст, генерируя отклик



Есть много случаев, когда вам может понадобиться запустить сценарий DSL и получить немедленный результат, такой как выполнение вычислений или запроса. Данный шаблон интерпретирует DSL-сценарий во время синтаксического анализа, поэтому результатом последнего является результат выполнения самого сценария.

26.1. Как это работает

Встроенная интерпретация работает, вычисляя DSL-выражения как можно раньше, собирая полученные результаты выражений вместе и возвращая общий результат. Встроенная интерпретация не использует семантическую модель ([Semantic Model \(171\)](#)); вместо этого осуществляется непосредственная интерпретация входного DSL. По мере распознавания каждого фрагмента сценария DSL он интерпретируется в той степени, в какой это возможно.

26.2. Когда это использовать

Я большой сторонник семантической модели ([Semantic Model \(171\)](#)), так что я обычно выступаю против встроенной интерпретации. Она полезна только тогда, когда у вас есть относительно малые выражения, которые вы хотите просто вычислять и выполнять.

Иногда построение семантической модели не стоит связанных с этим усилий. Но так бывает крайне редко; даже в случае относительно небольшого DSL обычно проще действовать путем создания семантической модели и ее интерпретации вместо того, чтобы выполнять всю работу в анализаторе. Кроме того, семантическая модель обеспечивает прочный фундамент для развития языка.

26.3. Калькулятор (ANTLR и Java)

Пожалуй, наилучшим примером встраиваемой интерпретации является калькулятор. В этом случае легко интерпретировать каждое выражение и соединять результаты в единое целое. Это также тот случай, когда синтаксическое дерево для арифметики представляет собой отличную семантическую модель (*Semantic Model (171)*), так что попытка создать обычную семантическую модель (как я предпочитаю поступать) не дает никакого выигрыша.

Создание калькулятора в ANTLR имеет свои хитрости, так как арифметические выражения представляют собой вложенные операторные выражения (*Nested Operator Expression (333)*), а ANTLR — нисходящий синтаксический анализатор. Таким образом, грамматика становится немного сложнее.

Я начинаю с правила верхнего уровня. Так как арифметические выражения рекурсивны, ANTLR требуется правило верхнего уровня, чтобы знать, с чего начать синтаксический анализ.

```
grammar "Arith.g" .....
prog returns [double result]
: e=expression {$result = $e.result;};
```

Я вызываю это правило верхнего уровня из простого класса Java, представляющего собой оболочку для файла грамматики ANTLR.

```
class Calculator...
public static double evaluate(String expression) {
    try {
        Lexer lexer =
            new ArithLexer(new ANTLRReaderStream(
                new StringReader(expression)));
        ArithParser parser =
            new ArithParser(new CommonTokenStream(lexer));
        return parser.prog();
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (RecognitionException e) {
        throw new RuntimeException(e);
    }
}
```

Работу с вложенными операторными выражениями я должен начинать с операторов с наименьшим приоритетом, которыми в данном случае являются сложение и вычитание.

```
grammar "Arith.g" .....
expression returns [double result]
: a=mult_exp {$result = $a.result;}
  ( '+' b=mult_exp {$result += $b.result;}
  | '-' b=mult_exp {$result -= $b.result;}
  )*;
```

Здесь показан базовый шаблон калькулятора. Каждое правило грамматики распознает один оператор, а встроенный код Java выполняет соответствующее арифметическое действие. Оставшаяся часть грамматики следует описанному шаблону.

```
grammar "Arith.g" .....
power_exp returns [double result]
: a=unary_exp {$result = $a.result;}
| '*' b=power_exp
  {$result = Math.pow($result,$b.result);}
| '/' b=power_exp
  {$result = Math.pow($result, (1.0 / $b.result));}
) ?
;

unary_exp returns [double result]
: '-' a= unary_exp {$result = -$a.result;}
| a=factor_exp {$result = $a.result;}
;

factor_exp returns [double result]
: n=NUMBER {$result = Double.parseDouble($n.text);}
| a=par_exp {$result = $a.result;}
;

par_exp returns [double result]
: '(' a=expression ')' {$result = $a.result;}
;
```

Фактически этот калькулятор настолько прост и так хорошо подходит к структуре синтаксического дерева, что я даже не применяю встроенный помощник (`Embedment Helper` (537)).

Арифметические выражения — распространенный выбор для иллюстрации применения синтаксического анализатора. Во многих статьях и докладах в качестве примеров используется тот или иной вид калькулятора. Но я не думаю, что это представительный пример для работы с DSL. Большая проблема, возникающая при использовании арифметических выражений в качестве примеров, состоит в том, что они заставляют вас иметь дело с довольно редкой задачей (вложенные операторные выражения), и при этом избегают распространенных проблем DSL, которые приводят к применению семантической модели и встраиваемого помощника.

Глава 27

Внешний код

Foreign Code

*Встраивание некоторого внешнего кода во внешний DSL
для обеспечения более сложного поведения, чем то, которое
обеспечивается средствами DSL*

DSL

```
scott handles floor_wax in MA RI CT when {/^Baker/.test(lead.name)};
```

Javascript

По определению DSL является ограниченным языком, который способен лишь на немногое. Однако иногда в сценарии DSL необходимо описать нечто, что находится за пределами возможностей этого DSL. Одним из решений может быть расширение DSL соответствующей функциональной возможностью, но этот путь может привести к существенному усложнению DSL, так как лишит его почти всей простоты, которая и делает его столь привлекательным.

Внешний код встраивает фрагменты другого языка (часто — языка общего назначения) в определенные места в DSL.

27.1. Как это работает

Размещение фрагментов другого языка в DSL предусматривает решение двух вопросов. Во-первых, как следует распознавать эти фрагменты и встраивать их в грамматику и, во-вторых, как выполнять этот код, чтобы он мог сделать свою работу?

Внешний код появляется только в определенных частях DSL, так что грамматики DSL должны помечать места, где он может находиться. Одной из неприятностей в обработке внешнего кода является то, что грамматика не в состоянии распознать его внутреннюю структуру. В результате, как правило, с внешним кодом необходимо использовать альтернативную токенизацию (Alternative Tokenization (325)) и считывать его в анализатор в виде одной длинной строки. Затем можно либо вставлять эту строку в

семантическую модель (*Semantic Model* (171)) в том виде, в каком она была считана, либо передавать ее отдельному синтаксическому анализатору для внешнего кода, чтобы еще более тесно срастить внешний код с семантической моделью. Последний вариант более сложен и применим, только если внешний код представляет собой другой DSL. Часто внешний код является языком программирования общего назначения, и в этом случае, как правило, достаточно простой строки.

После того как внешний код оказывается в семантической модели, нужно решить, что с ним делать. Самая большая проблема заключается в том, должен ли внешний код быть интерпретирован или требуется его компиляция.

Интерпретация внешнего кода обычно наиболее проста, если имеется механизм для взаимодействия интерпретатора с базовым языком. Если базовый язык системы также интерпретируемый, то проще использовать в качестве внешнего кода его же. Если же базовый язык компилируемый, то для внешнего кода придется использовать интерпретируемый язык, который может быть вызван из базового языка, обеспечивая при этом передачу данных. В последнее время встречаются статические языковые среды, позволяющие взаимодействовать с интерпретируемыми языками. Обычно это немного неудобно, особенно когда речь идет о передаче данных. Кроме того, включение в проект еще одного языка иногда может приводить к проблемам.

В качестве альтернативы можно встраивать сам базовый язык, даже если он компилируемый. Здесь сложность в том, что это вносит в процесс сборки дополнительный шаг компиляции, как и при использовании генерации кода. Конечно, если вы действительно применяете генерацию кода, вы в любом случае должны сделать этот дополнительный шаг компиляции, так что добавление компилируемого внешнего кода не должно усложнить ситуацию. Сложности возникают, если вы используете компилируемый код и интерпретируемую семантическую модель.

Всякий раз, используя внешний код общего назначения, вы должны серьезно рассмотреть применение встроенного помощника (*Embedment Helper* (537)). Так можно свести внешний код в контексте DSL к минимуму. Одной из больших проблем в применении внешнего кода является то, что он может подавить DSL, существенно уменьшая такое важное достоинство DSL, как его удобочитаемость. Встраиваемый помощник прост в применении, и его стоит применять всегда, кроме разве что самых маленьких случаев.

Иногда внешнему коду требуется обращение к символам, определенным в самом сценарии DSL. Это происходит только тогда, когда сценарий DSL включает переменные или другие средства создания косвенных конструкций. Хотя они вездесущи в языках общего назначения, в DSL они на самом деле не так распространены, поскольку DSL часто в подобной выразительности не нуждаются. В результате на практике такое встречается редко, но, тем не менее, это хорошо известная ситуация, хотя бы потому, что она осуществляется в грамматиках — типичном месте применения внешнего кода. Вот пример такого использования.

```
allocationRule
: salesman=ID pc=productClause lc=locationClause
  ('when' predicate=ACTION)? SEP
  {helper.recognizedAllocationRule(salesman, pc, lc,
    predicate);}
;
```

Здесь использован внешний код на языке программирования Java. Код Java включает ссылки на salesman, pc, lc и predicate, и все они являются символами, определенными в грамматике. При обработке внешнего кода генератор синтаксических анализаторов (*Parser Generator* (277)) должен разрешать эти ссылки.

27.2. Когда это использовать

Когда вы размышляете о применении внешнего кода, обычная альтернатива заключается в расширении DSL до такой степени, чтобы он был способен делать то, для чего вы используете внешний код. Внешний код, безусловно, имеет свои недостатки. Применяя его, вы нарушаете абстракции, которые предоставляет DSL. Любой, кто читает такой сценарий DSL, должен понимать и DSL, и внешний код. Кроме того, использование внешнего кода усложняет процесс синтаксического анализа, а возможно, и саму семантическую модель (*Semantic Model* (171)).

При принятии решения о дальнейшем развитии DSL эти дополнительные сложности должны быть положены на одну чашу весов, а дополнительные сложности, которые должны быть добавлены в DSL для поддержки интересующей вас функциональной возможности, — на вторую. Чем мощнее DSL, тем сложнее он для понимания и использования.

Так когда же следует прибегать к применению внешнего кода? Естественно, в первую очередь, тогда, когда вам действительно нужен язык программирования общего назначения. Поскольку вряд ли вы хотите превращать свой DSL в язык общего назначения, вы просто вынуждены подумать об использовании внешнего кода.

Другой случай применения внешнего кода — когда в сценариях DSL вам нужна некоторая очень редко используемая функциональная возможность. Редко применяемая возможность вряд ли стоит расширения DSL.

Еще одним фактором при принятии решения является целевая аудитория — кто именно будет использовать DSL. Если DSL используется только программистами, то добавление внешнего кода не вызовет проблем — профессионалы разберутся и во внешнем коде, и в DSL. Если же читать DSL будут не программисты, то это аргумент против применения внешнего кода, который они, скорее всего, не поймут и с которым не смогут работать. Впрочем, при крайне редком применении внешнего кода это также не является непреодолимой проблемой.

27.3. Встраивание динамического кода (ANTLR, Java и Javascript)

Для того чтобы что-то продавать, необходимы продавцы. Если у вас есть много продавцов, каким-то образом нужно решить, как распределить между ними покупателей. Общим понятием при этом являются “территории”, которые определяются набором правил распределения покупателей между продавцами. Это деление на территории может быть основано на различных факторах. Вот сценарий распределения, в котором использованы штаты США и товары.

```
scott handles floor_wax in WA;
helen handles floor_wax desert_topping in AZ NM;
brian handles desert_topping in WA OR ID MT;
otherwise scott
```

Это простой DSL, в котором поочередно проверяется каждое правило. И как только сделка будет соответствовать правилу, покупателю будет назначен свой продавец.

Теперь давайте представим, что Скотт подружился с исполнительным директором компании Baker Industries, работающей на юге Новой Англии. Так как они проводят много времени на поле для гольфа вместе, мы хотим, чтобы всю мастику для пола компании Baker Industries продавал только Скотт. Дело осложняется тем, что Baker Industries имеет множество филиалов с разными названиями: Baker Industrial Holdings, Baker Floor

Toppings и т.д. Поэтому принято решение, что все дела с компаниями, названия которых начинаются на “Baker”, будет вести Скотт.

Чтобы сделать это, можно было бы расширить наш DSL, но так как это один из тех частных случаев, которые в конечном итоге усложняют язык, лучше прибегнуть к внешнему коду. Вот как выглядит то, что мы только что описали обычными словами.

```
scott handles floor_wax in MA RI CT
when {/^Baker/.test(lead.name)};
```

Для внешнего кода я воспользовался Javascript, который я выбрал, потому что он легко интегрируется с Java и может вычисляться во время выполнения, что позволяет избежать перекомпиляции при изменении правил распределения. Код Javascript трудно назвать удобочитаемым — думаю, что для того, чтобы менеджер по продажам поверил, что все будет работать так, как надо, мне пришлось бы сказать ему что-то вроде “зуб даю” или “мамой клянусь”, но все действительно будет работать. Я не использую здесь встроенный помощник (Embedment Helper (537)), так как предикат очень мал.

27.3.1. Семантическая модель

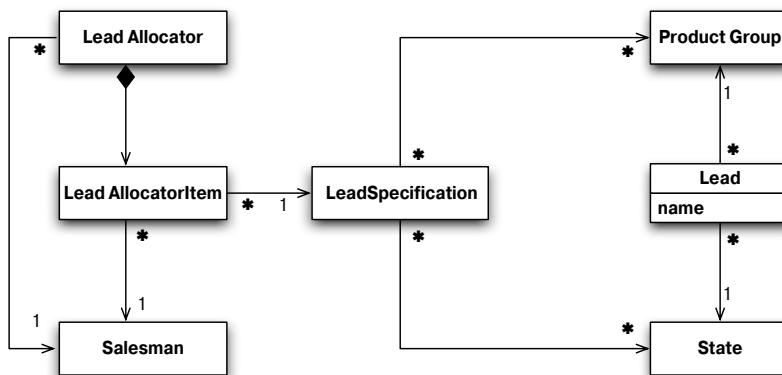


Рис. 27.1. Модель распределения покупателей

В нашей простой модели (рис. 27.1) есть покупатели, и каждый из них имеет группу товаров и состояние.

```

class Lead...
    private String name;
    private State state;
    private ProductGroup product;

    public Lead(String name, State state,
               ProductGroup product) {
        this.name = name;
        this.state = state;
        this.product = product;
    }

    public State getState() {return state;}
    public ProductGroup getProduct() {return product;}
    public String getName() {return name;}
  
```

Для распределения покупателей между продавцами имеется распределитель, который содержит список элементов, связывающих продавцов со спецификациями покупателей.

```
class LeadAllocator...
private List<LeadAllocatorItem> allocationList =
    new ArrayList<LeadAllocatorItem>();
private Salesman defaultSalesman;

public void appendAllocation(Salesman salesman,
                             LeadSpecification spec) {
    allocationList.add(new LeadAllocatorItem(salesman, spec));
}

public void setDefaultSalesman(Salesman defaultSalesman) {
    this.defaultSalesman = defaultSalesman;
}

private class LeadAllocatorItem {
    Salesman salesman;
    LeadSpecification spec;

    private LeadAllocatorItem(Salesman salesman,
                             LeadSpecification spec) {
        this.salesman = salesman;
        this.spec = spec;
    }
}
```

Спецификация покупателя следует шаблону *Specification* [7] и устанавливает соответствие, если атрибуты покупателя включены в список спецификаций.

```
class LeadSpecification...
private List<State> states = new ArrayList<State>();
private List<ProductGroup> products =
    new ArrayList<ProductGroup>();
private String predicate;

public void addStates(State... args) {
    states.addAll(Arrays.asList(args));
}
public void addProducts(ProductGroup... args) {
    products.addAll(Arrays.asList(args));
}
public void setPredicate(String code) {predicate = code;}

public boolean isSatisfiedBy(Lead candidate) {
    return statesMatch(candidate) &&
           productsMatch(candidate) &&
           predicateMatches(candidate)
    ;
}
private boolean productsMatch(Lead candidate) {
    return products.isEmpty() ||
           products.contains(candidate.getProduct());
}
private boolean statesMatch(Lead candidate) {
    return states.isEmpty() ||
           states.contains(candidate.getState());
}
private boolean predicateMatches(Lead candidate) {
    if (null == predicate) return true;
```

```

        return evaluatePredicate(candidate);
    }
}

```

Спецификация также содержит предикат, который представляет собой некоторый встроенный код Javascript. Он вычисляется спецификацией с применением Java-процессора Rhino Javascript.

```

class LeadSpecification...
boolean evaluatePredicate(Lead candidate) {
    try {
        ScriptContext newContext = new SimpleScriptContext();
        Bindings engineScope =
            newContext.getBindings(ScriptContext.ENGINE_SCOPE);
        engineScope.put("lead", candidate);
        return (Boolean) javascriptEngine()
            .eval(predicate, engineScope);
    } catch (ScriptException e) {
        throw new RuntimeException(e);
    }
}
private ScriptEngine javascriptEngine() {
    ScriptEngineManager factory = new ScriptEngineManager();
    ScriptEngine result =
        factory.getEngineByName("JavaScript");
    assert result != null : "Не найден процессор javascript";
    return result;
}
}

```

Я добавляю покупателя, для которого выполняется вычисление, в область видимости Javascript, так что встроенный код Javascript может получить доступ к свойствам покупателя.

Распределитель работает путем перебора списка элементов, возвращая первого продавца с соответствующей спецификацией.

```

class LeadAllocator...
public Salesman determineSalesman(Lead lead) {
    for (LeadAllocatorItem i : allocationList)
        if (i.spec.isSatisfiedBy(lead)) return i.salesman;
    return defaultSalesman;
}
}

```

27.3.2. Синтаксический анализатор

Основной управляющий класс трансляции в результате своей работы создает распределитель покупателей.

```

class AllocationTranslator...
private Reader input;
private AllocationLexer lexer;
private AllocationParser parser;
private ParsingNotification notification =
    new ParsingNotification();
private LeadAllocator result = new LeadAllocator();

public AllocationTranslator(Reader input) {
    this.input = input;
}

public void run() {
    try {
        lexer =

```

```

        new AllocationLexer(new ANTLRReaderStream(input));
parser =
    new AllocationParser(new CommonTokenStream(lexer));
parser.helper = this;
parser.allocationList();
} catch (Exception e) {
    throw new RuntimeException(
        "Исключение в процессе синтаксического анализа", e);
}
if (notification.hasErrors())
    throw new
        RuntimeException("Ошибка синтаксического анализа: \n"
            + notification);
}
}

```

Транслятор распределения для файла грамматики выступает в роли встроенного помощника (Embedment Helper (537)).

```

Грамматика...
@members {
    AllocationTranslator helper;

    public void reportError(RecognitionException e) {
        helper.addError(e);
        super.reportError(e);
    }
}

```

Я прохожу по файлу грамматики сверху вниз и использую встроенную трансляцию (Embedded Translation (305)).

Вот список использованных мною базовых токенов.

```

Грамматика...
ID   : ('a'...'z' | 'A'...'Z' | '0'...'9' | ' ' )+;
WS   : (' ' | '\t' | '\r' | '\n')+ {skip();};
SEP : ';' ;

```

Здесь определены обычные токены пробельных символов и идентификаторов с явным указанием точки с запятой в качестве разделителей инструкций.

Вот правило грамматики верхнего уровня.

```

Грамматика...
allocationList
: allocationRule* 'otherwise' ID
    {helper.recognizedDefault($ID);}
;

class AllocationTranslator...
void recognizedDefault(Token token) {
    if (!Registry.salesmenRepository()
        .containsId(token.getText()))
    {
        notification.error(token, "Неизвестный продавец: %s",
            token.getText());
        return;
    }
    Salesman salesman =
        Registry.salesmenRepository().findById(token.getText());
    result.setDefaultSalesman(salesman);
}

```

Предполагается, что все продавцы, товары и состояния имеются до интерпретации правил распределения, возможно, в некоторой базе данных. В данном примере я буду обращаться к этим данным с использованием репозиториев (*Repositories* [10]).

Вы можете сказать, что эта грамматика также демонстрирует применение внешнего кода, отличным примером которого является код действий. Но в ANTLR этот внешний код встраивается в сгенерированный синтаксический анализатор в процессе генерации его кода, что кардинально отличается от того, что я делаю с правилом распределения на Javascript. Однако здесь также работают базовые принципы применения внешнего кода; в частности, я использую встраиваемый помощник для сведения количества внешнего кода к минимуму.

Теперь пришло время вернуться к основной сюжетной линии и показать вам правила распределения.

Грамматика...

```
allocationRule
: salesman=ID pc=productClause lc=locationClause
  ('when' predicate=ACTION)? SEP
  {helper.recognizedAllocationRule(salesman, pc,
    lc, predicate);}
;
```

Правило довольно простое. Оно включает продавца, предложения (подправила) для товара и местоположения, а также необязательный токен предиката и разделитель. Тот факт, что предикат представляет собой токен, а не под правило, важен, потому что мы хотим собрать весь код Javascript в виде одной строки и не пытаться его разбирать.

Здесь имеется единственный вызов помощника для записи распознавания, который работает после того, как будут проанализированы предложения для продукта и местоположения. Я следовал соглашению, согласно которому для переменных токенов используются полные имена (в данном случае это продавец и предикат), поскольку сами токены имеют недостаточно понятные имена. Для подправил использованы аббревиатуры, так как их названия самоочевидны и применение полных имен для переменных будет простым дублированием имен подправил (и, таким образом, добавлением шума в текст правила).

Давайте посмотрим на подправила, в частности на под правило для товара.

Грамматика...

```
productClause returns [List<ProductGroup> result]
: 'handles' p+=ID+
  {$result = helper.recognizedProducts($p);}
;
```

Я сделал это под правило возвращающим список групп товаров. В результате само оно не наполняет семантическую модель (*Semantic Model* (171)), но возвращает объекты для такого наполнения в родительское правило. Я поступаю так потому, что иначе мне потребуется доступ к текущему правилу распределения внутри действия правила для товара. Это обычно требует применения переменной контекста (*Context Variable* (187)), чего я хотел бы избежать. ANTLR имеет возможность передачи объектов в качестве аргументов правила, так что я мог бы поступить описанным способом, но я предпочитаю работать с семантической моделью в одном месте.

Мне все еще нужно действие для преобразования токенов товаров в реальные объекты. Это действие представляет собой простой поиск в хранилище.

```
class AllocationTranslator...
List<ProductGroup>
recognizedProducts(List<Token> tokens) {
  List<ProductGroup> result =

```

```

        new ArrayList<ProductGroup>();
    for (Token t : tokens) {
        if (!Registry.productRepository()
            .containsId(t.getText())) {
            notification.error(t, "Товар для %s не найден",
                t.getText());
            continue;
        }
        result.add(Registry.productRepository()
            .findById(t.getText()));
    }
    return result;
}

```

Подправило для местоположения работает почти так же, поэтому я перейду к главному в данном примере — к получению кода Javascript. Как я говорил выше, я делаю это в лексическом анализаторе. Меня не заботит его содержание, поэтому я просто хочу передать всю строку в спецификацию покупателя. Нет смысла в создании или использовании синтаксического анализатора Javascript, если только я не хочу проверять во время синтаксического анализа корректность кода Javascript. Поскольку подобная проверка все равно выявила бы только синтаксические, но не семантические ошибки, я не думаю, что создание такого анализатора стоит затрачиваемых усилий.

Для получения интересующего меня текста я использую альтернативную токенизацию (Alternative Tokenization (325)). Простейший способ состоит в выборе пары разделителей, которые не используются ни для чего другого, и в применении правила для токена следующего вида.

```
ACTION : '{' .* '}' ;
```

Это достаточно разумное правило, работающее во многих ситуациях. Однако у него есть и потенциальная проблема — применение фигурных скобок в самом коде Javascript приведет к его неработоспособности. Решить ее можно путем применения необычных разделителей, например пар символов.

```
ACTION : ':' .* ':' ;
```

Однако в ANTLR я могу воспользоваться его собственными возможностями по обработке вложенных токенов.

Грамматика...

```

ACTION
: NESTED_ACTION;

fragment NESTED_ACTION
: '{' (ACTION_CHAR | NESTED_ACTION)* '}'
;

fragment ACTION_CHAR
: ~('{'|'}')
;
```

Это решение не идеальное, не способное справиться, например, с фрагментом Javascript `badThing = "{}";`, но в большинстве случаев оно вполне работоспособно.

Имея на руках всю коллекцию подправил и предикат Javascript, я могу обновить семантическую модель.

```

class AllocationTranslator...
void recognizedAllocationRule(Token salesmanName,
    List<ProductGroup> products,
```

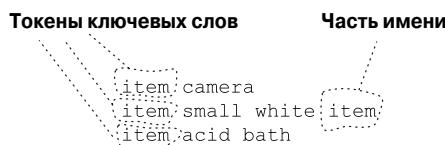
```
        List<State> states,
        Token predicate)
{
    if (!Registry.salesmenRepository()
        .containsId(salesmanName.getText())) {
        notification.error(salesmanName,
                            "Неизвестный продавец: %s",
                            salesmanName.getText());
        return;
    }
    Salesman salesman =
        Registry.salesmenRepository()
        .findById(salesmanName.getText());
    LeadSpecification spec = new LeadSpecification();
    spec.addStates(
        (State[])states.toArray(new State[states.size()]));
    spec.addProducts(
        (ProductGroup[])products.toArray(
            new ProductGroup[products.size()]));
    if (null != predicate)
        spec.setPredicate(predicate.getText());
    result.appendAllocation(salesman, spec);
}
```

Глава 28

Альтернативная токенизация

Alternative Tokenization

*Изменение поведения лексического анализатора
из синтаксического анализатора*



28.1. Как это работает

В моем простом обзоре работы генераторов синтаксических анализаторов (*Parser Generator* (277)) я говорил, что лексический анализатор поставляет поток токенов синтаксическому анализатору, который собирает его в синтаксическое дерево. Подразумевается, что это одностороннее взаимодействие: лексический анализатор является источником данных, которые потребляет синтаксический анализатор. Но оказывается, что это не всегда так. Есть моменты, когда способ токенизации (разбиения исходного текста на токены) должен изменяться в зависимости от местоположения в синтаксическом дереве. Это означает, что синтаксический анализатор должен иметь возможность управлять тем, как лексический анализатор выполняет токенизацию.

В качестве простого примера этой проблемы рассмотрим перечисление товаров, которые могут находиться в каталоге.

```
item camera;  
item small_power_plant;  
item acid_bath;
```

Применение подчеркиваний или соединения слов в одно с помощью прописных букв (*smallPowerPlant*) — знакомый и обычный способ записи для нас, но не для обычных пользователей, привыкших к пробелам. Они предпочли бы другой исходный текст.

```
item camera;
item small power plant;
item acid bath;
```

Насколько трудно этого достичь? Как оказалось, при использовании синтаксических анализаторов, управляемых грамматиками, это может быть на удивление сложно, поэтому я даже выделил для этой темы отдельный раздел. (Кстати, здесь я использую для разделения элементов точку с запятой. Можно использовать и другие символы, например символ новой строки, и он даже может оказаться предпочтительнее. Я мог бы поступить так же, но при этом возникает еще один сложный вопрос обработки разделителей строк (*Newline Separators* (339)), так что сейчас я буду использовать точку с запятой, на время отложив все сложности.)

Простейшая грамматика, позволяющая распознать любое количество слов после ключевого слова `item`, может выглядеть следующим образом.

```
catalog : item*;
item    : 'item' ID* ';' ;
```

Беда в том, что все это перестает работать, если имеется элемент, в названии которого содержится слово “`item`”, например `item small white item`.

Проблема в том, что лексический анализатор распознает `item` в качестве ключевого, а не обычного слова и, таким образом, возвращает токен ключевого слова, а не токен идентификатора. В действительности же необходимо рассматривать все находящееся между ключевым словом `item` и точкой с запятой, как идентификатор, изменения правила токенизации для этой точки синтаксического анализа.

Другой распространенный пример ситуации такого рода — внешний код (*Foreign Code* (315)), который может включать все виды значимых токенов DSL. Но нам нужно, чтобы все они были проигнорированы, а внешний код был воспринят как одна большая строка для встраивания в семантическую модель (*Semantic Model* (171)).

Имеется несколько подходов к решению поставленных задач, но не все они работают со всеми видами генераторов синтаксических анализаторов.

28.1.1. Кавычки

Простейший способ решения этой проблемы — заключение текста в кавычки для того, чтобы лексический анализатор распознавал его как нечто особенное. Что касается нашего примера, то это означает использование некоторого вида кавычек, как минимум при использовании слова `item`. Вот как это будет выглядеть.

```
item camera
item small power plant;
item "small white item";
```

Такой исходный текст распознается грамматикой наподобие следующей.

```
catalog      : item*;
item        : 'item' item_name ';' ;
item_name   : (ID | QUOTED_STRING)* ;
QUOTED_STRING : '"' (options{greedy = false;} : .)* '"';
```

Кавычки собирают в одно целое весь текст, заключенный между ними, так что на него не распространяются никакие другие правила лексического анализатора. Затем я могу делать с полученным текстом все что захочу.

Кавычки никак не связаны с синтаксическим анализатором, так что эта схема может использоваться в любом месте языка. У вас не может быть особых правил для заключения отдельных элементов языка в кавычки. Однако во многих случаях этот способ не-плохо работает.

Неприятный аспект применения кавычек — сложности при их использовании внутри строки наподобие Active "Marauders" Map, где необходимы кавычки внутри строки в кавычках. Есть определенные способы борьбы с этим явлением, которые должны быть знакомы вам по обычному программированию.

Первый заключается в использовании управляющих символов наподобие любой обратной косой черты в Unix или удвоенного символа разделителя. Для обработки элемента Active "Marauders" Map можно использовать следующее правило.

```
QUOTED_STRING : STRING_DELIM
    (STRING_ESCAPE | ~(STRING_DELIM) *) *
    STRING_DELIM;
fragment STRING_ESCAPE: STRING_DELIM STRING_DELIM;
fragment STRING_DELIM : ''';
```

Основной трюк заключается в использовании дополнительного разделителя в пределах текста для того, чтобы указать, что такой разделитель является не разделителем, а содержащим текста. То же самое можно записать и в более компактной форме.

```
QUOTED_STRING : '''' ('''' | ~( ''')) * '''';
```

Я предпочитаю многословную ясность, но такая ясность — очень большая редкость, когда речь идет о регулярных выражениях.

Описанный метод работает хорошо, но может привести к путанице, особенно среди не программистов.

Другой способ заключается в выборе необычного сочетания символов разделителя, которые вряд ли появятся в тексте. Хорошим примером этого является генератор синтаксических анализаторов (*Parser Generator* (277)) Java CUP. Большинство генераторов синтаксических анализаторов используют для указания кода действий фигурные скобки, но при этом сталкиваются с проблемой, связанной с тем, что фигурные скобки очень распространены, особенно в С-образных языках. Поэтому CUP использует " { : " и " : } " в качестве разделителей (комбинация, которую не найти в большинстве языков программирования, включая Java).

Такие разделители, очевидно, применимы в той же степени, в какой обеспечивается невозможность встретить их в тексте. Во многих DSL этот метод вполне работоспособен, так как в тексте сценариев DSL обычно может встречаться только ограниченное количество текстовых элементов.

Третья тактика заключается в использовании более чем одного вида разделителей, так что, если вам необходимо вставить в строку символ-разделитель, это можно сделать, переходя к альтернативным разделителям в качестве кавычек. Например, многие языки сценариев позволяют использовать как одинарные, так и двойные кавычки. Такой подход имеет дополнительное преимущество — сокращение путаницы, вызываемой в некоторых языках, допускающих применение только одного типа. Разрешение двойных или одинарных кавычек для нашего примера имеет достаточно простой вид.

```
catalog      : item*;
item        : 'item' item_name ',';
item_name   : (ID | QUOTED_STRING)* ;
QUOTED_STRING : DOUBLE_QUOTED_STRING |
    SINGLE_QUOTED_STRING ;
```

```
fragment DOUBLE_QUOTED_STRING : '"';
    (options{greedy = false;} : .)* '"';
fragment SINGLE_QUOTED_STRING : '\'';
    (options{greedy = false;} : .)* '\'';
```

Имеется и менее распространенный вариант, который иногда может быть полезным. Некоторые генераторы синтаксических анализаторов, в том числе ANTLR, используют для лексического анализа стековые машины, а не конечные автоматы. Это дает еще один вариант для тех случаев, когда символы разделителей представляют собой совпадающие пары (как "{...}"). Для этого требуется небольшое изменение моего примера. Представьте себе, что я хочу вставлять Javascript-фрагменты в список элементов, чтобы обеспечить указание условий для вставки элемента в каталог, как, например, показано ниже.

```
item lyncanthropic gerbil {!isFullMoon()};
```

Проблема здесь в том, что код Javascript может включать фигурные скобки. Однако мы можем разрешить их, только если имеются пары соответствующих скобок, записав правило наподобие следующего.

```
catalog : item*;
item   : 'item' item_name CONDITION? ';' ;

CONDITION           : NESTED_CONDITION;
fragment NESTED_CONDITION : '{' (CONDITION_CHAR |
                                NESTED_CONDITION)* '}';
fragment CONDITION_CHAR  : ~('{'|'}') ;
```

Так любые конфигурации фигурных скобок обработать не удастся: код `{System.out.print("Разбери: {}");}` приведет к некорректной работе анализатора. Для полной победы я должен был бы написать дополнительные правила лексического анализа, но для подавляющего большинства ситуаций указанных правил вполне достаточно. Самым большим недостатком этого метода является то, что он работает, только если лексический анализатор представляет собой стековую машину, а такие лексические анализаторы встречаются относительно редко.

28.1.2. Лексическое состояние

Возможно, самый логичный способ решения этой проблемы, по крайней мере в случае названия элемента, заключается в полной замене лексического анализатора на время получения имени. То есть, встретив ключевое слово `item`, мы заменим наш обычный лексический анализатор другим, пока не встретим точку с запятой, после чего вновь возвращаемся к прежнему лексическому анализатору.

Лексический анализатор flex, GNU-версия lex, поддерживает подобное поведение под названием **стартовых условий** (start conditions, называемые также **лексическим состоянием** (lexical state)). Хотя эта возможность использует один и тот же лексический анализатор, она позволяет грамматике переключить лексический анализатор в другой режим. Это почти то же самое, что и изменение анализатора, и, конечно, этого вполне достаточно для данного примера.

В этом примере я перейду к коду Java CUP, поскольку ANTLR не поддерживает изменение лексического состояния (в настоящее время это невозможно, так как лексический анализатор выполняет токенизацию всего входного потока до того, как синтаксический анализатор приступит к его обработке). Вот CUP- грамматика для обработки наших элементов.

```
<YYINITIAL> "item" {return symbol(K_ITEM); }
<YYINITIAL> {Word} {return symbol(WORD); }
```

```
<gettingName> {Word} {return symbol(WORD);}

";"      {return symbol(SEMI);}
{WS}     { /* Игнорируем */ }
{Comment} { /* Игнорируем */ }
```

Здесь используются два лексических состояния: YYINITIAL и gettingName. Состояние YYINITIAL является состоянием по умолчанию, из которого лексический анализатор начинает свою работу. Я могу использовать эти лексические состояния для аннотации правил лексического анализатора. В данном случае вы видите, что ключевое слово item распознается в качестве ключевого слова только в состоянии YYINITIAL. Правила лексического анализатора без состояния (такие, как ";") применяются во всех состояниях. (Строго говоря, мне не нужны два состояния — правила для {Word} остаются одинаковыми, но я показал их здесь, чтобы проиллюстрировать синтаксис.)

Затем я переключаюсь между лексическими состояниями в грамматике. Правила подобны правилам ANTLR, но немного отличаются, так как CUP использует другую версию БНФ. Я сначала покажу пару правил, которые не включаются в изменение лексического состояния. Первое правило — это правило верхнего уровня для каталога, которое представляет собой базовую БНФ-форму правила ANTLR.

```
catalog ::= item | catalog item ;
```

С другой стороны, имеется правило для сборки имени элемента.

```
item_name ::= WORD:w { : RESULT = w; :}
| item_name:n WORD:w { : RESULT = n + " " + w; :}
;
```

Правило для распознавания элемента включает лексическое переключение.

```
item ::= K_ITEM
{ : parser.helper.startingItemName(); :}
item_name:n
{ : parser.helper.recognizedItem(n); :}
SEMI
;

class ParsingHelper...
void recognizedItem(String name) {
    items.add(name);
    setLexicalState(Lexer.YYINITIAL);
}
public void startingItemName() {
    setLexicalState(Lexer.gettingName());
}
private void setLexicalState(int newState) {
    getLexer().yybegin(newState);
}
```

Базовый механизм очень прост. Как только анализатор распознает ключевое слово item, он переключает лексическое состояние, чтобы просто получить поступающие слова. Как только слова заканчиваются, анализатор переключается обратно.

Если судить по написанному, все довольно просто, но есть одна загвоздка. Для того чтобы разрешить свои правила, анализаторам требуется предпросмотр потока токенов. ANTLR использует предпросмотр произвольной глубины — отчасти поэтому перед тем, как синтаксиче-

ский анализатор приступает к работе, выполняется токенизация всего исходного текста. CUP, как и Yacc, выполняет предпросмотр одного токена. Но этого токена достаточно, чтобы вызвать проблемы в случае объявления вида `item item the troublesome`. Проблема в том, что первое слово в имени элемента обрабатывается прежде, чем происходит изменение лексического состояния, поэтому в данном случае оно будет анализироваться как ключевое слово `item`, тем самым нарушая работу анализатора.

Легко столкнуться и с более серьезной проблемой. Вы могли заметить, что я разместил вызов для сброса лексического состояния (`recognizedItem`) перед распознаванием разделителя инструкций. Если бы я поместил его после него, то ключевое слово `item` распознавалось бы при предпросмотре, до переключения анализатора в исходное состояние.

Это еще одна неприятность, которая заставляет быть предельно осторожными с лексическими состояниями. Если вы используете общую границу токенов (например, кавычки), то можете избежать проблем при предпросмотре на один токен вперед. В противном случае вы должны учитывать, как предпросмотр синтаксического анализатора взаимодействует с лексическими состояниями лексического анализатора. В результате комбинирование синтаксического анализа и лексических состояний может оказаться очень сложным делом.

28.1.3. Изменение типа токена

Правила синтаксического анализатора реагируют не на полное содержание токена, а на его тип. Если мы можем изменить тип токена, прежде чем он достигнет синтаксического анализатора, мы можем преобразовать ключевое слово `item` в обычное слово `item`.

Этот подход противоположен подходу лексических состояний. В случае лексических состояний лексический анализатор должен по одному передавать токены синтаксическому анализатору; при этом подходе вы можете свободно предпросматривать поток токенов. Не удивительно, что этот подход в большей степени пригоден для ANTLR, чем для Yacc, так что я опять возвращаюсь к любимому генератору синтаксических анализаторов.

```
catalog : item*;
item   :
  'item' {helper.adjustItemNameTokens(); }
  ID*
  SEP
  ;
SEP    : ';' ;
```

В грамматике нет ничего, что показывало бы, что же происходит, — все действия выполняются в помощнике.

```
void adjustItemNameTokens() {
  for (int i = 1;
       !isEndOfItemName(parser.getTokenStream().LA(i));
       i++) {
    assert i<100:"Такое количество токенов означает ошибку";
    parser.getTokenStream().LT(i).setType(parser.ID);
  }
}
private boolean isEndOfItemName(int arg) {
  return (arg == parser.SEP);
}
```

Код движется по потоку токенов, превращая типы токенов в `ID`, пока не достигнет разделителя. (Я объявил здесь тип токена для разделителя с тем, чтобы сделать его доступным из помощника.)

Этот метод не отражает точное содержание исходного текста, так как до синтаксического анализатора не доберется то, что было отброшено лексическим анализатором (например, пробелы при использовании такого подхода не сохраняются). Если это важно, то данный метод вам не подходит.

Чтобы увидеть применение этого метода в большем контексте, взглянем на синтаксический анализатор языка запросов Hibernate — HQL. HQL работает со словом `order`, которое появляется либо в качестве ключевого слова (в `"order by"`) или в качестве имени столбца или таблицы. Лексический анализатор по умолчанию возвращает `order` в качестве ключевого слова, но действие синтаксического анализатора выполняет предпросмотр, чтобы увидеть, не следует ли за ним слово `by`, и если не следует, то преобразует его в идентификатор.

28.1.4. Игнорирование типов токенов

Если сами токены не имеют значения и вы хотите получить полный текст, можете полностью игнорировать типы токенов и собирать все токены до тех пор, пока не будет достигнут токен-ограничитель (в данном случае — разделитель).

```
catalog    : item*;
item      : 'item' item_name SEP;
item_name : ~SEP* ;
SEP       : ';' ;
```

Основная идея заключается в том, чтобы записать правило для имени элемента каталога так, чтобы оно принимало любые токены, кроме токена разделителя. ANTLR делает это с помощью оператора отрицания, но в других генераторах синтаксических анализаторов ([Parser Generator \(277\)](#)) такой возможности может не оказаться. В этом случае вы должны сделать что-то наподобие следующего.

```
item : (ID | 'item')* SEP;
```

В правиле вам потребуется указать список всех возможных ключевых слов, что существенно менее удобно, чем применение оператора отрицания.

Токены по-прежнему имеют верный тип, но в данном контексте они не используются. Грамматика с действиями может выглядеть следующим образом.

```
catalog returns [Catalog catalog = new Catalog()] :
  (i=item {$catalog.addItem(i.itemName);})*
;
item returns [String itemName] :
  'item' name=item_name SEP
  {$itemName = $name.result;}
;
item_name returns [String result = ""] :
  (n=~SEP {$result += $n.text + " ";})*
  {$result = $result.trim();}
;
SEP : ';' ;
```

В этом случае текст собирается из всех токенов в имени, независимо от их типов. В случае построения дерева ([Tree Construction \(289\)](#)) вы должны были бы поступить аналогично, собирая все токены имени в один список и игнорируя их типы при обработке дерева.

28.2. Когда это использовать

Альтернативная токенизация актуальна при использовании синтаксически управляемой трансляции (Syntax-Directed Translation (229)), когда лексический и синтаксический анализы разделены, что является распространенным случаем. Вам стоит подумать о ее применении, если у вас есть раздел специального текста, который не должен быть токенизирован по обычной схеме.

Часто возникающие ситуации, в которых возможно применение альтернативной токенизации, — это использование ключевых слов, которые в определенном контексте не должны рассматриваться как таковые, требование возможности использования текста произвольного вида (обычно для текстовых описаний) или применение внешнего кода (Foreign Code (315)).

Глава 29

Вложенные операторные выражения

Nested Operator Expression

Операторное выражение, могущее рекурсивно содержать выражение того же вида (например, арифметические или логические выражения)

2 * (4 + 5)

Шаблон, именуемый вложенным операторным выражением, достаточно растяжим, поскольку является не столько решением, сколько распространенной проблемой синтаксического анализа. Это в особенности относится к восходящим синтаксическим анализаторам, в которых необходимо избегать левой рекурсии.

29.1. Как это работает

Вложенные операторные выражения имеют два аспекта, которые усложняют работу с ними, — это их рекурсивная природа (соответствующее правило появляется в своем теле) и наличие приоритетов операторов. Точный рецепт борьбы с этим частично зависит от конкретного используемого генератора синтаксических анализаторов (*Parser Generator* (277)), но имеются и некоторые полезные общие принципы, применяемые в данном случае. Самое большое различие заключается в том, как с такими выражениями работают восходящие и нисходящие синтаксические анализаторы.

Здесь мой рабочий пример — это калькулятор, который может обрабатывать четыре обычные арифметические операции (+, -, *, /), группирование с помощью скобок, а также возвведение в степень (с использованием "**) и вычисление корня (с использованием "/"). Можно также добавить унарный минус для получения отрицательных чисел.

Этот выбор операторов означает наличие нескольких уровней приоритета. Наивысший приоритет имеет унарный минус, после чего идут возвведение в степень и вычисление корня, затем — умножение и деление, и наконец — сложение и вычитание. Я добавил в задачу возвведение в степень и вычисление корня, так как они являются правоассоциативными операторами, в то время как прочие бинарные операторы левоассоциативны.

29.1.1. Применение восходящих синтаксических анализаторов

Я начну с восходящих синтаксических анализаторов, описать которые более легко. Базовая грамматика для арифметических выражений с четырьмя действиями и скобками выглядит следующим образом.

```
expr ::= NUMBER:n { : RESULT = new Double(n); : }
| expr:a PLUS expr:b { : RESULT = a + b; : }
| expr:a MINUS expr:b { : RESULT = a - b; : }
| expr:a TIMES expr:b { : RESULT = a * b; : }
| expr:a DIVIDE expr:b { : RESULT = a / b; : }
| expr:a POWER expr:b { : RESULT = Math.pow(a,b); : }
| expr:a ROOT expr:b { : RESULT = Math.pow(a,(1.0/b)); : }
MINUS expr:e { : RESULT = - e; : } %prec UMINUS
LPAREN expr:e RPAREN { : RESULT = e; : }

;
```

Эта грамматика использует Java-генератор синтаксических анализаторов (Parser Generator (277)) CUP, который, по сути, представляет собой версию классического Yacc для Java. Грамматика охватывает структуру синтаксиса выражения в одном правиле, с альтернативами для каждого вида операторов, с которыми мы должны работать, включая базовый случай, когда в качестве выражения просто представлено число.

В отличие от обычно использующегося для примеров в данной книге ANTLR, здесь нельзя помещать лiteralные токены в грамматические правила (вот почему я использовал имена токенов типа PLUS вместо знака +). Отдельный лексический анализатор транслирует операторы и числа в требуемый синтаксическим анализатором вид.

Для проведения вычислений я использую встроенную интерпретацию (Embedded Interpretation (311)), поэтому после каждой альтернативы вы видите код действия, вычисляющий результат выражения. (Коды действий отделяются разделителями { : и :}, чтобы иметь возможность использовать в коде фигурные скобки.) Для возвращаемого значения используется специальная переменная RESULT; элементы правила помечены суффиксной меткой вида :label.

Базовые правила грамматики обрабатывают рекурсивную структуру, но не приоритеты: $1 + 2 * 3$ должно трактоваться как $1 + (2 * 3)$, но сейчас это не так. Чтобы добиться верной трактовки, можно использовать набор объявлений приоритетов.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
precedence right POWER, ROOT;
precedence left UMINUS;
```

Каждая инструкция приоритета перечисляет некоторые операторы с одним уровнем приоритета и указывает их ассоциативность (левую или правую). Приоритеты перечисляются от низшего к высшему.

Приоритеты могут упоминаться и в правилах грамматики, как в случае унарного минуса, — %prec UMINUS. UMINUS представляет собой ссылку на токен, которая не является реальным токеном; она используется только для указания приоритета данного правила. С помощью такого контекстно-зависимого приоритета я указываю генератору синтаксических анализаторов, что данное правило должно использовать не приоритет по умолчанию для оператора " - ", а приоритет несуществующего оператора UMINUS.

В терминах языка программирования проблема, которую решают приоритеты, заключается в устранении неоднозначности. При отсутствии приоритетов синтаксический анализатор данной грамматики может рассматривать $1 + 2 * 3$ и как $(1 + 2) * 3$,

и как `1 + (2 * 3)`, что делает ее неоднозначной. То же самое относится и к выражению `1 + 2 + 3`, хотя мы (люди) знаем, что в данном случае это никакого значения не имеет. Этот пример показывает, почему мы обязаны указывать направление ассоциативности (хотя оно и не имеет значения для выражения `c + i *`).

Комбинация простого рекурсивного правила грамматики и объявлений приоритетов делает обработку вложенных выражений в восходящем синтаксическом анализаторе очень простой.

29.1.2. Нисходящие синтаксические анализаторы

Работа с вложенными операторными выражениями в случае нисходящих синтаксических анализаторов оказывается более сложной. Вы не можете использовать простую рекурсивную грамматику, так как она приводит к левой рекурсии, с которой такие синтаксические анализаторы справиться не в состоянии. В результате требуется использовать ряд грамматических правил, которые одновременно решают проблему как левой рекурсии, так и приоритетов. Однако получающаяся в результате грамматика оказывается гораздо менее ясной. Именно эта неясность грамматик и является основной причиной предпочтения многими программистами восходящих синтаксических анализаторов.

Давайте рассмотрим правила в ANTLR. Начнем с правил верхнего уровня, вводящих операторы с наименьшим уровнем приоритета. Для чистого синтаксического анализа без действий эти правила имеют следующий вид.

```
expression : mult_exp ( ('+' | '-') mult_exp )* ;
mult_exp   : power_exp ( ('*' | '/') power_exp )* ;
```

Здесь вы видите шаблон для левоассоциативных операторов. Тело правила начинается со ссылки на следующее по уровню приоритета правило, за которым следует повторяющаяся группа с операторами и той же ссылкой. Всегда упоминается правило, следующее за данным правилом в порядке возрастания приоритетов.

Правило для операторов возведения в степень и извлечения корня демонстрирует шаблон для правоассоциативного оператора.

```
power_exp : unary_exp ( ('**' | '//') power_exp )? ;
```

Обратите внимание на пару отличий от шаблона для левоассоциативных операторов. Во-первых, правая часть правила представляет собой рекурсивную ссылку на само правило, а не на правило с более высоким приоритетом. Во-вторых, вместо повторения группы используется необязательная группа. Рекурсия допускает объединение нескольких возведений в степень, а правая рекурсия по своей сути правоассоциативна.

Унарное выражение требует поддержки необязательного знака минуса.

```
unary_exp
: '-' unary_exp
| factor_exp
; ;
```

Обратите внимание на то, что я использую рекурсию при наличии минуса (чтобы разрешить несколько минусов подряд в выражении), но когда его нет, используется следующее в порядке возрастания приоритета правило.

Теперь перейдем к правилам наименьшего уровня (с самым высоким приоритетом) — к атомам языка (в данном случае это просто числа) и скобкам.

```
factor_exp : NUMBER | par_exp ;
par_exp    : '(' expression ')' ;
```

Выражения в скобках приводят к глубокой рекурсии, так как ссылаются на правила верхнего уровня.

(*Примечание для ANTLR*. Если грамматика ограничивается только приведенными правилами, ANTLR может не работать, поскольку при этом нет правила верхнего уровня, не вызываемого другими правилами (что может привести к сообщению об отсутствии стартового правила). Поэтому в грамматику необходимо добавить правило наподобие `prog : expression;;`)

Как вы можете видеть, грамматика оказывается намного сложнее, чем в случае восходящего синтаксического анализатора. Вы тратите время на обход ограничений генератора синтаксических анализаторов (*Parser Generator* (277)), а не на выражение своих намерений. В результате получается искаженная грамматика, из-за чего многие программисты предпочитают иметь дело с восходящими синтаксическими анализаторами. Сторонники нисходящего синтаксического анализа утверждают, что такое происходит только с вложенными выражениями, и что это вполне допустимая цена за отсутствие других недостатков, связанных с восходящими синтаксическими анализаторами.

Еще одним следствием искаженной грамматики является то, что в результате получается более сложное синтаксическое дерево. Можно было бы ожидать, что синтаксическое дерево для $1 + 2$ должно быть чем-то вроде следующего.

```
+  
 1  
 2
```

Но вместо этого мы получаем вот что.

```
+  
  mult_exp  
    power_exp  
      unary_exp  
        factor_exp  
          1  
  mult_exp  
    power_exp  
      unary_exp  
        factor_exp  
          2
```

Все грамматические правила для приоритетов добавляют много лишних узлов в синтаксическое дерево. На практике это не так важно; вам нужно писать код для обработки этих узлов только для случаев, когда они полезны, но иногда они просто раздражают.

Только что показанная грамматика — “чистая”, не создающая никакого вывода. Попытки что-то делать в ходе синтаксического анализа часто привносят некоторые дополнительные искажения. Для повторения калькулятора из встроенной интерпретации (*Embedded Interpretation* (311)) правило верхнего уровня должно выглядеть следующим образом.

```
expression returns [double result]  
: a=mult_exp      {$result = $a.result;}  
| '+' b=mult_exp {$result += $b.result;}  
| '-' b=mult_exp {$result -= $b.result;}  
)*;  
;
```

Здесь взаимодействие кода действия и грамматики сложнее, чем обычно. Поскольку у нас может быть любое количество членов на этом уровне (например, $1 + 2 + 3 + 4$), нужно объявить переменную для накопления результата в начале выражения и накапливать

вать значения при повторении группы. Кроме того, поскольку нужно выполнять разные действия в зависимости от того, плюс это или минус, следует расширить альтернативы, то есть перейти от `('+' | '-') mult_expr` к `('+' mult_expr | '-' mult_expr)`. Это привносит определенное дублирование, но так часто бывает, как только требуется выполнять какие-то осмыслиенные действия с грамматикой. Построение дерева (Tree Construction (289)) часто снижает остроту этой проблемы, но даже в этом случае вы можете захотеть возвращать разные типы узлов для плюса и минуса, что потребует расширения альтернатив.

Во всех приведенных выше примерах использовался ANTLR, поскольку это нисходящий синтаксический анализатор, с которым вы встретитесь вероятнее всего. Различные нисходящие синтаксические анализаторы имеют немного отличающиеся проблемы и решения. Обычно в их документации говорится о том, как именно следует справляться с левой рекурсией.

29.2. Когда это использовать

Как я говорил ранее, вложенные операторные выражения не совсем соответствуют моему обычному описанию шаблонов, и если бы я был писателем получше, то постарался бы описать их как-то иначе. Так что здесь раздел “Когда это использовать” служит только того, чтобы показать мою приверженность традиции, — в книге не должно быть описания шаблона без такого раздела, и точка.

Глава 30

Символ новой строки в качестве разделителя

Newline Separators

В качестве разделителя между инструкциями используется символ новой строки

Первая инструкция
Вторая инструкция
Третья инструкция

30.1. Как это работает

Использование символов новой строки в качестве конца инструкции — распространенная возможность языков программирования. Эта практика хорошо укладывается в шаблон трансляции, управляемой разделителями (Delimiter-Directed Translation (213)), поскольку символы новой строки используются в качестве основного разделителя для разбиения исходного текста. В результате здесь мне, по сути, нечего добавить к указанному контексту.

Однако в случае синтаксически управляемой трансляции (Syntax-Directed Translation (229)) символы новой строки в качестве разделителя вносят определенные тонкости и ставят ловушки, в которые вы можете попасть.

(Конечно, возможно применение символов новой строки и для синтаксических цепей, отличных от разделения инструкций, но я еще никогда с этим не сталкивался.)

Причина, по которой символы новой строки в качестве разделителей и синтаксически управляемая трансляция не слишком хорошо работают вместе, заключается в том, что символы новой строки играют две роли при их применении в качестве разделителей. Помимо их синтаксической роли, они еще участвуют в форматировании. В результате они могут появляться в таких местах, где никто не ожидает разделителя инструкций.

Вот как, по моему представлению, должна выглядеть обычная грамматика, использующая окончания строк в качестве разделителей.

```
catalog    : statement*;
statement  : 'item' ID EOL;
EOL        : '\r'? '\n';
```

```
ID  : ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')+;
WS  : (' ' | '\t')+ {$channel = HIDDEN;};
```

Эта грамматика работает с простым списком элементов, каждая строка которого представляет собой ключевое слово `item`, за которым следует идентификатор элемента. У меня уже вошло в привычку использовать эту грамматику в качестве аналога "Hello World" для синтаксического анализа — настолько она проста: ключевое слово, идентификатор, символ новой строки... Однако имеется ряд распространенных ситуаций, вызывающих вопросы.

- Пустые строки между инструкциями
- Пустые строки до первой инструкции
- Пустые строки после последней инструкции
- Последняя инструкция в последней строке не завершается символом новой строки

Первые три пункта связаны с пустыми строками, но несмотря на схожесть, для них могут потребоваться различные способы обработки в грамматике, так что обязательно нужно протестировать каждый из перечисленных случаев. Пожалуй, это самое важное — убедиться в том, что все тесты корректно проходятся. Ниже у меня приведены решения всех этих проблем, но хорошие тесты являются ключом к обеспечению гарантии корректной обработки всех ситуаций.

Один из способов обработки пустых строк — использовать правило для конца инструкции, гласящее, что ему может соответствовать несколько символов новой строки. Логичное место для размещения этого правила — лексический анализатор, так как это регулярное правило (я использую здесь термин “регулярный” в смысле теории языков, как означающий возможность использования регулярных выражений для проверки соответствия). Все несколько осложняется последней ситуацией — инструкцией, у которой отсутствует завершение символом новой строки. Для обработки этого случая требуется возможность проверки соответствия концу файла в лексическом анализаторе, которая, в зависимости от вашего генератора синтаксических анализаторов ([Parser Generator \(277\)](#)), может оказаться недоступной. Так, в ANTLR для этого нужно новое правило для конца инструкции.

```
catalog      : verticalSpace statement*;
statement    : 'item' ID eos;
verticalSpace : EOL*;
eos          : EOL+ | EOF;
```

Отсутствие символа новой строки в последней строке файла часто оказывается достаточно неприятной ситуацией. Насколько неприятной — зависит от того, как генератор синтаксических анализаторов работает с концом файла. ANTLR делает его доступным для анализатора в виде токена, и именно поэтому я могу выполнять проверку на соответствие в правиле синтаксического анализатора (а не в правиле лексического анализатора). Другие генераторы синтаксических анализаторов делают проверку соответствия концу файла очень трудной или попросту невозможной. Один из вариантов, который стоит рассмотреть, — добавление в конец файла символа новой строки либо посредством лексического анализатора (если это возможно), либо еще до него. Такой подход может помочь избежать применения запутанного решения.

Другой подход к терминаторам инструкций — подход, который позволяет избежать проблемы завершающего терминатора в общем случае (т.е. рассматривать их как разделители, а не как терминаторы). Это приводит к правилу такого вида.

```
catalog      : verticalSpace statement
              (separator statement)* verticalSpace;
```

```
statement      : 'item' ID;
separator      : EOL+;
verticalSpace : EOL*;
```

Я предпочитаю именно этот стиль. Вместо определения дополнительного правила `verticalSpace` я могу использовать `separator?`.

Третья альтернатива — рассматривать тело инструкции как необязательный элемент каждой строки каталога.

```
catalog      : line* ;
line        : EOL | statement EOF | statement EOL;
statement   : 'item' ID;
```

Это правило для корректной обработки завершающего символа новой строки требует наличия функциональной возможности проверки соответствия концу файла. При ее отсутствии необходимо что-то наподобие следующего.

```
catalog      : line* statement? ;
line        : statement? EOL;
statement   : 'item' ID;
```

На мой взгляд, это не столь понятно, как представленные ранее грамматики, но и не требует функциональной возможности проверки соответствия концу файла

Отдельным элементом, который также может вызвать много проблем с символом новой строки в качестве разделителя, являются комментарии. Комментарии, которые продолжаются до конца строки, очень полезны. При игнорировании символов новой строки можно легко выполнять проверку соответствия комментарию так, чтобы символ новой строки также входил в него (кстати, этот способ может выручить вас, когда последняя строка представляет собой комментарий, не заканчивающийся символом новой строки). Но при использовании символа новой строки в качестве разделителя такое его поглощение может привести к проблемам, поскольку комментарии часто появляются после инструкций, как, например, показано ниже.

```
item laser # Некоторое пояснение
```

Если проверка соответствия комментарию поглощает символ новой строки, вы теряете терминатор инструкции.

Обычно эта проблема решается с помощью выражения наподобие следующего.

```
COMMENT : '#' ~'\n'* {skip();};
```

В терминах классического регулярного выражения оно имеет такой вид.

```
Comment = #[^\\n]*
```

Последний вопрос, о котором не следует забывать, заключается в предоставлении некоторого вида символа продолжения для слишком длинных строк. Его легко обработать с помощью правила лексического анализа наподобие следующего.

```
CONTINUATION : '&' WS* EOL {skip();};
```

30.2. Когда это использовать

Решение использовать символы новой строки как разделители на самом деле представляет собой пару решений: решение о применении разделителей и решение об использовании в качестве разделителя символа новой строки.

Ограниченная структура DSL часто означает, что вы можете прожить и без разделителей инструкций. Синтаксический анализатор, как правило, в состоянии выяснить контекст анализа на основании различных используемых ключевых слов. Так, например, грамматика контроллера мисс Грант из введения не использует какие-либо разделители инструкций, но при этом весьма просто анализируется.

Разделители инструкций могут упростить локализацию ошибок. Для того чтобы синтаксический анализатор мог локализовать ошибки, нужен некий вид контрольных точек, способных отслеживать выполнение анализа. Без таких контрольных точек ошибка в одной строке сценария может не быть очевидной для синтаксического анализатора до прохода им нескольких следующих строк, что приводит к путанице в сообщениях об ошибках. Разделители инструкций часто могут играть эту роль, хотя они и не являются единственным подходящим механизмом; часто эту роль играют ключевые слова.

Если вы решили использовать разделители инструкций, перед вами встает вопрос выбора между видимыми символами, такими как точка с запятой, и символами новой строки. В применении символов новой строки есть тот положительный момент, что вы получаете по одной инструкции в каждой строке, так что этот разделитель не приводит к появлению синтаксического шума в DSL. Это особенно ценно при работе с не программистами, хотя и многие программисты (включая меня) предпочитают такие разделители. Недостатком символа новой строки в качестве разделителя является то, что синтаксически управляемая трансляция (*Syntax-Directed Translation* (229)) становится более сложной и приходится использовать методы, которые я описал выше. Кроме того, необходимо создание и выполнение тестов, охватывающих распространенные проблемные случаи. В целом, однако, я по-прежнему предпочитаю использовать в качестве разделителя инструкций символ новой строки, а не какой-то видимый символ.

Глава 31

Прочие вопросы

Дойдя до этой главы, я остановился и задумался. Как при написании программного обеспечения наступает момент, когда следует остановиться и сократить список функциональных возможностей программы, просто для того чтобы она была написанной, а не вечно пишущейся, так и при написании книги следует притормозить и сократить список рассматриваемых в ней тем, для того чтобы книга была издана и ее можно было поднять руками.

Необходимость компромисса стала особенно очевидной для меня при описании внешних DSL. Имеется целый ряд тем, которые заслуживают дальнейшего расследования и написания книг. Это интересные темы, и, вероятно, полезные для читателя этой книги. Но каждая тема требует времени для изучения, а следовательно, отсрочки выпуска книги в свет. Поэтому мне пришлось оставить многие темы нетронутыми. Но несмотря на это, я не смог не собрать здесь ряд кратких замечаний по ряду тем.

Прошу вас помнить при чтении этой главы, что представленный в ней материал достаточно “сырой”. Это скорее предварительные наброски для полноценных глав, чем сколько-нибудь серьезно изученный материал.

31.1. Синтаксические отступы

Во многих языках программирования имеется строгая иерархическая структура элементов. Зачастую она кодируется с помощью вложенных блоков некоторого вида. Так, структуру Европы можно описать с помощью синтаксиса наподобие следующего.

```
Europe {
    Denmark
    France
    Great Britain {
        England
        Scotland
        #...
    }
    #...
}
```

В этом примере иллюстрируется распространенный способ, которым программисты всех мастей указывают иерархическую структуру своих программ. Синтаксическая информация о структуре находится между разделителями, в данном случае это фигурные скобки. Однако, читая структуру, вы уделяете больше внимания форматированию. Основная информация о структуре при чтении поступает к нам от отступов, а не от разделителей. Как истинный англичанин по крови я могу предпочесть форматировать приведенный выше список так.

```
Europe {
    Denmark
    France
    Great Britain {
        England
        Scotland
    }
}
```

Здесь отступы вводят в заблуждение, поскольку они не соответствуют фактической структуре, указываемой фигурными скобками. (Хотя отступы также достаточно информативны: они показывают распространенный британский взгляд на весь мир.)

То, что мы в основном воспринимаем структуры через отступы, является аргументом в пользу применения отступов для описания структуры. В этом случае я могу описать европейскую структуру примерно так.

```
Europe
  Denmark
  France
  Great Britain
    England
    Scotland
```

Здесь отступы не только определяют структуру, но и сообщают о ней глазам читающего исходный текст. Наиболее известно применение этого подхода в языке программирования Python; он используется также в YAML — языке описания структур данных.

С точки зрения удобства использования большое преимущество синтаксического отступа состоит в синхронизации определения и внешнего вида — вы не сможете ввести себя в заблуждение, изменения форматирование без изменения реальной структуры. (Текстовые редакторы с функцией автоматического форматирования убирают большую часть этого преимущества, но такая поддержка со стороны редакторов для DSL не характерна.)

При использовании синтаксического отступа нужно быть очень осторожными с символами табуляции и пробелов. Поскольку размер табуляции варьируется в зависимости от настроек редактора, смешивание символов табуляции и пробелов в файле может привести к путанице. Я рекомендую следовать подходу YAML и запрещать символы табуляции в любом языке, использующем синтаксические отступы. Любые неудобства от запрета табуляции будут значительно меньшими, чем от возможной путаницы при их разрешении.

Синтаксические отступы очень удобны в использовании, но вызывают реальные трудности при выполнении синтаксического анализа. Я затратил некоторое время на изучение синтаксических анализаторов Python и YAML и обнаружил немалые сложности, связанные с синтаксическими отступами.

Рассмотренные мною синтаксические анализаторы обрабатывают синтаксические отступы в лексическом анализаторе, который является частью синтаксически управляемой трансляции (*Syntax-Directed Translation* (229)), работающей с символами. (Управляемая разделителями трансляция (*Delimiter-Directed Translation* (213)), вероятно, не слишком удачный выбор для синтаксического отступа, поскольку синтаксические отступы представляют собой разновидность блочной структуры, с которой эта методика имеет проблемы.)

Распространенной и, я думаю, эффективной тактикой является использование в выделе лексического анализатора специальных токенов типа `indent` и `dedent`, когда он обнаруживает изменения в отступе. Такие мнимые токены позволяют создать синтаксический анализатор с помощью обычных методов обработки блоков — токены `indent` и `dedent` заменят `{` и `}`. Однако выполнение этих действий в обычном лексическом анализаторе — дело, которое можно оценить как лежащее между “трудно” и “невозможно”.

Лексические анализаторы создаются не для обнаружения изменений отступа и не для генерации мнимых токенов, не соответствующих конкретным символам исходного текста. Так что вам, вероятно, придется написать собственный лексический анализатор. (Хотя ANTLR может сделать это; взгляните на советы Парра (Parr) по работе с Python [22].)

Еще один возможный подход, который я бы, безусловно, был склонен попробовать, — это предварительная обработка входного текста до его передачи лексическому анализатору. Эта предварительная обработка сосредоточена исключительно на задаче распознавания изменения отступов и должна при их обнаружении вставлять в исходный текст специальные маркеры. Эти маркеры могут распознаваться лексическим анализатором обычным способом. В этом случае задача заключается в том, чтобы выбрать маркеры, которые не будут конфликтовать ни с одним из элементов языка. Вы также должны разобраться, как эта обработка может помешать диагностике, при которой требуется указать номер строки и столбца. Но такой подход значительно упрощает лексический анализ синтаксических отступов.

31.2. Модульная грамматика

Предметно-ориентированные языки тем лучше, чем больше они ограничены. Ограниченнная выразительность обеспечивает простоту их понимания, применения и обработки. Одной из самых больших опасностей DSL является желание добавить выразительность, что приводит язык в ловушку общего назначения.

Для того чтобы избежать этой ловушки, полезно иметь возможность объединять независимые DSL. Для этого понадобится возможность выполнять независимый синтаксический анализ различных частей. Если вы используете синтаксически управляемую трансляцию (*Syntax-Directed Translation* (229)), это означает применение различных грамматик для разных DSL при возможности соединения этих грамматик в одном общем синтаксическом анализаторе. Хотелось бы иметь возможность обращаться из одной грамматики к другой, причем так, чтобы изменения в грамматике, к которой выполняется обращение, не требовали изменений в грамматике, из которой выполняется такое обращение. Модульные грамматики позволяют применять повторно используемые грамматики таким же образом, как в настоящее время применяются повторно используемые библиотеки.

Модульные грамматики, весьма полезные для работы DSL, — не слишком хорошо разработанная и изученная область мира языков. Эта тема изучается, но на момент написания этого раздела мне не встречалось ничего действительно серьезного.

Большинство генераторов синтаксических анализаторов (*Parser Generator* (277)) используют отдельные лексические анализаторы, что еще больше усложняет применение модульных грамматик, поскольку дочерней грамматике обычно нужен лексический анализатор, отличный от лексического анализатора родительской грамматики. Эту проблему можно обойти, используя альтернативную токенизацию (*Alternative Tokenization* (325)), но это накладывает ограничения на то, как дочерняя грамматика может существовать с родительской. В настоящее время складывается впечатление, что для работы с модульными грамматиками в большей степени подходят анализаторы, не разделенные на лексический и синтаксический.

В данный момент простейший способ работы с раздельными языками — рассматривать их как внешний код (*Foreign Code* (315)), помещая исходный текст на дочернем языке в буфер, который затем анализируется отдельно.

Часть IV

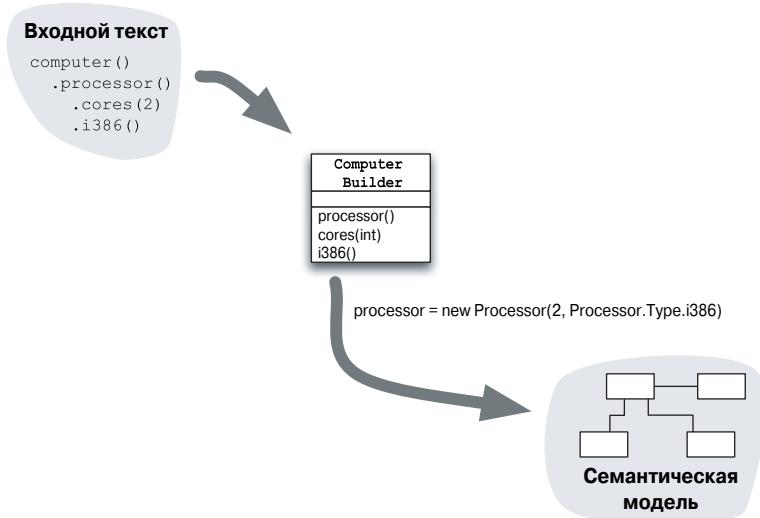
Вопросы создания внутренних DSL

Глава 32

Построитель выражений

Expression Builder

Объект или семейство объектов, предоставляющее свободный интерфейс поверх обычного API командных запросов



Интерфейсы прикладного программирования (API), как правило, предназначены для предоставления набора самостоятельных методов объектов. В идеальном случае каждый из этих методов может быть понятен по отдельности. Я называю этот стиль “API командных запросов”; он настолько естественный, что у нас нет для него общего названия. DSL требуют API другого вида, который я называю **свободным интерфейсом** (fluent interface) и который разработан для повышения удобочитаемости всего выражения. Применение свободных интерфейсов приводят к методам, которые имеют мало смысла сами по себе и часто нарушают правила, принятые для разработки хороших API командных запросов.

Построитель выражений предоставляет свободный интерфейс в качестве отдельного слоя над регулярным API. Таким способом мы получаем два стиля интерфейса, которые при этом четко изолированы один от другого, что позволяет упростить работу с ними.

32.1. Как это работает

Построитель выражений — это объект, предоставляющий свободный интерфейс, который он затем транслирует в вызовы лежащего в его основе API командных запросов. Вы рассматриваете его как слой, транслирующий свободный интерфейс в API командных запросов. Построитель выражений часто представляет собой композит (*Composite* [15]), использующий дочерние построители выражений для создания подвыражений в общем предложении.

Как именно вы организуете построитель выражений, во многом зависит от вида предложений, с которыми вы работаете. Соединение методов в цепочки (*Method Chaining* (375)) представляет собой последовательность вызовов методов, каждый из которых возвращает построитель выражений; вложенные функции (*Nested Function* (361)) могут использовать построитель выражений, который является суперклассом или набором глобальных функций. В результате в этом описании шаблона я не могу давать какие-либо общие правила, на что должен быть похож построитель выражений. Вам нужно рассмотреть различные виды построителей выражений, приведенные в описаниях других шаблонов внутренних DSL. Все, что я могу сделать, — это немного поговорить о некоторых общих принципах, которые, как я надеюсь, помогут вам в разработке собственных построителей.

Одним из наиболее важных является вопрос о том, достаточно ли одного объекта построителя выражений для всего DSL или же нужно использовать несколько построителей для различных частей DSL. Множественные построители выражений обычно следуют древовидной структуре, которая в действительности является синтаксическим деревом DSL. Чем сложнее DSL, тем более важным становится дерево построителей выражений.

Один из самых полезных советов относительно получения ясно отделимых множеств построителей выражений заключается в наличии четко определенной семантической модели (*Semantic Model* (171)). Семантическая модель должна иметь объекты с интерфейсами командных запросов, которыми можно управлять без каких-либо свободных конструкций. Убедиться в этом можно, написав тесты для семантической модели, которые не используют какой-либо DSL. Слишком строго следовать этому правилу может оказаться неразумным. В конце концов, весь смысл внутренних DSL состоит в упрощении работы с этими объектами, так что обычно и в тестах манипулировать ими легче с помощью DSL, чем с помощью интерфейса командных запросов. Тем не менее обычно я включаю по крайней мере несколько тестов, использующих исключительно интерфейс командных запросов.

Построители выражений могут работать поверх этих объектов модели. Вы должны быть в состоянии протестировать построители выражений, сравнивая работу с объектами семантической модели с прямыми вызовами API командных запросов семантической модели.

32.2. Когда это использовать

Я рассматриваю построитель выражений как шаблон, применяемый по умолчанию, т.е. я использую его почти всегда, если нет уважительной причины не делать этого.

Это, конечно, приводит к вопросу “А когда же построитель выражений оказывается не очень хорошей идеей?”

Альтернативой использованию построителя выражений является размещение свободных методов в самой семантической модели (*Semantic Model* (171)). Главной причиной, по которой я не люблю этот подход, является смешивание API для построения се-

мантической модели с методами, которые работают с этой моделью. Как правило, каждый из этих двух аспектов и так достаточно сложен. Понимание логики выполнения семантической модели часто требует больших усилий, особенно если она представляет собой альтернативную вычислительную модель. Свободные интерфейсы обладают собственной логикой. Таким образом, моя аргументация в пользу построителя выражений сводится к разделению сфер влияния. Отделение логики построения от логики выполнения облегчает понимание.

Еще одной причиной разделения является необычность свободного интерфейса. Смешивание свободных методов и методов командных запросов в одном классе является смешиванием двух совершенно разных способов представления API. Тот факт, что свободные API встречаются реже, а потому разработчики менее знакомы с ними, только усугубляет ситуацию.

Наилучший аргумент против применения построителя выражений, который я могу себе представить, — когда логика выполнения семантической модели очень проста, и ее смешивание с логикой построения на самом деле не привносит дополнительной сложности.

Тем не менее довольно часто программисты их объединяют. Это происходит отчасти потому, что некоторые из них не знакомы с шаблоном построителя выражений, а отчасти потому, что люди не видят смысла в дополнительных классах для построителя выражений. Я же предпочитаю много маленьких классов нескольким большим, так что мои фундаментальные принципы проектирования призывают меня использовать построитель выражений.

32.3. Свободный календарь с построителем и без него (Java)

Чтобы изучить работу построителя выражений, рассмотрим создание календаря событий с построителем и без него. По сути, я хочу добавлять события в календарь с помощью DSL следующим образом.

```
cal = new Calendar();
cal.add("DSL tutorial")
    .on(2009, 11, 8)
    .from("09:00")
    .to("16:00")
    .at ("Aarhus Music Hall")
;

cal.add("Making use of Patterns")
    .on(2009, 10, 5)
    .from("14:15")
    .to("15:45")
    .at("Aarhus Music Hall")
;
```

Для этого я создаю свободные интерфейсы для классов календаря и событий.

```
class Calendar...
private List<Event> events = new ArrayList<Event>();
public Event add(String name) {
    Event newEvent = new Event(name);
    events.add(newEvent);
    return newEvent;
}

class Event...
private String name, location;
```

```

private LocalDate date;
private LocalTime startTime, endTime;

public Event(String name) {
    this.name = name;
}
public Event on(int year, int month, int day) {
    this.date = new LocalDate(year, month, day);
    return this;
}
public Event from (String startTime) {
    this.startTime = parseTime(startTime);
    return this;
}
public Event to (String endTime) {
    this.endTime = parseTime(endTime);
    return this;
}
private LocalTime parseTime(String time) {
    final DateTimeFormatter fmt =
        ISODateTimeFormat.hourMinute();
    return new LocalTime(fmt.parseDateTime(time));
}
public Event at(String location) {
    this.location = location;
    return this;
}

```

(Встроенные классы даты и времени Java более чем ужасны, поэтому я использую здесь очень удобный пакет JodaTime.)

Это весьма удобный для моих целей интерфейс, но его стиль отличается от стиля, к которому привыкло большинство программистов. Показанные методы выглядят несколько странными рядом с такими методами, как `getStartTime()` или `contains(LocalDate)`. Это особенно справедливо, если вы хотите, чтобы люди могли изменять события извне контекста DSL. В этом случае наряду с показанными методами вам понадобятся обычные модифицирующие методы, такие как `setStartTime`. (Использование свободного интерфейса вне контекста может привести к появлению трудно читаемого кода.)

Основное назначение построителя выражений — перемещать свободные методы в отдельный класс построителя.

```

class CalendarBuilder...
private Calendar content = new Calendar();

public CalendarBuilder add(String name) {
    content.addEvent(new Event());
    getCurrentEvent().setName(name);
    return this;
}
private Event getCurrentEvent() {
    return content.getEvents()
        .get(content.getEvents().size() - 1);
}
public CalendarBuilder on(int year, int month, int day) {
    getCurrentEvent().setDate(new LocalDate(year,month,day));
    return this;
}
public CalendarBuilder from(String startTime) {
    getCurrentEvent().setStartTime(parseTime(startTime));
    return this;
}

```

```

    }
    public CalendarBuilder to(String startTime) {
        getCurrentEvent().setEndTime(parseTime(startTime));
        return this;
    }
    private LocalTime parseTime(String startTime) {
        final DateTimeFormatter fmt =
            ISODateTimeFormat.hourMinute();
        return new LocalTime(fmt.parseDateTime(startTime));
    }
    public CalendarBuilder at (String location) {
        getCurrentEvent().setLocation(location);
        return this;
    }
}

```

Это влечет за собой некоторое изменение в использовании DSL.

```

CalendarBuilder builder = new CalendarBuilder();
builder
    .add("DSL tutorial")
    .on (2009, 11, 8)
    .from("09:00")
    .to ("16:00")
    .at ("Aarhus Music Hall")
    .add("Making use of Patterns")
    .on (2009, 10, 5)
    .from("14:15")
    .to ("15:45")
    .at ("Aarhus Music Hall")
;
calendar = builder.getContent();

class CalendarBuilder...
public Calendar getContent() {
    return content;
}

```

32.4. Использование для календаря нескольких построителей (Java)

Вот абсурдно простая версия использования нескольких построителей с одним и тем же примером календаря. Предположим, что событие является неизменным, и все данные должны быть созданы в конструкторе. Это избавляет меня от необходимости создавать еще один пример.

При этом для создания свободного выражения мне нужно собрать данные события. Это можно было бы сделать с помощью полей в построителе календаря (например, `currentEventStartTime`), но мне кажется, что для этого лучше использовать построитель события (по сути — построитель конструкции (`Construction Builder (191)`)).

Сценарий DSL остается тем же, что и в случае единственного построителя.

```

CalendarBuilder builder = new CalendarBuilder();
builder
    .add("DSL tutorial")
    .on (2009, 11, 8)
    .from("09:00")
    .to ("16:00")
    .at ("Aarhus Music Hall")
    .add("Making use of Patterns")
    .on (2009, 10, 5)

```

```

.from("14:15")
.to ("15:45")
.at ("Aarhus Music Hall")
;
calendar = builder.getContent();

```

Построитель календаря отличается тем, что он хранит список построителей событий, а `add` возвращает построитель события.

```

class CalendarBuilder...
private List<EventBuilder> events =
new ArrayList<EventBuilder>();

public EventBuilder add(String name) {
    EventBuilder child = new EventBuilder(this);
    events.add(child);
    child.setName(name);
    return child;
}

```

Построитель события собирает данные о событии в своих полях с помощью свободного интерфейса.

```

class EventBuilder...
private CalendarBuilder parent;

private String name, location;
private LocalDate date;
private LocalTime startTime, endTime;

public EventBuilder(CalendarBuilder parent) {
    this.parent = parent;
}
public void setName(String arg) {
    name = arg;
}
public EventBuilder on(int year, int month, int day) {
    date = new LocalDate(year, month, day);
    return this;
}
public EventBuilder from(String startTime) {
    this.startTime = parseTime(startTime);
    return this;
}
public EventBuilder to(String endTime) {
    this.endTime = parseTime(endTime);
    return this;
}
private LocalTime parseTime(String startTime) {
    final DateTimeFormatter fmt =
        ISODateTimeFormat.hourMinute();
    return new LocalTime(fmt.parseDateTime(startTime));
}
public EventBuilder at (String location) {
    this.location = location;
    return this;
}

```

Метод `add` служит акцентом для указания следующего события. Поскольку этот вызов получает построитель события, ему требуется соответствующий метод, делегирующий родителю создание нового построителя события.

```
class EventBuilder...
public EventBuilder add(String name) {
    return parent.add(name);
}
```

Когда у построителя запрашивается его содержимое, он создает полную структуру объектов семантической модели (**Semantic Model** (171)).

```
class CalendarBuilder...
public Calendar getContent() {
    Calendar result = new Calendar();
    for (EventBuilder e : events)
        result.addEvent(e.getContent());
    return result;
}

class EventBuilder...
public Event getContent() {
    return new Event(name, location, date,
                    startTime, endTime);
}
```

В Java вариацией этой схемы является превращение построителя дочернего объекта во внутренний класс родителя. При таком подходе поле родителя не понадобится. (Я не поступаю так в примерах данной книги, поскольку, как мне кажется, это заведет нас в слишком глубокие для многоязычной книги дебри Java.)

Глава 33

Последовательность функций

Function Sequence

Объединение вызовов функций в последовательность инструкций

```
computer();
  processor();
    cores(2);
    speed(2500);
    i386();
  disk();
    size(150);
  disk();
    size(75);
    speed(7200);
    sata();
```

33.1. Как это работает

Последовательность функций производит серию вызовов, не связанных друг с другом ничем, кроме порядка вызовов во времени; самое главное, что они не связаны один с другим никакими данными. В результате любые отношения между вызовами должны осуществляться путем анализа данных, так что интенсивное использование последовательности функций означает интенсивное применение переменных контекста (Context Variable (187)).

Для использования последовательности функций удобочитаемым образом обычно желательны вызовы “голых”, т.е. неквалифицированных, функций. Наиболее очевидный способ достичь этого — применить вызовы глобальных функций, если это позволяет ваш язык программирования. Однако такое решение имеет два основных недостатка: статические данные синтаксического анализа и тот факт, что функции являются глобальными.

Проблема применения глобальных функций заключается в том, что они видимы везде в программе. Если ваш язык имеет некоторую разновидность пространства имен, вы можете (и должны) ее использовать, чтобы уменьшить область видимости вызовов функций построителем выражений (Expression Builder (349)). В частности, в Java соответствующим механизмом является статический импорт. Если ваш язык вообще не поддерживает механизмы глобальных функций (как C# или Java до версии 1.5), вам придется использовать явные методы класса для обработки вызовов. Это часто добавляет в DSL шум.

Глобальная видимость — очевидный недостаток глобальных функций, но часто самая раздражающая проблема заключается в том, что они вынуждают вас использовать статические данные. Статические данные зачастую приводят к проблемам, потому что вы никогда не можете точно знать, кто их использует, особенно в случае многопоточности. Эта проблема особенно пагубна для последовательности функций, потому что в этом случае для нормальной работы требуется много переменных контекста.

Хорошим решением как с точки зрения глобально видимых функций, так и с точки зрения статических данных синтаксического анализа является перенос в область видимости объекта (*Object Scoping* (387)). Это позволяет размещать функции в классе естественным объектно-ориентированным способом и дает объект для размещения данных синтаксического анализа. В результате при применении последовательности функций я предлагаю использовать перенос в область видимости объекта во всех, кроме самых простых, случаях.

33.2. Когда это использовать

В целом последовательность функций является наименее полезной из комбинаций вызовов функций, используемых в DSL. Применять переменные контекста (*Context Variable* (187)) для отслеживания местоположения в процессе анализа всегда неудобно, и это приводит к коду, который трудно понять и в котором легко ошибиться.

Несмотря на это иногда нужно использовать и последовательность функций. Часто DSL включает несколько инструкций высокого уровня; в этом случае часто имеет смысл организовать список инструкций в виде последовательности функций, если вам достаточно только одного списка и одной переменной контекста для отслеживания. Таким образом, последовательность функций является разумным выбором на самом верхнем уровне языка или на самом верхнем уровне вложенных замыканий (*Nested Closure* (403)). Однако на более низком уровне лучше образовывать выражения с помощью вложенных функций (*Nested Function* (361)) или соединения методов в цепочки (*Method Chaining* (375)).

Пожалуй, самая главная причина применения последовательности функций в том, что всегда нужно с чего-то начинать свой DSL, и это что-то должно быть последовательностью функций, даже если в ней имеется только один вызов. Это связано с тем, что все другие методы вызовов функций требуют некоторого контекста. Конечно, можно спорить о том, является ли последовательность из одного элемента настоящей последовательностью, но мне представляется, что это наилучший способ вписать последовательность функций в используемую мною концептуальную схему.

Простая последовательность функций представляет собой список элементов, поэтому очевидной альтернативой ее применению является список литералов (*Literal List* (415)).

33.3. Простая конфигурация компьютера (Java)

Рассмотрим пример повторяющейся конфигурации компьютера в виде DSL с применением последовательности функций.

```
computer();
processor();
cores(2);
speed(2500);
i386();
disk();
```

```

    size(150);
disk();
size(75);
speed(7200);
sata();

```

Отступы здесь используются исключительно для визуального указания структуры конфигурации; допускается совершенно произвольное использование пробелов. Сценарий представляет собой просто последовательность вызовов функций без каких-либо более глубоких отношений между ними. Эти более глубокие отношения создаются только с помощью переменных контекста (Context Variable (187)).

Последовательность функций использует вызов функций верхнего уровня, которые я должен разрешить некоторым способом. Я мог бы использовать статические методы и глобальное состояние, но я думаю, что это оскорбило бы ваш (да и мой) вкус проектировщика. Поэтому я воспользуюсь внесением в область видимости объекта (*Object Scoping* (387)). Это означает, что сценарий должен находиться в подклассе построителя компьютера, но это цена устранения необходимости в глобальных переменных.

Построитель содержит два типа данных: содержимое, связанное с создаваемыми процессорами и дисками, и переменные контекста для указания, над чем в настоящее время идет работа.

```

class ComputerBuilder...
private ProcessorBuilder processor;
private List<DiskBuilder> disks = new ArrayList<DiskBuilder>();

private ProcessorBuilder currentProcessor;
private DiskBuilder currentDisk;

```

Я использую построитель конструкции (*Construction Builder* (191)) для сбора данных для (неизменяемых) объектов семантической модели (*Semantic Model* (171)).

Вызов `computer` очищает переменные контекста.

```

class ComputerBuilder...
void computer() {
    currentDisk = null;
    currentProcessor = null;
}

```

Вызовы `processor()` и `disk()` создают значения для сбора данных и набор переменных контекста для отслеживания, над чем в настоящее время работает построитель.

```

class ComputerBuilder...
void processor() {
    currentProcessor = new ProcessorBuilder();
    processor = currentProcessor;
    currentDisk = null;
}

void disk() {
    currentDisk = new DiskBuilder();
    disks.add(currentDisk);
    currentProcessor = null;
}

```

После этого я могу собирать данные в соответствующий источник.

```

class ComputerBuilder...
void cores(int arg) {
    currentProcessor.cores = arg;
}

```

```

void i386() {
    currentProcessor.type = Processor.Type.i386;
}
void size(int arg) {
    currentDisk.size = arg;
}
void sata() {
    currentDisk iface = Disk.Interface.SATA;
}

```

Указать скорость несколько сложнее, так как я могу указывать скорость как процессора, так и диска в зависимости от контекста.

```

class ComputerBuilder...
void speed(int arg) {
    if (currentProcessor != null)
        currentProcessor.speed = arg;
    else if (currentDisk != null)
        currentDisk.speed = arg;
    else throw new IllegalStateException();
}

```

Завершив работу, построитель может вернуть семантическую модель.

```

class ComputerBuilder...
Computer getValue() {
    return new Computer(processor.getValue(),
                        getDiskValues());
}
private Disk[] getDiskValues() {
    Disk[] result = new Disk[disks.size()];
    for(int i = 0; i < disks.size(); i++)
        result[i] = disks.get(i).getValue();
    return result;
}

```

Чтобы все это начало работать, нужно “завернуть” сценарий в подкласс построителя выражений.

```

class ComputerBuilder...
public Computer run() {
    build();
    return getValue();
}
abstract protected void build();

public class Script extends ComputerBuilder {
    protected void build() {
        computer();
        processor();
        cores(2);
        speed(2500);
        i386();
        disk();
        size(150);
        disk();
        size(75);
        speed(7200);
        sata();
    }
}

```

Глава 34

Вложенные функции

Nested Function

Объединение функций путем вложения вызовов функций в качестве аргументов других вызовов

```
computer(
    processor(
        cores(2),
        speed(2500),
        i386
    ),
    disk(
        size(150)
    ),
    disk(
        size(75),
        speed(7200),
        SATA
    )
);
```

34.1. Как это работает

Представляя предложение DSL в виде вложенной функции, можно отражать иерархическую природу языка способом, принятым в базовом языке, а не только соглашением о форматировании исходного текста.

Важным свойством вложенных функций является порядок вычисления аргументов. И последовательность функций (Function Sequence (357)), и соединение методов в цепочку (Method Chaining (375)) вычисляют функции последовательно слева направо. В случае же вложенных функций аргументы функции вычисляются до самой функции. Проще всего это понять на примере песенки. Так, для верного порядка слов надо ввести елочка (`родилась (лесу(в()))`). Порядок вычисления влияет как на использование вложенных функций, так и на выбор этого метода из прочих альтернатив.

Вычисление внешней функции в последнюю очередь может быть очень полезным, поскольку оно обеспечивает встроенный контекст при работе с аргументами. Рассмотрим определение конфигурации процессора компьютера.

```
processor(cores(2), speed(2500), i386())
```

Положительным фактором здесь является то, что функции-аргументы могут вернуть полностью сформированные значения, которые затем функция `processor` может собрать в единое возвращаемое ею значение. Поскольку функция `processor` вычисляется последней, нам не стоит беспокоиться ни об останавливающем методе, как в соединении методов в цепочки, ни о переменной контекста (`Context Variable (187)`), которая нужна при использовании последовательности функций.

С обязательными элементами грамматики наподобие строк `parent ::= first second` вложенные функции справляются особенно хорошо. Функция-родитель может точно указать аргументы, необходимые от дочерних функций, а статически типизированный язык позволяет определить их типы возвращаемых значений, что обеспечивает возможность применения функциональности автозаполнения в интегрированной среде разработки.

Один из вопросов, связанных с аргументами функций, — как сделать их удобочитаемыми. Рассмотрим указание размера и скорости дисков. Естественный вызов `disk(150, 7200)` оказывается не слишком удобным, так как в нем нет никаких указаний, что означают эти цифры, если, конечно, ваш язык не использует в аргументах ключевые слова. С этим можно бороться с помощью “упаковки” данных в функцию, которая не делает ничего, кроме предоставления имени: `disk(size(150), speed(7200))`. В простейшем случае эта функция-упаковка просто возвращает значение аргумента, но это означает, что смысл функций может быть перепутан. Так, ничто не мешает получить очень большой и медленный диск с помощью вызова `disk(speed(7200), size(150))`. Этого можно избежать, заставляя вложенные функции возвращать промежуточные данные, такие как построитель или токен, хотя это и требует больших усилий по их настройке.

Необязательные аргументы также могут приводить к проблемам. Если базовый язык поддерживает для функций аргументы по умолчанию, можно воспользоваться ими. Если же такой возможности у вас нет, один из возможных обходных путей — определение различных функций для каждой комбинации необязательных аргументов. Если таких случаев лишь несколько, это утомительный, но разумный подход. По мере увеличения числа необязательных аргументов утомительность возрастает (а разумность уменьшается). Одним из путей выхода из этой проблемы вновь является использование промежуточных данных, и токены могут оказаться особенно эффективным выбором.

Если ваш язык поддерживает ассоциативные массивы литералов (`Literal Map (417)`), они могут оказаться хорошим средством выхода из сложной ситуации. В этом случае вы всегда получаете корректную структуру данных, и единственная проблема состоит в том, что С-подобные языки обычно не поддерживают эту возможность.

В случае нескольких аргументов в одном вызове наилучшим выбором является использование функций с переменным количеством параметров — если, конечно, их поддерживает базовый язык. Можно также рассматривать их как вложенный список литералов (`Literal List (415)`). Несколько аргументов различных типов в конечном счете приводят к тем же сложностям, что и необязательные аргументы.

Наихудший случай — это грамматика наподобие `parent ::= (this | that) *`. Здесь проблема в том, что если у нас нет возможности указания аргументов с ключевыми словами, то единственный способ их идентификации — позиция в вызове функции и тип. Это может помочь в ситуации, когда `this` и `that` имеют разные типы, но если их тип один и тот же, идентифицировать аргумент совершенно невозможно. В этом случае необходимо либо возвращаться к промежуточным результатам, либо прибегать к помощи переменной контекста. Применять переменную контекста особенно сложно, так как родительская функция не вычисляется до последнего момента, заставляя вас использовать для правильной установки этой переменной более широкий контекст языка.

Чтобы сохранить DSL удобочитаемым, обычно хочется, чтобы вложенные функции были неквалифицированными вызовами функций. Это означает, что следует либо сделать их глобальными функциями, либо прибегнуть к внесению области видимости в объект (*Object Scoping* (387)). Обычно глобальные функции означают глобальные проблемы, и я предпочитаю внесение области видимости в объект. Однако в случае вложенных функций применение глобальных функций зачастую может оказаться гораздо менее проблематичным, потому что самая большая проблема при использовании глобальных функций в том, что им сопутствует глобальное состояние синтаксического анализа. Глобальные же функции, просто возвращающие значение, такие как статический метод наподобие `DayOfWeek.MONDAY`, часто являются хорошим выбором.

34.2. Когда это использовать

Одной из самых сильных и одновременно слабых сторон вложенных функций является порядок вычисления. Аргументы функции вычисляются до вычисления самой функции (если вы не используете в качестве аргументов замыкания (*Closure* (397))). Это очень полезно для построения иерархии значений, потому что к моменту вызова родительской функции, которая создает объект семантической модели, у вас будут полностью сформированы все необходимые аргументы. Так можно избежать многих неприятностей с перемещением и промежуточными данными, которые сопутствуют применению последовательности функций (*Function Sequence* (357)) и соединению методов в цепочки (*Method Chaining* (375)).

С другой стороны, этот порядок вычислений приводит к проблемам, связанным с обратным порядком вычислений (вспомните пример с песенкой). Так что, если вы хотите иметь последовательность, которую надо читать слева направо, воспользуйтесь последовательностью функций или соединением методов. Для точного контроля за вычислением нескольких аргументов воспользуйтесь вложенными замыканиями (*Nested Closure* (403)).

Вложенным функциям часто приходится бороться с необязательными аргументами и переменным количеством аргументов. Вложенные функции ожидают, что вы точно укажете, что именно вы хотите и в каком именно порядке, поэтому, если вам нужна большая гибкость, имеет смысл рассмотреть связывание методов в цепочки или отображение литералов (*Literal Map* (417)). Отображение литералов часто оказывается хорошим выбором, поскольку позволяет отсортировать аргументы перед вызовом функции, предостав员я вам гибкость в упорядочении и решение вопроса с необязательными аргументами.

Еще одним недостатком вложенной функции (*Nested Function* (361)) является пунктуация, которая обычно требует применения парных скобок и размещения запятых в правильных местах. В результате вы можете получить нечто, напоминающее изуродованный Lisp. Впрочем, это не слишком большая проблема для DSL, предназначенных для программистов.

Коллизии имен в данном случае доставляют существенно меньше неприятностей, чем при использовании последовательностей функций, поскольку родительская функция предоставляет контекст для интерпретации вызова вложенной функции. В результате вы можете легко использовать функцию `speed` и для процессора, и для диска, и использовать одну и ту же функцию, лишь бы осуществлялась совместимость по типам.

34.3. Простой пример конфигурации компьютера (Java)

Рассмотрим пример кода для определения конфигурации простого компьютера.

```
computer(  
    processor(
```

```

cores(2),
speed(2500),
i386
),
disk(
size(150)
),
disk(
size(75),
speed(7200),
SATA
)
);

```

В этом случае каждое предложение сценария возвращает объект семантической модели (Semantic Model (171)), так что я могу использовать порядок вложенных вычислений для построения всего выражения без помощи переменных контекста (Context Variable (187)). Начнем с самого низа и рассмотрим предложение процессора.

```

class Builder...
static Processor processor(int cores, int speed,
                           Processor.Type type) {
    return new Processor(cores, speed, type);
}
static int cores(int value) {
    return value;
}
static final Processor.Type i386 = Processor.Type.i386;

```

Я определил элементы построителя как статические методы и константы класса построителя. С помощью функциональной возможности статического импорта Java я могу использовать в сценарии неквалифицированные вызовы.

Методы `cores` и `speed` представляют собой “синтаксический сахар”: все, для чего они нужны, — это повышение удобочитаемости.

Предложение `disk` имеет два необязательных аргумента. Поскольку их только два, я еще могу позволить себе написать соответствующие комбинации.

```

class Builder...
static Disk disk(int size, int speed,
                  Disk.Interface iface) {
    return new Disk(size, speed, iface);
}
static Disk disk(int size) {
    return disk(size, Disk.UNKNOWN_SPEED, null);
}
static Disk disk(int size, int speed) {
    return disk(size, speed, null);
}
static Disk disk(int size, Disk.Interface iface) {
    return disk(size, Disk.UNKNOWN_SPEED, iface);
}

```

В предложении верхнего уровня, определяющем конфигурацию компьютера, я использую функцию с переменным числом аргументов для обработки нескольких дисков.

```

class Builder...
static Computer computer(Processor p, Disk... d) {
    return new Computer(p, d);
}

```

Обычно я большой поклонник переноса в область видимости объекта (*Object Scoping* (387)) для устранения беспорядка в коде из-за применения глобальных функций и переменных контекста. Однако в случае статического импорта и вложенных функций я могу использовать статические элементы, не внося в программу “глобального мусора”.

34.4. Обработка различных аргументов с помощью токенов (C#)

Использовать вложенные функции сложнее, когда имеется несколько аргументов различных видов. Рассмотрим язык для определения окна на экране.

```
box(
    topBorder(2),
    bottomBorder(2),
    leftMargin(3),
    transparent
);
box(
    leftMargin(2),
    rightMargin(5)
);
```

В этой ситуации может быть любое количество устанавливаемых свойств. Нет никакой необходимости придерживаться точного порядка в объявлении свойств, так что обычный стиль идентификации аргументов в C# (по позиции) работает не слишком хорошо. В данном примере рассматривается применение токенов для идентификации аргументов для их объединения в структуре.

Вот как выглядит объект целевой модели.

```
class Box {
    public bool IsTransparent = false;
    public int[] Borders = {1,1,1,1}; // Вверху справа снизу слева
    public int[] Margins = {0,0,0,0}; // Вверху справа снизу слева
```

Различные функции возвращают в качестве типа данных токен и имеют следующий вид.

```
class BoxToken {
    public enum Types { TopBorder, BottomBorder,
                      LeftMargin, RightMargin,
                      Transparent }
    public readonly Types Type;
    public readonly Object Value;
    public BoxToken(Types type, Object value) {
        Type = type;
        Value = value;
    }
}
```

Я использую перенос в область видимости объекта (*Object Scoping* (387)) и определяю предложения DSL как функции надтипа построителя.

```
class Builder...
protected BoxToken topBorder(int arg) {
    return new BoxToken(BoxToken.Types.TopBorder, arg);
}
protected BoxToken transparent {
    get {
        return new BoxToken(BoxToken.Types.Transparent, true);
    }
}
```

Я показал только пару функций, но уверен, что этого достаточно. Вы без труда сможете сами определить, как выглядят остальные функции.

Теперь родительская функция просто проходит по результатам аргументов и собирает окно.

```
class Builder...
protected void box(params BoxToken[] args) {
    Box newBox = new Box();
    foreach (BoxToken t in args) updateAttribute(newBox, t);
    boxes.Add(newBox);
}

List<Box> boxes = new List<Box>();

private void updateAttribute(Box box, BoxToken token) {
    switch (token.Type) {
        case BoxToken.Types.TopBorder:
            box.Borders[0] = (int)token.Value;
            break;
        case BoxToken.Types.BottomBorder:
            box.Borders[2] = (int)token.Value;
            break;
        case BoxToken.Types.LeftMargin:
            box.Margins[3] = (int)token.Value;
            break;
        case BoxToken.Types.RightMargin:
            box.Margins[1] = (int)token.Value;
            break;
        case BoxToken.Types.Transparent:
            box.Transparent = (bool)token.Value;
            break;
        default:
            throw new InvalidOperationException("Unreachable");
    }
}
```

34.5. Использование токенов подтипов для поддержки интегрированной среды разработки (Java)

Большинство языков различают аргументы функции по их позициям. Так что в приведенном выше примере мы могли бы устанавливать размер и скорость дисков с помощью вызовов функции наподобие `disk(150, 7200)`. Такой вызов не слишком удобочитаем, поэтому в приведенном выше примере я обернул числа в простые функции и получил более удобочитаемый вызов `disk(size(150), speed(7200))`. В рассмотренном ранее примере кода эти функции просто возвращали свои аргументы, что способствовало улучшению удобочитаемости, но не мешало ошибиться и ввести `disk(speed(7200), size(150))`.

Использование простых токенов, как в рассмотренном выше примере с определением окна, обеспечивает механизм проверки ошибок. При возвращении токена `[size, 150]` вы можете использовать тип токена для проверки того, что у вас имеется верный аргумент в правильной позиции, или даже заставить аргументы работать в любом порядке.

Проверка — это всегда хорошо, но в статически типизированном языке с современной интегрированной средой разработки можно пойти дальше. Представим, что вы хотите, чтобы функция автозаполнения в интегрированной среде разработки заставляла вас сначала указывать размер, а затем — скорость. Это можно осуществить с помощью подклассов.

В рассмотренных выше токенах тип токена представлял собой свойство токена. Альтернативой является создание различных подтипов для каждого токена; затем я могу использовать подтипы в определении родительской функции.

Бот краткий сценарий, который я хотел бы иметь возможность поддерживать.

```
disk(
    size(150),
    speed(7200)
);
```

А вот объект целевой модели.

```
public class Disk {
    private int size, speed;
    public Disk(int size, int speed) {
        this.size = size;
        this.speed = speed;
    }
    public int getSize() {
        return size;
    }
    public int getSpeed() {
        return speed;
    }
}
```

Для обработки размера и скорости я создаю общий целочисленный токен с подклассами для предложений двух видов.

```
public class IntegerToken {
    private final int value;
    public IntegerToken(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}

public class SpeedToken extends IntegerToken {
    public SpeedToken(int value) {
        super(value);
    }
}

public class SizeToken extends IntegerToken {
    public SizeToken(int value) {
        super(value);
    }
}
```

Теперь я могу определить в построителе статические функции с корректными аргументами.

```
class Builder...
    public static Disk disk(SizeToken size, SpeedToken speed) {
        return new Disk(size.getValue(), speed.getValue());
    }
    public static SizeToken size (int arg) {
        return new SizeToken(arg);
    }
}
```

```
public static SpeedToken speed (int arg) {
    return new SpeedToken(arg);
}
```

Теперь интегрированная среда разработки будет предлагать правильные функции в правильном месте, так что вероятность неверного ввода существенно уменьшится.

(Еще один способ добавления статической типизации состоит в применении шаблонов Java (generics), но это я оставляю читателям для самостоятельного рассмотрения.)

34.6. Использование инициализаторов объектов (C#)

В C# наиболее естественным способом обработки иерархии данных является применение инициализаторов объектов.

```
new Computer() {
    Processor = new Processor() {
        Cores = 2,
        Speed = 2500,
        Type = ProcessorType.i386
    },
    Disks = new List<Disk>() {
        new Disk() {
            Size = 150
        },
        new Disk() {
            Size = 75,
            Speed = 7200,
            Type = DiskType.SATA
        }
    };
};
```

Такой способ может работать с простым набором классов модели.

```
class Computer {
    public Processor Processor { get; set; }
    public List<Disk> Disks { get; set; }
}

class Processor {
    public int Cores { get; set; }
    public int Speed { get; set; }
    public ProcessorType Type { get; set; }
}
public enum ProcessorType {i386, amd64}

class Disk {
    public int Speed { get; set; }
    public int Size { get; set; }
    public DiskType Type { get; set; }
}
public enum DiskType {SATA, IDE}
```

Инициализаторы объектов можно рассматривать как вложенные функции, которые принимают аргументы с ключевыми словами (подобно отображениям литералов ([Literal Map \(417\)](#))), но ограничены созданием объектов. Их нельзя использовать в произвольных местах, но в ситуациях наподобие показанной выше они могут оказаться очень удобными.

34.7. Повторяющиеся события (C#)

Я жил в южной части Бостона. Эта жизнь почти ничем не отличалась от жизни в центральной части города, по крайней мере в том, что касалось мест, где можно без труда потратить время и деньги. Однако был один очень раздражавший меня момент — уборка улиц. В первый и третий понедельники с апреля по октябрь убиралась улица, на которой я жил, и я не должен был парковать там свой автомобиль. Но я очень часто забывал об этом и получал уведомления о штрафе.

Итак, правило уборки моей улицы — первый и третий понедельники с апреля по октябрь. Вот как можно было бы записать соответствующее выражение DSL.

```
Schedule.First(DayOfWeek.Monday)
    .And(Schedule.Third(DayOfWeek.Monday))
    .From(Month.April)
    .Till(Month.October);
```

В этом примере с вложенными функциями комбинируется соединение методов в цепочки (Method Chaining (375)). Обычно при использовании вложенных функций я предпочитаю применять перенос области видимости в объект (Object Scoping (387)), но в данном случае вкладываемые функции просто возвращают значение, так что наущной потребности в таком переносе здесь я не ощущаю.

34.7.1. Семантическая модель

Повторяющиеся события — это достаточно часто повторяющееся событие в программах. Зачастую требуется расписание, построенное на определенной комбинации дат. Я рассматриваю их как *спецификацию* (Specification [7]) дат. Нам нужен код, который бы сообщал, включена ли некоторая дата в расписание. Это делается путем определения общего интерфейса спецификации, который мы можем сделать обобщенным, так как спецификации полезны во многих ситуациях.

```
internal interface Specification<T> {
    bool Includes(T arg);
}
```

При построении модели спецификации для конкретного типа я предполагаю иметь небольшие “кирпичики”, которые можно соединять. Одним таким кирпичиком является понятие определенного периода года, например с апреля по октябрь.

```
internal class PeriodInYear : Specification<DateTime>
{
    private readonly int startMonth;
    private readonly int endMonth;

    public PeriodInYear(int startMonth, int endMonth) {
        this.startMonth = startMonth;
        this.endMonth = endMonth;
    }
    public bool Includes(DateTime arg) {
        return arg.Month >= startMonth && arg.Month <= endMonth;
    }
}
```

Другим элементом является понятие первого понедельника месяца. Этот класс немного сложнее, так как требуется проход по нескольким датам месяца, чтобы выяснить, какая из них нас интересует.

```

internal class DayInMonth : Specification<DateTime> {
    private readonly int index;
    private readonly DayOfWeek dayOfWeek;

    public DayInMonth(int index, DayOfWeek dayOfWeek) {
        this.index = index;
        this.dayOfWeek = dayOfWeek;
        if (index <= 0)
            throw new NotSupportedException(
                "Индекс должен быть положительным");
    }

    public bool Includes(DateTime arg) {
        int currentMatch = 0;
        foreach(DateTime d in
            new MonthEnumerator(arg.Month, arg.Year)) {
            if (d > arg) return false;
            if (d.DayOfWeek == dayOfWeek) {
                currentMatch++;
                if (currentMatch == index) return (d == arg);
            }
        }
        return false;
    }
}

```

Для прохода по дням месяца данная спецификация использует специальный перечислитель, для которого указываются конкретные месяц и год.

```

internal class MonthEnumerator
    : IEnumerator<DateTime>, IEnumerable<DateTime> {
    private int year;
    private Month month;

    public MonthEnumerator(int month, int year) {
        this.month = new Month(month);
        this.year = year;
        Reset();
    }
}

```

Он реализует методы `IEnumerable`.

```

class MonthEnumerator...
private DateTime current;
DateTime IEnumerator<DateTime>.Current {
    get { return current; }
}
public object Current { get { return current; } }

public void Reset() {
    current =
        new DateTime(year, month.Number, 1).AddDays(-1);
}

public void Dispose() {}

public bool MoveNext() {
    current = current.AddDays(1);
    return month.Includes(current);
}

```

Кроме того, чтобы класс можно было применять в цикле `foreach`, реализованы методы `IEnumerable`.

```

class MonthEnumerator...
IEnumerator<DateTime>
    IEnumerable<DateTime>.GetEnumerator() {
        return this;
    }
public IEnumerator GetEnumerator() {
    return this;
}

```

Наконец, у нас имеется очень простой класс Month, который также действует как спецификация.

```

class Month...
private readonly int number;
public int Number { get { return number; } }
public Month(int number) {
    this.number = number;
}
public bool Includes(DateTime arg) {
    return number == arg.Month;
}

```

Все это весьма полезные кирпичики, но сами по себе они не способны даже упасть кому-то на голову. Чтобы заставить их сделать что-то полезное, нужно иметь возможность объединять их в логические выражения, чего я добиваюсь с помощью еще пары спецификаций.

```

abstract class CompositeSpecification<T>
    : Specification<T> {
protected IList<Specification<T>> elements =
    new List<Specification<T>>();
public CompositeSpecification(
    params Specification<T>[] elements) {
    this.elements = elements;
}
public abstract bool Includes(T arg);
}

internal class AndSpecification<T>
    : CompositeSpecification<T> {
public AndSpecification(params Specification<T>[] elements)
    : base(elements) {}
public override bool Includes(T arg) {
    foreach (Specification<T> s in elements)
        if (! s.Includes(arg)) return false;
    return true;
}
}

internal class OrSpecification<T>
    : CompositeSpecification<T> {
public OrSpecification(params Specification<T>[] elements)
    : base(elements) {}
public override bool Includes(T arg) {
    foreach (Specification<T> s in elements)
        if (s.Includes(arg)) return true;
    return false;
}
}

```

Я думаю, что вы и сами сумеете реализовать NotSpecification.

Есть одна вещь, которая не нравится мне в данной модели. Это необходимость применять класс `DateTime`. Дело в том, что точность `DateTime` — доли секунды, а я работаю только на уровне дней. Использование типов данных для работы со временем, обеспечивающих более высокую, чем требуется, точность, — явление достаточно распространенное, поскольку обычно к нему нас подталкивают библиотеки. Однако это может привести к неприятной ошибке при сравнении двух `DateTime`, которые отличаются друг от друга, но меньше текущего уровня точности (т.е. с точки зрения конкретной задачи одинаковы). В реальном проекте я бы создал собственный класс `Date` с необходимым уровнем точности.

34.7.2. DSL

Вот текст DSL для расписания уборки моей улицы.

```
Schedule.First(DayOfWeek.Monday)
    .And(Schedule.Third(DayOfWeek.Monday))
    .From(Month.April)
    .Till(Month.October);
```

Подобно другим реалистичным DSL, наш предметно-ориентированный язык использует комбинацию технологий внутренних DSL, а именно — вложенных функций с соединением методов в цепочки (*Method Chaining* (375)). Однако сейчас меня интересует только способ применения вложенных функций. Поскольку каждая вложенная функция возвращает простое значение, у меня нет острой необходимости переносить область видимости в объект (*Object Scoping* (387)), так же как и использовать переменные контекста (*Context Variable* (187)). В результате я использую статические методы. Так как я работаю с C#, все статические методы используют в качестве префикса их имя класса. Это вполне удобочитаемо, хотя и привносит определенный шум по сравнению с переносом области видимости в объект.

Две из вложенных функций представляют собой вызовы, возвращающие простые значения. `DayOfWeek.Monday` на самом деле встроена в библиотеки .NET. `Month.April` и прочие я добавляю самостоятельно.

```
class Month...
    public static readonly Month January = new Month(1);
    public static readonly Month February = new Month(2);
    public static readonly Month March = new Month(3);
    public static readonly Month April = new Month(4);
    public static readonly Month May = new Month(5);
    // Думаю, продолжать не нужно?
```

Вызовы `Schedule` немного отличаются. Начальное применение `Schedule.First` представляет собой пример распространенной особенности таких языков — применение обычной функции для создания стартового объекта для соединения методов в цепочку. Здесь `Schedule` выступает в роли построителя выражений (*Expression Builder* (349)). Я не именую его с помощью слова `builder`, поскольку, как мне кажется, просто `Schedule` проще и прочесть, и понять.

```
class Schedule...
    public static Schedule First(DayOfWeek dayOfWeek) {
        return new Schedule(new DayInMonth(1, dayOfWeek));
    }
```

Подобно большинству построителей выражений, `Schedule` создает содержимое, являющееся спецификацией.

```
class Schedule...
    private Specification<DateTime> content;
    public Specification<DateTime>
        Content { get { return content; } }
    public Schedule(Specification<DateTime> content) {
        this.content = content;
    }
```

Обратите внимание на то, как начальный вызов возвращает расписание, которое “заворачивает” первый элемент в спецификацию. Точно таков же и последующий вызов `Third` (за исключением параметра). Я обычно выступаю против написания различных методов для того, что можно передать в качестве параметра, но это еще один пример изменения правил хорошего тона в программировании при использовании построителя выражений.

Композиционная структура создается с помощью соединения методов в цепочки. Взгляните на интересно названный метод `And`.

```
class Schedule...
    public Schedule And(Schedule arg) {
        content = new OrSpecification<DateTime>(content, arg.content);
        return this;
    }
```

В разговорной речи мы говорим “первый *и* третий понедельники”, но в терминах спецификации логическое условие говорит о первом *или* третьем понедельнике. Это интересный пример противоречия DSL модели для того, чтобы обеспечить естественное прочтение исходного текста.

Период указывается с помощью соединения методов в цепочки.

```
class Schedule...
    public Schedule From(Month m) {
        Debug.Assert(null == periodStart);
        periodStart = m;
        return this;
    }
    public Schedule Till(Month m) {
        Debug.Assert(null != periodStart);
        PeriodInYear period =
            new PeriodInYear(periodStart.Number, m.Number);
        content =
            new AndSpecification<DateTime>(content, period);
        return this;
    }
    private Month periodStart;
```

Здесь для корректного построения периода используется переменная контекста.

В данном примере в качестве вложенных функций использованы простые статические методы. Получим ли мы выгоду от устранения имен классов? Мне кажется, что `Monday` читается лучше, чем `DayOfWeek.Monday`. Эта возможность предоставляется благодаря переносу области видимости в объект ценой использования наследования. В Java можно прибегнуть к статическому импорту. Получаемый выигрыш невелик, но, пожалуй, стоит затраченных усилий.

Глава 35

Соединение методов в цепочки

Method Chaining

Методы-модификаторы возвращают базовый объект, так что в одном выражении может быть вызвано несколько таких методов

```
computer()
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size(150)
    .disk()
        .size(75)
        .speed(7200)
        .sata()
.end();
```

35.1. Как это работает

Соединение методов в цепочки быстро распространилось среди программистов как пример того, как должен выглядеть внутренний DSL. Это распространение оказалось слишком эффективным: многие стали считать, что соединение методов в цепочки — это синоним свободных интерфейсов и внутренних DSL. Но на мой взгляд, соединение методов в цепочки — хотя и важный и заметный, но только один из множества методов, применяемых во внутренних DSL.

Его распространенной формой является построитель выражений (Expression Builder (349)). Рассмотрим представленный в начале главы набросок жесткого диска. Используя обычный API командных запросов, его можно настроить следующим образом.

```
//java...
HardDrive hd = new HardDrive();
hd.setCapacity(150);
hd.setExternal(true);
hd.setSpeed(7200);
```

Я создаю объект, помещаю его в переменную и применяю к нему функции установки значений его свойств. Если, как в приведенном примере, таких параметров всего три,

я бы предпочел воспользоваться конструктором, но предположим, что их у нас гораздо больше. DSL часто применяется для настройки объектов, выполнять которую в конструкторах достаточно сложно. Кроме того, такой исходный текст трудно читать, поскольку конструкторы допускают в основном только позиционные параметры.

Применив соединения методов в цепочки, я могу получить что-то наподобие следующего.

```
new HardDrive().capacity(150).external().speed(7200);
```

Чтобы соединение методов в цепочки работало, эти методы должны быть реализованы иначе, не так, как обычно реализуются методы установки значений. В Java мы обычно реализуем метод установки значения следующим образом.

```
public void setSpeed(int arg) {
    this.speed = arg;
}
```

Но метод, разрабатываемый для применения в соединении методов в цепочки, для продолжения цепочки должен возвращать объект. Для данного построителя он должен возвращать самого себя.

```
private HardDrive speed(int arg) {
    speed = arg;
    return this;
}
```

Возвращаемое модифицирующим методом значение нарушает принцип разделения команд и запросов (с. 87). В большинстве случаев я следую этому принципу, и он хорошо мне служит. Свободный интерфейс — практически единственный случай, когда мы должны нарушить этот принцип.

Второе следствие применения соединения методов в цепочки — именование методов. В множестве соглашений об именовании метод наподобие `sata()` выглядит запросом, а не модификатором. Такое именование весьма проблематично, так как может сбить с толку любого, кто ожидает применения API командных запросов. Технология соединения методов в цепочки нарушает много обычных правил проектирования обычного API (командных запросов).

Соединение методов в цепочки приводит к изменению не только правил проектирования API, но и форматирования. Обычно мы пытаемся располагать множественные вызовы методов в одной строке, но длинные цепочки часто выглядят плохо при таком форматировании, особенно если нужно подчеркнуть иерархию. В результате зачастую лучше форматировать код соединения методов в цепочки по одному вызову в строке.

```
new HardDrive()
    .capacity(150)
    .external()
    .speed(7200);
```

И Java, и C# игнорируют большинство символов новой строки, что обеспечивает нас высокой гибкостью форматирования. Обычно предпочтение отдается размещению точки в начале строки, так как это делает ее очень заметной и подчеркивает применение соединения методов в цепочки. Языки, использующие символы новой строки в качестве разделителей инструкций, оказываются существенно менее гибкими. Ruby, например, будет работать, но точку при этом следует располагать в конце, а не в начале строки. Размещение методов по одному в строке, кроме того, облегчает отладку, так как обычно сообще-

ния об ошибках и управление отладчиком используют номера строк. Таким образом, вполне разумно ограничиваться в каждой строке минимумом работы.

35.1.1. Построители или значения

Выше я показал пример применения соединения методов в цепочки для построителя выражений (Expression Builder (349)). Я предпочитаю использовать соединение методов в цепочки и другие свободные API для построителей выражений, поскольку это уменьшает путаницу между соглашениями свободного API и API командных запросов.

Однако иногда может оказаться полезным применение соединения методов в цепочки за пределами построителей выражений, например в чем-то наподобие `42.grams.flour`. В этом случае мы создаем выражение посредством последовательности *объектов значений* (Value Objects [10]). Метод `grams` определен для целых чисел (с использованием расширения литералов (Literal Extension (473))) и возвращает объект количества, который, в свою очередь, является базой для метода `flour`, возвращающего соответствующий ингредиент⁵. Вместо одного построителя выражений у нас получается последовательность обычных объектов. Часто в подобных случаях объекты являются объектами значений.

На каждом шаге выражения мы видим преобразование в новый тип — явление, которое мой коллега Нил Форд (Neal Ford) называет **метаморфозами типа** (type transmogrification). (Я просто обязан упомянуть этот термин, иначе Нил обидится и больше никогда не уостит меня своим прекрасным чаем.)

Есть много хороших разработчиков, которым очень нравится подобное применение соединения методов в цепочки, так что я должен возражать против него очень осторожно. Тем не менее я склоняюсь, насколько это возможно, к применению построителей выражений, чтобы четко разделять стили свободного API и API командных запросов.

35.1.2. Проблема окончания

Проблема окончания — обычный вопрос при соединении методов в цепочки. Она сводится к отсутствию явно указанной конечной точки цепочки методов. Представим построитель для назначений, который допускает выражения наподобие следующих.

```
//C#...
var dentist = new AppointmentBuilder()
    .From(1300)
    .To(1400)
    .For("dentist")
;
var dinner = new AppointmentBuilder()
    .From(1900)
    .To(2100)
    .For("dinner")
    .At("Turners")
;
```

Мне хотелось бы, чтобы возвращаемое значение было объектом `Appointment`, поскольку это наиболее естественное решение. Однако для того, чтобы работало соединение методов в цепочки, каждый метод должен возвращать построитель. Ничто в цепочке не говорит о том, что работа завершена, так что я должен добавить метод-маркер некоторого вида, чтобы показать, что работа завершена.

⁵ Grams — граммы, flour — мука (англ.). — Примеч. пер.

```
Appointment dentist = new AppointmentBuilder()
    .From(1300)
    .To(1400)
    .For("dentist")
    .End
;
```

Это не так уж и страшно, тем не менее применение `End` вносит небольшой синтаксический шум. Вот почему важной альтернативой является использование вложенных функций (`Nested Function` (361)) или вложенных замыканий (`Nested Closure` (403)). В C# их можно избежать с помощью неявного оператора преобразования типа, хотя это означает, что вы не сможете воспользоваться `var` для описания.

35.1.3. Иерархическая структура

С проблемой завершения тесно связана проблема, заключающаяся в том, что соединение методов в цепочки не имеет явной иерархической структуры. Иерархические структуры — распространенное явление в языках программирования; именно поэтому получили такое распространение синтаксические деревья. Рассмотрим уже знакомый нам пример.

```
computer()
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size(150)
    .disk()
        .size(75)
        .speed(7200)
        .sata()
.end();
```

Здесь, определенно, имеется иерархия, но она выражена только форматированием кода, но никак не его структурой. В результате мы должны управлять этой структурой самостоятельно. Такая же проблема возникает и при использовании последовательности функций (`Function Sequence` (357)).

Хорошим примером такого управления служит проверка, работаем ли мы с корректным диском при вызове метода наподобие `size`. Для этого применяется пара подходов. Один из них — использование переменной контекста (`Context Variable` (187)), такой как `currentDisk`. Всякий раз, когда вызывается метод `disk`, выполняется обновление переменной контекста. Мы можем также поддерживать список дисков и всякий раз выполнять обновление последнего диска в списке.

Часто оказывается полезным иметь новый дочерний построитель диска. Отдельный построитель позволяет ограничить доступные методы только теми, которые требуются для предоставления информации о диске, и методом завершения.

35.1.4. Последовательные интерфейсы

Важным вариантом базового соединения методов в цепочки является применение нескольких интерфейсов для управления фиксированной последовательностью методов в цепочке. Рассмотрим создание сообщения электронной почты. Мы хотим, чтобы программист сначала определил адрес назначения, адреса копий, тему, а затем — тело сообще-

щения. Мы можем сделать это, предоставив последовательность интерфейсов построителю выражений (*Expression Builder* (349)). Первый интерфейс имеет только метод `to`. Метод `to` возвращает интерфейс, имеющий только методы `cc`, `ss` и `subject`. Метод `cc` возвращает интерфейс, у которого имеются только методы `cc` и `subject`. Метод `subject` возвращает интерфейс, у которого имеется только один метод — `body`.

Такой подход хорошо работает в статически типизированных языках с поддержкой интегрированных сред разработки. Функциональная возможность автозаполнения интегрированной среды разработки может помочь программисту написать полное предложение DSL, предлагая только те методы, которые корректны в данной точке цепочки.

Возможность управления тем, какие методы являются корректными в том или ином контексте, аналогична тому, что вы получаете с помощью дочернего построителя. В самом деле, вы можете применять дочерний построитель для выполнения тех же действий, для которых применяете последовательные интерфейсы, но последние проще, так что лучше использовать именно их (если, конечно, у вас нет каких-то особых причин прибегнуть именно к дочерним построителям).

Последовательные интерфейсы могут применяться и для того, чтобы обеспечить наличие в цепочке обязательных элементов; для этого определите интерфейс, который принимает единственный обязательный элемент.

35.2. Когда это использовать

Соединение методов в цепочки может существенно повысить удобочитаемость внутренних DSL, а потому в некоторых умах оно превратилось в синонимом внутренних DSL. Однако его лучше применять в сочетании с другими комбинациями функций.

Соединение методов в цепочки эффективнее всего работает при использовании в языке необязательных предложений. Оно позволяет пишущему сценарий DSL легко выбирать предложения, необходимые в данной конкретной ситуации. В языке программирования сложно указать, что некоторые предложения должны присутствовать в обязательном порядке. Использование последовательных интерфейсов обеспечивает некоторое упорядочение предложений, но все равно конечные предложения всегда могут быть опущены без возражений со стороны языка. В случае обязательных предложений наилучшим выбором являются вложенные функции (*Nested Function* (361)).

Время от времени возникает проблема завершающего вызова. Хотя для ее решения имеются некоторые обходные пути, как правило, при столкновении с ней лучше использовать вложенные функции или вложенные замыкания (*Nested Closure* (403)). Эти же альтернативы представляют собой наилучший выбор, если вы сталкиваетесь с беспорядком, вызванным применением переменных контекста (*Context Variable* (187)).

35.3. Простой пример конфигурации компьютера (Java)

Рассмотрим пример базовой конфигурации компьютера с применением соединения методов в цепочки.

```
computer()
  .processor()
    .cores(2)
    .speed(2500)
    .i386()
  .disk()
    .size(150)
```

```
.disk()
.size(75)
.speed(7200)
.sata()
.end();
```

Для начала выражения с применением соединения методов в цепочки необходим вызов некоторого метода, инициирующего цепочку. В данном случае использован статический метод, который можно вызвать в сценарии DSL благодаря статическому импорту.

```
public static ComputerBuilder computer() {
    return new ComputerBuilder();
}
```

Здесь использован построитель компьютера для определения различных методов, требующихся в цепочке. Он также содержит данные синтаксического анализа.

В случае процессора я храню построитель конструкции (*Construction Builder* (191)) для текущего процессора в переменной контекста (*Context Variable* (187)).

```
class ComputerBuilder...
public ComputerBuilder processor() {
    currentProcessor = new ProcessorBuilder();
    return this;
}
private ProcessorBuilder currentProcessor;

public ComputerBuilder cores(int arg) {
    currentProcessor.cores = arg;
    return this;
}
public ComputerBuilder i386() {
    currentProcessor.type = Processor.Type.i386;
    return this;
}

class ProcessorBuilder {
    private static final int DEFAULT_CORES = 1;
    private static final int DEFAULT_SPEED = -1;

    int cores = DEFAULT_CORES;
    int speed = DEFAULT_SPEED;
    Processor.Type type;
    Processor getValue() {
        return new Processor(cores, speed, type);
    }
}
```

Постройтель в каждом вызове возвращает самого себя, чтобы цепочка могла продолжаться, что является неотъемлемой характеристикой соединения методов в цепочки.

Определение дисков немного сложнее, поскольку каждый диск имеет собственные данные. Я могу определить больше переменных контекста в построителе компьютера, так же, как для процессора, но в данном случае я использую отдельный построитель для сброса атрибутов диска.

```
class DiskBuilder...
public DiskBuilder size(int arg) {
    size = arg;
    return this;
}
public DiskBuilder speed(int arg) {
```

```

    speed = arg;
    return this;
}
public DiskBuilder sata() {
    iface = Disk.Interface.SATA;
    return this;
}
}

```

Определенная сложность заключается в смешивании построителя компьютера с построителем диска и в поддержке переменных контекста. Вызов `disk` вводит новый диск, так что построитель компьютера помещает в переменную контекста новый построитель диска и перенаправляет вызовы ему.

```

class ComputerBuilder...
public DiskBuilder disk() {
    if (currentDisk != null)
        loadedDisks.add(currentDisk.getValue());
    currentDisk = new DiskBuilder(this);
    return currentDisk;
}
private DiskBuilder currentDisk;
private List<Disk> loadedDisks = new ArrayList<Disk>();

class DiskBuilder...
public DiskBuilder(ComputerBuilder parent) {
    this.parent = parent;
}
private int size = Disk.UNKNOWN_SIZE;
private int speed = Disk.UNKNOWN_SPEED;
private Disk.Interface iface;
private ComputerBuilder parent;

```

Вызов `disk` может встречаться и между дисками. Поэтому я добавляю текущий диск в список загруженных дисков перед тем, как создавать новый построитель. Такой вызов будет получен построителем диска, поэтому я просто передаю его построителю компьютера.

```

class DiskBuilder...
public DiskBuilder disk() {
    return parent.disk();
}

```

В этом примере я сталкиваюсь с проблемой завершения. Для ее решения я прибегаю к простейшему обходному пути: методу `end`. Как и в случае с дисками, метод `end` может появляться как вызов построителя дисков, так что я передаю построителю компьютера и его.

```

class DiskBuilder...
public Computer end() {
    return parent.end();
}

```

В построителе компьютера метод `end` применяется для того, чтобы создать и вернуть сконфигурированный компьютер.

```

class ComputerBuilder...
public Computer end() {
    return getValue();
}

public Computer getValue() {
    return new Computer(currentProcessor.getValue(),
                        disks());
}

```

```

    }

private Disk[] disks() {
    List<Disk> result = new ArrayList<Disk>();
    result.addAll(loaderDisks);
    if (currentDisk != null)
        result.add(currentDisk.getValue());
    return result.toArray(new Disk[result.size()]);
}

public ComputerBuilder speed(int arg) {
    currentProcessor.speed = arg;
    return this;
}
}

```

Это позволяет использовать построитель следующим образом.

```

Computer c = ComputerBuilder
    .computer()
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size(150)
    .disk()
        .size(75)
        .speed(7200)
        .sata()
.end();

```

В противном случае придется написать нечто наподобие следующего.

```

ComputerBuilder builder = new ComputerBuilder();
builder
    .processor()
        .cores(2)
        .speed(2500)
        .i386()
    .disk()
        .size(150)
    .disk()
        .size(75)
        .speed(7200)
        .sata();
Computer c = builder.getValue();

```

В этом примере я был непоследователен в применении вспомогательных построителей для процессора и дисков. Построитель процессора — простой построитель конструкции, использующийся просто для хранения промежуточных значений. В случае построителя диска я делегирую ему свободные методы. Простой построитель конструкции лучше работает в простых случаях, а полное делегирование — в более сложных. Здесь в педагогических целях показаны оба варианта, хотя я в большей степени склоняюсь к полному делегированию.

Этот пример достаточно хорошо иллюстрирует многие вопросы применения соединения методов в цепочки, особенно по сравнению с применением вложенных функций (*Nested Function* (361)). Соединение методов в цепочки повышает удобочитаемость исходного текста без особого синтаксического шума, который характерен для вложенных

функций. Однако, чтобы это осуществить, нужно сделать много мелкой работы с переменными контекста и справиться с проблемой завершения.

35.4. Соединение методов в цепочки со свойствами (C#)

C# и Java — похожие языки программирования, так что множество комментариев, относящихся к Java, применимы и к C#. Наибольшее отличие в том, что C# имеет специализированный синтаксис свойств, в то время как Java работает с методами получения и установки значений. В результате обычный текст должен выглядеть примерно так.

```
HardDrive hd = new HardDrive() {  
    Size = 150,  
    Type = HardDriveType.SATA,  
    Speed = 7200  
};
```

Соединение методов в цепочки имеет почти такой же вид.

```
new HardDriveBuilder()  
.Size(150)  
.SATA  
.Speed(7200);
```

Модификаторы скорости и размера в цепочке идентичны (за исключением соглашений об употреблении прописных букв в именовании). Однако имеется одно интересное отличие в обработке внешнего свойства. С помощью метода получения значения для внешних свойств можно избавиться от ненужных и раздражающих скобок. Я реализую этот метод следующим образом.

```
private HardDriveBuilder SATA {  
    get {  
        type = HardDriveType.SATA;  
        return this;  
    }  
}
```

Этот код должен заставить вас в некоторой степени прийти в изумление: это метод получения значения свойства, но ведет себя, как метод установки значения, возвращая сам объект, а не значение свойства. Это противоречит всем представлениям о том, как должны работать методы получения свойств. Почти во всех ситуациях я называл бы сделанное чрезвычайно плохим кодом. Он приемлем в единственном случае, а именно — при размещении в свободном контексте. И вновь я бы предпочел надежно запереть эту мерзость в построителе выражений (Expression Builder (349)).

35.5. Последовательные интерфейсы (C#)

Автозаполнение — одна из приятных особенностей современных интегрированных сред разработки. При этом не нужно запоминать все возможные вызываемые методы того или иного класса — достаточно нажать соответствующую комбинацию клавиш и тут же получить их в виде меню. Так как мой мозг переполнился информацией еще лет пятнадцать назад, я очень рад, что теперь мне нужно запоминать гораздо меньше.

Многие DSL имеют определенный порядок построения тех или иных вещей. Мы сможем воспользоваться автозаполнением, если применим последовательные интерфейсы. Предположим, необходимо создавать сообщения электронной почты.

```
message = MessageBuilder.Build()
    .To("fowler@acm.org")
    .Cc("editor@publisher.com")
    .Subject("error in book")
    .Body("Sally Shipton should read Sally Sparrow");
```

Нам нужно гарантировать построение элементов сообщения в определенном порядке: адрес назначения, адрес копий, тема и тело сообщения. При простом соединении методов в цепочки нет никакой силы, которая бы заставляла выполнять вызовы в некотором определенном порядке.

Отличной приправой к пресному соединению методов в цепочки в данном случае может стать использование нескольких интерфейсов с построителем выражений (Expression Builder (349)). Начнем с вызова Build.

```
public static IMessengerBuilderPostBuild Build() {
    return new MessengerBuilder();
}

interface IMessengerBuilderPostBuild {
    IMessengerBuilderPostTo To(String arg);
}
```

Я возвращаю построитель выражений так, как обычно, но возвращаемый тип представляет собой специальный интерфейс, который допускает только разрешенный шаг последовательности. Постройтель выражений реализует этот интерфейс, и теперь следующим может быть только данный конкретный вызов. В качестве небольшой премии меню автозаполнения будет включать только допустимые на очередном шаге вызовы (правда, вместе с методами, унаследованными от Object). Такое автозаполнение будет доступно в любой точке нашего процесса.

Следующий шаг продолжает наш рассказ.

```
public IMessengerBuilderPostTo To(String arg) {
    Content.To.Add(new Email(arg));
    return this;
}

interface IMessengerBuilderPostTo : IMessengerBuilderPostBuild {
    IMessengerBuilderPostCc Cc(String arg);
    IMessengerBuilderPostSubject Subject(String arg);
}
```

Новое здесь в том, что корректные шаги после To включают корректные шаги после Build. Я могу указать это без дублирования тела IMessengerBuilderPostBuild, наследуя интерфейсы. Этот способ вряд ли стоит того, чтобы применять его в данном конкретном примере, но зачастую он весьма полезен.

Оставшаяся часть последовательности продолжается так, как вы, вероятно, и ожидаете.

```
public IMessengerBuilderPostCc Cc(String arg) {
    Content.Cc.Add(new Email(arg));
    return this;
}
public IMessengerBuilderPostSubject Subject(String arg) {
    Content.Subject = arg;
    return this;
}
public Message Body(String arg) {
    Content.Body = arg;
    return Content;
}
```

```
interface IMessageBuilderPostCc
{
    IMessageBuilderPostCc Cc(String arg);
    IMessageBuilderPostSubject Subject(String arg);
}
interface IMessageBuilderPostSubject {
    Message Body(String arg);
}
```

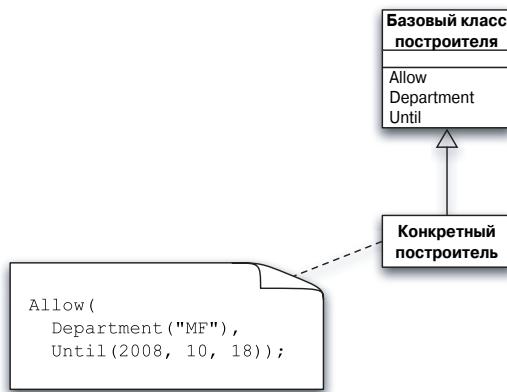
В этом примере естественным завершающим методом является `Body`, поэтому он возвращает готовое сообщение.

Глава 36

Перенос области видимости в объект

Object Scoping

Размещение сценария DSL таким образом, чтобы неквалифицированные ссылки разрешались в единственный объект



Вложенные функции (Nested Function (361)) и (в определенной степени) последовательность функций (Function Sequence (357)) могут обеспечить хороший синтаксис DSL, но в основной своей форме они требуют за это высокой цены: глобальных функций и, что еще хуже, глобального состояния.

Перенос области видимости в объект снимает эти проблемы путем разрешения всех неквалифицированных вызовов в один объект, что позволяет избежать загромождения глобального пространства имён глобальными функциями и хранить любые данные синтаксического анализа в пределах этого базового объекта. Наиболее распространенный способ добиться этого — написать сценарий DSL внутри подкласса построителя, который определяет функции. Это позволяет хранить данные синтаксического анализа в единственном объекте.

36.1. Как это работает

Одним из многих полезных свойств объектов является то, что каждый объект определяет область видимости для содержащихся в нем функций и данных. Наследование позволяет использовать эту область видимости отдельно от места, где она определена. DSL могут применять эту возможность, определяя DSL функции в базовом классе, а затем позволяя разработчикам писать программы DSL в подклассах. Базовый класс может также определить поля для хранения любых необходимых данных синтаксического анализа.

Такой базовый класс — очевидное место для построителя выражений (*Expression Builder* (349)). После этого клиенты пишут DSL-программы в подклассе построителя выражений. Наследование позволяет им добавлять в подкласс другие DSL-функции или при необходимости даже перекрывать базовые функции в объекте DSL.

Хотя наследование — наиболее распространенный механизм, используемый для такой работы, некоторые языки предоставляют другие способы переноса области видимости в объект. Примером этого является вычисление экземпляра Ruby, которое позволяет выполнить любой программный код в контексте конкретного объекта (с помощью метода `instance_eval`). Это позволяет автору DSL написать исходный текст без объявления каких-либо ссылок на базовый класс, который определяет язык.

Другой метод, который доступен в Java, — инициализаторы экземпляров. Он не так известен и не часто используется, но в данном случае может пригодиться.

36.2. Когда это использовать

Перенос области видимости в объект решает проблемы глобальности во вложенных функциях (*Nested Function* (361)) и в последовательности функций (*Function Sequence* (357)) и как таковой всегда стоит того, чтобы подумать о его применении. Перенос области видимости в объект позволяет иметь неквалифицированные вызовы функций в DSL и разрешать их в методы экземпляра объекта. Это не только позволяет избежать путаницы в глобальном пространстве имен, но и хранить данные синтаксического анализа в построителе выражений (*Expression Builder* (349)). Я нахожу эти преимущества весьма убедительными и всегда предлагаю использовать перенос области видимости в объект, если имеется такая возможность.

Иногда, впрочем, это невозможно. Как минимум, для этого понадобится объектно-ориентированный язык программирования. Для меня это не представляет никакой проблемы, так как я в любом случае предпочитаю объектно-ориентированные языки программирования.

Более распространенной проблемой является то, что перенос области видимости в объект накладывает ограничения на то, где может работать ваш DSL-сценарий. В наиболее распространном случае наследования это означает, что вы должны поместить сценарий DSL в метод подкласса построителя выражений. Это не слишком проблематично для автономных DSL-сценариев. Такие сценарии часто располагаются в собственных файлах и хорошо отделены от прочего кода. В этом случае добавляется небольшой синтаксический шум при создании структуры наследования, но он не слишком навязчив. (Можно избежать и его, используя такие технологии, как `instance_eval` из Ruby.) Реальная проблема возникает в случае фрагментарного DSL, когда наследование, которое заставляет вас применять перенос области видимости в объект, может быть неудобным или даже невозможным.

Перенос области видимости в объект представляет собой в основном противоядие от глобальных функций, так что не стоит забывать, что самые большие проблемы глобальных функций связаны с изменением глобальных данных. Распространенная ситуация, когда вы не сталкиваетесь с этой проблемой, — когда глобальная функция просто создает и возвращает новый объект, такой как `Date.today()`. Статические методы, которые, по сути, являются глобальными, могут возвращать объекты, представляющие собой обычные объекты или построители выражений. Если вы можете организовать свои неквалифицированные функции таким образом, то необходимость в переносе области видимости в объект станет гораздо меньшей.

Если DSL создан так, чтобы позволять пользователю использовать свой подкласс класса для переноса области видимости в объект, то это делает DSL более расширяемым. Пользовательский подкласс может добавить методы для расширения языка. Действительно, если некоторые конкретные методы нужны только в одном сценарии, то они могут быть определены непосредственно подклассом данного сценария.

36.3. Коды безопасности (C#)

Предположим, что у нас есть здание, в котором хранятся всякие секретные проекты. Здание разделено на зоны, и в каждой из зон есть свои правила безопасности, которые определяют, какие сотрудники могут входить в эти зоны. Когда работник подходит к двери в зону, система безопасности проверяет соответствие работника правилам зоны и решает, следует ли предоставлять ему доступ в зону.

DSL, который я собираюсь создать, будет поддерживать правила наподобие следующих.

```
class MyZone : ZoneBuilder {
    protected override void doBuild() {
        Allow(
            Department("MF"),
            Until(2008, 10, 18));
        Refuse(Department("Finance"));
        Refuse(Department("Audit"));
        Allow(
            GradeAtLeast(Grade.Director),
            During(1100, 1500),
            Until(2008, 5, 1));
        Refuse(
            Department("k9"),
            GradeAtLeast(Grade.Director));
        Allow(Department("k9"));
    }
}
```

36.3.1. Семантическая модель

Семантическая модель (*Semantic Model* (171)) имеет класс зоны с множественными правилами допуска. Каждое правило допуска представляет собой либо разрешение (определяющее условия, при которых доступ разрешен), либо запрет (определяющий условия, при которых доступ закрыт). Правило допуска имеет тело (которое мы рассмотрим позже) и метод проверки допуска сотрудника.

```
abstract class AdmissionRule {
    protected RuleElement body;
    protected AdmissionRule(RuleElement body) {
        this.body = body;
    }
}
```

```

    public abstract AdmissionRuleResult CanAdmit(Employee e);
}
enum AdmissionRuleResult {ADMIT, REFUSE, NO_OPINION};

```

Два вида правил обрабатываются с помощью наследования. Каждый класс предоставляет реализацию метода `CanAdmit`.

```

class AllowRule : AdmissionRule {
    public AllowRule(RuleElement body) : base(body) {}
    public override AdmissionRuleResult CanAdmit(Employee e) {
        if (body.eval(e)) return AdmissionRuleResult.ADMIT;
        else return AdmissionRuleResult.NO_OPINION;
    }
}

class RefusalRule : AdmissionRule {
    public RefusalRule(RuleElement body) : base(body) {}
    public override AdmissionRuleResult CanAdmit(Employee e) {
        if (body.eval(e)) return AdmissionRuleResult.REFUSE;
        else return AdmissionRuleResult.NO_OPINION;
    }
}

```

При выполнении запроса доступа сотрудника класс зоны по очереди проверяет правила, изучая возвращаемые результаты.

```

class Zone...
private IList<AdmissionRule> rules =
    new List<AdmissionRule>();
public void AddRule(AdmissionRule arg) {
    rules.Add(arg);
}
public bool WillAdmit(Employee e) {
    foreach (AdmissionRule rule in rules) {
        switch(rule.CanAdmit(e)) {
            case AdmissionRuleResult.ADMIT:
                return true;
            case AdmissionRuleResult.NO_OPINION:
                break;
            case AdmissionRuleResult.REFUSE:
                return false;
            default:
                throw new InvalidOperationException();
        }
    }
    return false;
}

```

В случае, если ни одно из правил не дает ответа, по умолчанию метод отказывает в доступе (возвращает `false`).

Тело правила доступа представляет собой составную конструкцию элементов правила, по сути — спецификацию (*Specification* [7]). Объявленный тип является интерфейсом.

```

internal interface RuleElement {
    bool eval(Employee emp);
}

```

Все реализации проверяют соответствующие атрибуты сотрудника. Вот проверка ранга работника и отдела, в котором он работает.

```

internal class MinimumGradeExpr : RuleElement {
    private readonly Grade minimum;
    public MinimumGradeExpr(Grade minimum) {

```

```

        this.minimum = minimum;
    }
    public bool eval(Employee emp) {
        if (null == emp.Grade) return false;
        return emp.Grade.IshigherOrEqualTo(minimum);
    }
}

internal class DepartmentExpr : RuleElement {
    private readonly string dept;
    public DepartmentExpr(string dept) {
        this.dept = dept;
    }
    public bool eval(Employee emp) {
        return emp.Department == dept;
    }
}

```

Составные элементы можно объединять в логические структуры.

```

class AndExpr : RuleElement {
    private readonly List<RuleElement> elements;
    public AndExpr(params RuleElement[] elements) {
        this.elements = new List<RuleElement>(elements);
    }
    public bool eval(Employee emp) {
        return elements.TrueForAll(element => element.eval(emp));
    }
}

```

Так, если необходимо разрешить доступ старшим программистам из отдела K9, можно настроить зону следующим образом.

```

zone.AddRule(new AllowRule(
    new AndExpr(
        new MinimumGradeExpr(Grade.SeniorProgrammer),
        new DepartmentExpr("K9")));
}

```

36.3.2. DSL

Для переноса области видимости в объект создается надкласс построителя, который затем можно наследовать для создания DSL. Вот пример подкласса, демонстрирующего поддерживаемый DSL.

```

class MyZone : ZoneBuilder {
    protected override void doBuild() {
        Allow(
            Department("MF"),
            Until(2012, 10, 18));
        Refuse(Department("Finance"));
        Refuse(Department("Audit"));
        Allow(
            GradeAtLeast(Grade.Director),
            During(1100, 1500),
            Until(2012, 5, 1));
        Refuse(
            Department("k9"),
            GradeAtLeast(Grade.Director));
        Allow(Department("k9"));
    }
}

```

Первое правило разрешает доступ сотрудникам отдела MF вплоть до определенной даты в 2012 году. Затем явным образом закрывается доступ для сотрудников финансового отдела и отдела аудита (с помощью двух отдельных конструкций) и наконец открывается доступ для всех директоров в определенное время и до другой конечной даты.

Хотя лежащая в основе модель допускает произвольные логические выражения, DSL проще. Каждое правило доступа представляет собой конъюнкцию (“И”) своих составных частей. Вот почему для двух отделов мне потребовались два запрещающих правила. Если бы я разместил их в одной конструкции, то правило запрещало бы доступ только тем сотрудникам, которые одновременно работают в двух указанных отделах.

Произвольные логические выражения — мощный инструмент, но часто он сложен для пользователей, в особенности для тех, кто не страдает любовью к компьютерам и булевой алгебре в острой форме. Поэтому некоторый вид упрощенных структур в DSL может оказаться очень кстати.

DSL включает методы, определенные в базовом классе построителя. Это позволяет мне вызывать их в подклассе, никак их не квалифицируя. Метод `Allow` добавляет к зоне новое разрешающее правило, тело которого представляет собой конъюнкцию аргументов этого метода.

```
class ZoneBuilder...
private Zone zone;
public ZoneBuilder Allow(params RuleElement[] rules) {
    var expr = new AndExpr(rules);
    zone.AddRule(new AllowRule(expr));
    return this;
}
```

Я использую метод с переменным количеством аргументов как список литералов (`Literal List (415)`). (Если имеется только одно подвыражение, в показанном обертывании выражения `and` нет необходимости.)

Каждый аргумент создается с помощью функций базового построителя.

```
class ZoneBuilder...
internal RuleElement GradeAtLeast(Grade grade) {
    return new MinimumGradeExpr(grade);
}
internal RuleElement Department(String name) {
    return new DepartmentExpr(name);
}
```

Для добавления в систему нового элемента я определяю новое выражение модели и функцию в построителе.

```
class ZoneBuilder...
internal RuleElement Until(int year, int month, int day) {
    return new EndDateExpr(year, month, day);
}

internal class EndDateExpr : RuleElement {
    private readonly DateTime date;
    public EndDateExpr(int year, int month, int day) {
        date = new DateTime(year, month, day);
    }
    public bool eval(Employee emp) {
        return Clock.Date < date;
    }
}
```

Аналогично добавлению правил для всех пользователей DSL можно расширить DSL для конкретных DSL-программ. Предположим, что моему отделу необходимо ограничить доступ в определенные часы. Можно поместить соответствующий код непосредственно в подкласс.

```
class MyZone...
private RuleElement During(int begin, int end) {
    return new TimeOfDayExpr(begin, end);
}

private class TimeOfDayExpr : RuleElement {
    private readonly int begin, end;
    public TimeOfDayExpr(int begin, int end) {
        this.begin = begin;
        this.end = end;
    }
    public bool eval(Employee emp) {
        return (Clock.Time >= begin) && (Clock.Time <= end);
    }
}
```

Если эта функциональная возможность требуется и в других сценариях, но нельзя внести изменения в библиотеку, можно создать собственный класс построителя зоны, являющийся подклассом библиотечного построителя, и позволить сценариям наследовать его. После этого можно размещать в таком абстрактном построителе зоны любые необходимые методы.

Перенос области видимости в объект помогает снизить шум в DSL, но вносит шум в код, объявляющий класс DSL. Первые две строки (и фигурные скобки) представляют собой такой шум. Ситуация могла бы быть немного хуже; естественный способ использования этого класса заключается в передаче зоны построителю в конструкторе, но это заставило бы добавить объявление конструктора в подкласс. Я избежал этого путем передачи зоны отдельным методом.

```
class ZoneBuilder...
internal void Build(Zone zone) {
    this.zone = zone;
    doBuild();
}
protected abstract void doBuild();
```

Язываю его следующим образом.

```
class DslTest...
new MyZone().Build(zone);
```

Это мелочь, но она снижает количество шума в тексте DSL. В сумме такие мелочи дают большой эффект.

36.4. Использование вычисления экземпляра (Ruby)

Хотя перенос области видимости в объект — очень ценный шаблон, поскольку он обеспечивает применение неквалифицированных имен без глобальных артефактов, использование подтипов вносит ограничения. В случае автономного DSL сценарий нуждается в некотором начальном и конечном “шуме” для настройки контекста. В случае фрагментарных DSL для написания выражений DSL вы должны находиться в подклассе построителя.

Ruby имеет очень хороший механизм, который можно использовать для обхода этих проблем, — вычисление экземпляра. Идея вычисления экземпляра заключается в том, что можно взять некоторый текст и вычислить его в контексте конкретного экземпляра объекта Ruby. Любые неквалифицированные вызовы методов разрешаются в вызовы этого экземпляра, как если бы они находились внутри некоторого метода экземпляра класса. Это позволяет писать DSL с помощью переноса области видимости в объект без необходимости возиться с наследованием.

Таким образом, для приведенного выше примера с зонами получается следующий сценарий.

```
allow {
  department:mf
  ends 2008, 10, 18
}

refuse department :finance

refuse department :audit

allow {
  gradeAtLeast :director
  during 1100, 1500
  ends 2008, 5, 1
}

refuse {
  department :k9
  gradeAtLeast :director
}

allow department :k9
```

Построитель выполняет этот код с помощью простого вызова.

```
class Builder...
  def load_file aFilename
    self.load(File.readlines(aFilename).join("\n"))
  end
```

Неквалифицированные вызовы функций разрешаются в методы построителя.

```
class Builder...
  def allow anExpr = nil, &block
    @zone << AllowRule.new(form_expression(anExpr, &block))
  end
```

В этом заключается сущность использования вычисления экземпляра для обработки переноса области видимости в объект. Однако, когда я вставлял этот пример в книгу, нашлось несколько интересных моментов, мимо которых я не смог пройти.

В коде примера я старался следовать структуре примера на C#. Тем не менее я чувствовал, что код будет более удобочитаемым, если составное условие будет использовать вложенные замыкания (*Nested Closure* (403)), а не вложенные функции (*Nested Function* (361)). Отказ от этого приводит к усложнению кода. Если в разрешающей или запрещающей инструкции есть только одно предложение, нужно вернуть его значение; если же имеется вложенный блок, нужно вернуть выражение *and* для значений каждого из предложений.

```
class Builder...
  def form_expression anExpr = nil, &block
    if block_given?
      AndExprBuilder.interpret(&block)
    else
      anExpr
    end
  end
```

В простом случае я делаю метод построителя просто возвращающим элемент правила, так что этот элемент правила оказывается обернутым в родительское разрешающее правило.

```
class Builder...
  def gradeAtLeast gradeSymbol
    return RuleElementBuilder.new.gradeAtLeast gradeSymbol
  end

class RuleElementBuilder
  def gradeAtLeast gradeSymbol
    return MinimumGradeExpr.new gradeSymbol
  end
```

Если же у меня есть вложенное замыкание, я использую для этого выражения дочерний построитель и вновь применяю к нему вычисление экземпляра, так что выражения в DSL связываются с дочерним, а не родительским построителем.

```
class AndExprBuilder...
  def initialize &block
    @rules = []
    @block = block
  end

  def self.interpret &block
    return self.new(&block).value
  end

  def value
    instance_eval(&@block)
    return AndExpr.new(*@rules)
  end

  def gradeAtLeast gradeSymbol
    @rules << RuleElementBuilder.new.gradeAtLeast(gradeSymbol)
  end
```

Этот механизм позволяет обрабатывать вызовы методов наподобие `gradeAtLeast` по-разному в разных частях DSL. В Ruby хорошо использовать последовательность функций (Function Sequence (357)) внутри вложенного замыкания, так как это позволяет отделить содержимое группы символами новой строки, а не запятыми.

36.5. Использование инициализатора экземпляра (Java)

Сравнительно ненавязчивый способ встроенного использования переноса области видимости в объект заключается в применении инициализатора экземпляра. Этот метод был популяризирован библиотекой JMock; признаюсь, что я полностью пренебрегал этой возможностью языка, пока не увидел ее в действии. Ее использование в сценарии DSL выглядит следующим образом.

```

ZoneBuilder builder = new ZoneBuilder() {{
    allow(department(MF));
    refuse(department(FINANCE));
    refuse(department(AUDIT));
    allow(
        gradeAtLeast(DIRECTOR),
        department(K9));
}}
zone = builder.getValue();

```

Построитель, который делает все это работающим, очень похож на свой аналог на C#.

```

class ZoneBuilder...
private Zone value = new Zone();
public Zone getValue() {
    return value;
}
public ZoneBuilder refuse(RuleElement... rules) {
    value.addRule(new RefusalRule(new AndExpr(rules)));
    return this;
}
public ZoneBuilder allow(RuleElement... rules) {
    value.addRule(new AllowRule(new AndExpr(rules)));
    return this;
}
public RuleElement gradeAtLeast(Grade g) {
    return new MinimumGradeExpr(g);
}
public RuleElement department(Department d) {
    return new DepartmentExpr(d);
}

```

Вся хитрость заключается в использовании двойных фигурных скобок в сценарии DSL. Это создает не экземпляр построителя зоны, а внутренний класс, который является экземпляром *подкласса* построителя зоны. Код конструктора этого одноразового подкласса располагается между двойными фигурными скобками. Это всегда можно сделать в Java, хотя я и не видел, чтобы данная возможность широко использовалась. Поскольку код между двойными фигурными скобками находится в подклассе построителя зоны, мы получаем требующийся нам перенос области видимости в объект.

Глава 37

Замыкание

Closure

Блок кода, который может быть представлен как объект (или структура данных) и помещен в поток кода с обращением к своей лексической области видимости

```
var threshold =  
    ComputeThreshold();  
var heavyTravellers =  
    employeeList.FindAll(e => e.MilesOfCommute > threshold);
```

Известен также как “лямбда”, “блок” или “анонимная функция”

Предположим, у вас есть коллекция объектов и вы хотите отфильтровать ее некоторым способом. Написание метода для каждого фильтра приводит к дублированию в настройке и работе фильтра.

Используя замыкание, можно разделить настройку и работу фильтра и передать в качестве условия фильтра произвольный блок кода.

37.1. Как это работает

Замыкания представляют собой особенность языка, которая, несмотря на достаточно долгое существование, лишь недавно начала прокладывать путь к умам многих разработчиков программного обеспечения. Вероятно, это связано с тем, что языки, в которых имеются замыкания (такие как Lisp и Smalltalk), не были частью культуры C, определяющей основное направление развития современных языков программирования.

В этой книге я использую термин “замыкание” (closure), но стандартного термина для этого элемента языка нет. Их называют “лямбда”, “анонимные функции” и “блоки”. В каждом использующем их языке, как правило, есть свой термин. Например, программисты на Lisp употребляют термин “лямбда”, программисты на Smalltalk и Ruby — “блок”. Несмотря на совпадение названий блоки в Smalltalk и Ruby — это далеко не то же самое, что блоки в С-образных языках программирования.

Теперь, отложив в сторону всю эту терминологическую болтовню, я могу сказать, что же это такое на самом деле. Вот мое отправное определение: замыкание — это фрагмент

кода, который можно рассматривать как объект. Чтобы начать серьезно относиться к нему, следует рассмотреть пример.

Рассмотрим задачу получения подмножества данных из коллекции. Представим, что у нас есть список сотрудников, и мы хотим отобрать из него тех, кому приходится много разъезжать.

```
int threshold = ComputeThreshold();
var heavyTravellers = new List<Employee>();
foreach (Employee e in employeeList)
    if (e.MilesOfCommute > threshold) heavyTravellers.Add(e);
```

Где-то в другом месте нам нужно выполнить отбор из списка бездельников... конечно, я хотел сказать — менеджеров.

```
var managerList = new List<Employee>();
foreach (Employee e in employeeList)
    if (e.IsManager) managerList.Add(e);
```

Эти два фрагмента во многом дублируют друг друга. В обоих случаях нам нужен список, который формируется путем поочередного получения членов исходного списка, применения к каждому из них булевой функции и сбору в новую коллекцию тех членов, для которых функция возвращает значение `true`. Удаление повторов легко представить, но сложно написать на многих языках, потому что варьируется небольшой фрагмент кода, который обычно не так-то легко параметризовать.

Наиболее очевидный способ параметризации чего-то наподобие рассматриваемого фрагмента — это превратить его в объект. Мне нужен метод, работающий со списком, который позволит осуществить выбор из списка на основе отдельного переданного ему объекта.

```
class MyList<T> {
    private List<T> contents;
    public MyList(List<T> contents) {
        this.contents = contents;
    }
    public List<T> Select(FilterFunction<T> p) {
        var result = new List<T>();
        foreach (T candidate in contents)
            if (p.Passes(candidate)) result.Add(candidate);
        return result;
    }
}
interface FilterFunction<T> {
    Boolean Passes(T arg);
}
```

Затем его можно использовать для отбора менеджеров следующим образом.

```
class ManagersPredicate : FilterFunction<Employee> {
    public Boolean Passes(Employee e) {
        return e.IsManager;
    }
}
```

Определенное удовлетворение такой подход приносит, но здесь требуется так много кода для настройки объекта предиката, что лечение хуже, чем болезнь. Это особенно справедливо для отбора путешественников. Здесь объекту предиката следует передать параметр, а это значит, что моему предикату нужен конструктор.

```
var threshold = ComputeThreshold();
var heavyTravellers =
    new MyList<Employee>(employeeList)
```

```

    .Select(new HeavyTravellerPredicate(threshold));

class HeavyTravellerPredicate : FilterFunction<Employee> {
    private int threshold;
    public HeavyTravellerPredicate(int threshold) {
        this.threshold = threshold;
    }
    public Boolean Passes(Employee e) {
        return e.MilesOfCommute > threshold;
    }
}

```

По сути, замыкание — это более элегантное решение проблемы, упрощающее создание фрагмента кода и его передачу в качестве объекта.

Как вы вскоре увидите, примеры созданы на C#. Это связано с тем, что в последние годы язык программирования C# устойчиво развивается по направлению к более удобному использованию замыканий. В C# 2.0 было введено понятие анонимных делегатов, которые являются большим шагом в этом направлении. Вот тот же пример отбора путешественников с помощью анонимных делегатов.

```

var threshold = ComputeThreshold();
var heavyTravellers = employeeList.FindAll(
    delegate(Employee e)
    {return e.MilesOfCommute > threshold;});

```

Первое, что нужно отметить, — гораздо меньшее количество кода. Дублирование между этим выражением и аналогичным для поиска менеджеров значительно уменьшено. Далее, чтобы все это работало, я применил библиотечную функцию к классу списка C#, аналогичную функции отбора, которую я написал самостоятельно при разработке предиката вручную. В C# 2 в библиотеки был внесен ряд изменений, использующих преимущества делегатов. Это важный момент: чтобы замыкания стали действительно полезной частью языка, библиотеки должны быть написаны с учетом их применения.

Последнее, что иллюстрирует этот фрагмент, — простота использования параметра: я просто обращаюсь к нему в логическом выражении. Я могу воспользоваться любой локальной переменной, находящейся в области видимости этого выражения, что позволяет избежать всей этой сути с параметрами в версии с объектом предиката.

Такое обращение к переменным в области видимости и есть то, что формально делает это выражение замыканием. Можно сказать, что делегат замкнут в лексической области видимости, где он определен. Даже если взять делегата и сохранить его где-нибудь для последующего выполнения, эти переменные останутся видимыми и доступными для использования. По сути, система должна получить копию кадра стека, чтобы предоставить замыканию доступ ко всему, что оно должно видеть. И теория, и реализация этого довольно сложны, но получающийся результат используется вполне естественно.

(Некоторые люди определяют “замыкание” только для обозначения инстанцированного фрагмента кода, замкнутого вокруг некоторых переменных в лексической области видимости. Как это часто бывает, использование слова “замыкание” весьма противоречиво.)

Версия C# 3 пошла еще дальше. Вот еще одно выражение для отбора путешественников.

```

var threshold = ComputeThreshold();
var heavyTravellers =
    employeeList.FindAll(e => e.MilesOfCommute > threshold);

```

Обратите внимание на небольшие на самом деле изменения — здесь главным фактом является существенно более компактный синтаксис. Это небольшая, но жизненно

важная разница. Полезность замыканий прямо пропорциональна краткости их применения. Такой синтаксис делает их гораздо более удобными для чтения.

Есть и другое отличие, которое является важной частью краткости синтаксиса. В примере с делегатом мне нужно было указать тип параметра `Employee e`. В лямбде не нужно указывать тип, потому что C# 3.0 обладает возможностью выведения типа, а это означает, что, поскольку он может выяснить тип результата в правой части присваивания, вам не нужно указывать его слева.

Следствием всего этого является то, что можно создавать замыкания и относиться к ним так же, как к любому другому объекту. Их можно хранить в переменных и выполнять всякий раз в случае необходимости. Чтобы проиллюстрировать это, можно создать класс клуба путешественников с полем селектора.

```
class Club...
    Predicate<Employee> selector;
    internal Club(Predicate<Employee> selector) {
        this.selector = selector;
    }
    internal Boolean IsEligable(Employee arg) {
        return selector(arg);
    }
```

Его можно использовать следующим образом.

```
public void clubRecognizesMember() {
    var rebecca = new Employee { MilesOfCommute = 5000 };
    var club = createHeavyTravellersClub(1000);
    Assert.IsTrue(club.IsEligable(rebecca));
}

private Club createHeavyTravellersClub(int threshold) {
    return new Club(e => e.MilesOfCommute > threshold);
}
```

Этот код создает клуб с помощью одной функции, используя параметр для установки порога. Клуб содержит замыкание, включающее ссылку на параметр, который находится вне области видимости. После этого я могу использовать клуб для выполнения замыкания в любой момент в будущем.

В этом случае замыкание селектора при создании не вычисляется. Вместо этого мы создаем замыкание, храним его и вычисляем позже (возможно, несколько раз). Эта возможность создавать блок кода для последующего выполнения и делает замыкания столь полезными для адаптивных моделей (Adaptive Model (478)).

Еще один язык программирования, применяемый в этой книге и активно использующий замыкания, — это Ruby. Изначально Ruby создавался с поддержкой замыканий, поэтому большинство программ и библиотек Ruby активно их использует. Определение класса клуба выглядит в Ruby следующим образом.

```
class Club...
    def initialize &selector
        @selector = selector
    end
    def eligible? anEmployee
        @selector.call anEmployee
    end
```

И мы используем его следующим образом.

```
def test_club
    rebecca = Employee.new(5000)
```

```
club = create_heavy_travellers_club
assert club.eligible?(rebecca)
end
def create_heavy_travellers_club
  threshold = 1000
  return Club.new { |e| e.miles_of_commute > threshold}
end
```

В Ruby замыкание можно определить с помощью либо фигурных скобок, либо пары `do...end`.

```
threshold = 1000
return Club.new do |e|
  e.miles_of_commute > threshold
end
```

Эти два синтаксиса практически эквивалентны. На практике программисты используют фигурные скобки для односторонних блоков и `do...end` — для многострочных.

Неприятность при использовании этого синтаксиса Ruby заключается в том, что он позволяет передавать в функцию только одно замыкание. Для передачи нескольких замыканий придется использовать менее элегантный синтаксис.

37.2. Когда это использовать

Как и многим программистам, использовавшим языки программирования с поддержкой замыканий, мне очень не хватает этой возможности в языках без такой поддержки. Замыкания представляют собой ценный инструмент для получения фрагментов логики и их размещения для устранения дублирования и поддержки пользовательских управляющих структур.

Замыкания играют несколько полезных ролей в DSL. Совершенно очевидно, что они являются жизненно важным элементом для вложенных замыканий ([Nested Closure \(403\)](#)). Они также могут существенно упростить определение адаптивной модели ([Adaptive Model \(478\)](#)).

Глава 38

Вложенные замыкания

Nested Closure

Выражают подэлементы инструкции вызова функции путем их помещения в аргументы в виде замыканий

```
computer do
  processor do
    cores 2
    i386
    speed 2.2
  end
  disk do
    size 150
  end
  disk do
    size 75
    speed 7200
    sata
  end
end
```

38.1. Как это работает

Основная идея вложенных замыканий подобна идее вложенных функций (Nested Function (361)), но дочерние выражения вызова функции “обернуты” в замыкания. Чтобы увидеть отличия, рассмотрим вызов для создания процессора с помощью вложенных функций в Ruby.

```
processor(
  cores 2,
  i386
)
```

Вот как эта же задача решается с помощью вложенных замыканий.

```
processor do
  cores 2
  i386
end
```

Вместо двух аргументов вложенной функции я передаю один аргумент, представляющий собой вложенное замыкание и содержащий две вложенные функции. (Здесь я использую язык программирования Ruby, предоставляющий замыкания с синтаксисом, подходящим для нашего рассмотрения.)

Размещение подэлементов во вложенном замыкании непосредственно влияет на мою реализацию — я должен разместить код вычисления замыкания. В случае вложенной функции мне не нужно этого делать, поскольку язык автоматически вычисляет функции `cores` и `i386` перед вызовом функции `processor`. При использовании в качестве аргумента замыкания первой вызывается функция `processor`, а вычисление замыкания осуществляется только тогда, когда я явно его программирую. Так, обычно я вычисляю замыкание в функции `processor`. Эта функция может решать и другие задачи до и после вычисления замыкания, как, например, создание переменных контекста (*Context Variable* (187)).

В приведенном выше примере содержание замыкания представляет собой последовательность функций (*Function Sequence* (357)). Одной из проблем при использовании последовательности функций является то, что несколько функций взаимодействуют одна с другой с помощью скрытых переменных контекста. Хотя все равно необходимо прибегать к ним внутри вложенных замыканий, функция `processor` может создать такую переменную контекста до вычисления замыкания и уничтожить ее после вычисления. Это может значительно уменьшить проблему переменных контекста.

Еще одним вариантом для подэлементов является использование соединения методов в цепочки (*Method Chaining* (375)). При его применении имеется дополнительное преимущество, заключающееся в том, что родительская функция может настроить заголовок цепочки и передать его замыканию в качестве аргумента.

```
processor do |p|
  p.cores(2).i386
end
```

Достаточно распространена также передача в качестве аргумента переменной контекста.

```
processor do |p|
  p.cores 2
  p.i386
end
```

В этом случае есть последовательность функций с явно указанной переменной контекста. Это зачастую облегчает работу с ней, не внося при этом в код дополнительный беспорядок.

Неквалифицированные функции внутри вложенных замыканий вычисляются в области видимости, в которой они были определены. Так что, опять же, имеет смысл воспользоваться переносом области видимости в объект (*Object Scoping* (387)). Передача переменной контекста в явном виде или использование соединения методов в цепочки позволяет избежать этого и, кроме того, разнести код построения по разным построителям.

Некоторые языки позволяют управлять контекстом выполнения замыканий. Это может позволить вам использовать неквалифицированные функции и по-прежнему применять несколько построителей. Так, в примере с применением `instance_eval` Ruby (“Применение вычисления экземпляра (Ruby)” на с. 411) показано, как это может работать.

В приведенных выше примерах я размещал все подэлементы родительской функции в одном замыкании. Но можно использовать и несколько замыканий. Преимуществом такого подхода является то, что он позволяет вычислять каждое подзамыкание отдельно.

Хороший пример того, где это может оказаться удобным, — условное выражение, такое как в этом примере в Smalltalk.

```
aRoom
  ifDark: [aLight on]
  ifLight: [aLight off]
```

38.2. Когда это использовать

Вложенные замыкания являются полезным средством, поскольку оно сочетает в себе явно выраженную иерархическую структуру вложенных функций (*Nested Function* (361)) с возможностью управления моментом вычисления аргументов. Управление вычислением обеспечивает вас большей гибкостью, помогая избежать многих ограничений вложенных функций.

Самое большое ограничение вложенных замыканий заключается в способе поддержки замыканий базовым языком. Многие языки программирования вовсе не поддерживают замыкания. Языки же с поддержкой этой возможности часто обеспечивают такой синтаксис, что применять его в DSL неудобно.

Обычно стоит рассматривать вложенные замыкания в качестве расширения вложенных функций, последовательности функций (*Function Sequence* (357)) и соединения методов в цепочки (*Method Chaining* (375)). Явное управление вычислениями дает свои преимущества при работе с каждым из указанных методов. Все они, однако, сводятся к тому, что вы можете выполнить конкретные операции инициализации и деструкции “по обе стороны” от вычисления замыкания. В случае последовательности функций это означает, что вы можете подготовить переменные контекста (*Context Variable* (187)) непосредственно перед их применением в замыкании. В случае соединения методов в цепочки можно настроить заголовок цепочки перед вызовом замыкания.

38.3. Заворачивание последовательности функций во вложенное замыкание (Ruby)

Свой первый пример я решил сделать максимально простым — применение вложенных замыканий в соединении с последовательностью функций (*Function Sequence* (357)). Вот сценарий DSL.

```
class BasicComputerBuilder < ComputerBuilder
  def doBuild
    computer do
      processor do
        cores 2
        i386
        processorSpeed 2.2
      end
      disk do
        size 150
      end
      disk do
        size 75
        diskSpeed 7200
        sata
      end
    end
  end
```

```
    end
end
```

Приступая к рассмотрению, сравним этот исходный текст с версией, использующей только последовательность функций, которая имеет следующий вид.

```
class BasicComputerBuilder < ComputerBuilder
  def doBuild
    computer
      processor
        cores 2
        i386
        processorSpeed 2.2
    disk
      size 150
    disk
      size 75
      diskSpeed 7200
      sata
  end
end
```

С точки зрения сценария единственное изменение при использовании вложенного замыкания заключается в добавлении ограничителей `do...end`. Тем самым вносится явная иерархическая структура в то, что иначе было бы простой линейной последовательностью с соглашением о форматировании. Этот дополнительный синтаксис не доставляет неприятностей, поскольку указывает структуру таким образом, который имеет смысл для читателя.

Теперь перейдем к реализации. Как обычно, я использую перенос области видимости в объект (*Object Scoping* (387)), так что неквалифицированные функции разрешаются в вызовы методов построителя выражений (*Expression Builder* (349)). (Примечание для программистов на Ruby: я использую подтипы в педагогических целях; в реальности я бы применил `instance_eval`.) Базовую структуру применения вложенного замыкания можно увидеть в предложении `computer`.

```
class ComputerBuilder...
  def computer &block
    @result = Computer.new
    block.call
  end
```

Я передаю в замыкание аргумент (*Ruby* называет замыкания “блоками”), настраиваю контекст и вызываю замыкание. Функция `processor` может использовать этот контекст и повторить процесс для своих потомков.

```
class ComputerBuilder...
  def processor &block
    @result.processor = Processor.new
    block.call
  end
  def cores arg
    @result.processor.cores = arg
  end
  def i386
    @result.processor.type = :i386
  end
  def processorSpeed arg
    @result.processor.speed = arg
  end
```

То же самое я делаю и для дисков. На этот раз единственным отличием является применение более идиоматичного ключевого слова `yield` для вызова неявно переданного блока. (Этот механизм Ruby использует для упрощения работы с единственным аргументом блока.)

```
class ComputerBuilder...
  def disk
    @result.disks << Disk.new
    yield
  end
  def size arg
    @result.disks.last.size = arg
  end
  def sata
    @result.disks.last.interface = :sata
  end
  def diskSpeed arg
    @result.disks.last.speed = arg
  end
```

38.4. Простой пример (C#)

Для контраста — практически тот же пример на C#.

```
class Script : Builder {
  protected override void doBuild() {
    computer(() => {
      processor(() => {
        cores(2);
        i386();
        processorSpeed(2.2);
      });
      disk(() => {
        size(150);
      });
      disk(() => {
        size(75);
        diskSpeed(7200);
        sata();
      });
    });
  }
}
```

Как можно видеть, структура здесь в точности та же, что и в примере Ruby; главное отличие состоит в наличии гораздо большего количества знаков пунктуации.

Построитель также выглядит удивительно похожим.

```
class Builder...
  protected void computer(BuilderElement child) {
    result = new Computer();
    child.Invoke();
  }
  public delegate void BuilderElement();
  private Computer result;
```

Функция `computer` следует тому же шаблону, что и в случае Ruby: передача аргумента замыкания, инициализация, вызов замыкания, действия по деструкции. Наиболее важное отличие в случае C# в том, что нужно определить тип замыкания, передаваемого предложе-

нием делегирования. В данном случае замыкание не имеет аргументов и ничего не возвращает, но в более сложной ситуации может потребоваться несколько различных типов.

Остальная часть кода аналогична коду Ruby, так что я сэкономлю немного места в книге.

На мой взгляд, вложенное замыкание в C# работает существенно хуже, чем в Ruby. Ограничители `do...end` в Ruby, как мне кажется, гораздо более естественны по сравнению с `() => { ... }` в C#, особенно при наличии обязательных скобок. (В Ruby также можно использовать скобки `{ ... }` в качестве ограничителей замыкания.) Однако чем больше используется обозначений C#, тем меньше это беспокоит. Кроме того, в данном примере в замыкания не передаются аргументы, что существенно увеличивает количество знаков пунктуации в Ruby, но облегчает чтение исходного текста C#, поскольку при этом пустые скобки уже не будут пустыми, а в них будет что-то заключено.

38.5. Применение соединения методов в цепочки (Ruby)

Вложенные замыкания могут работать разными способами. Вот пример применения соединения методов в цепочки (Method Chaining (375)).

```
ComputerBuilder.build do |c|
  c.
    processor do |p|
      p.cores(2).
        i386.
        speed(2.2)
    end.
    disk do |d|
      d.size 150
    end.
    disk do |d|
      d.size(75).
        speed(7200).
        sata
    end
end
```

Здесь отличие заключается в том, что каждый вызов передает замыканию объект, использующийся в заголовке цепочки. Это использование аргументов замыкания может добавить в DSL-сценарий шум (так как требует наличия скобок вокруг аргументов), но зато обладает тем преимуществом, что больше не нужно прибегать к переносу области видимости в объект (Object Scoping (387)), а значит, можно легко использовать код в фрагментарном стиле.

Вызов метода `build` создает экземпляр построителя и передает его замыканию в качестве аргумента.

```
class ComputerBuilder...
  attr_reader :content
  def initialize
    @content = Computer.new
  end
  def self.build &block
    builder = self.new
    block.call(builder)
    return builder.content
  end
```

Другой полезной особенностью этого подхода является то, что он упрощает разделение различных методов построителя на группы малых, связанных построителей выражений (*Expression Builder* (349)). В предложении `processor` вводится новый построитель (использующий более компактное ключевое слово `yield`).

```
class ComputerBuilder...
def processor
  p = ProcessorBuilder.new
  yield p
  @content.processor = p.content
  return self
end

class ProcessorBuilder
attr_reader :content
def initialize
  @content = Processor.new
end
def cores arg
  @content.cores = arg
  self
end
def i386
  @content.type = :i386
  self
end
def speed arg
  @content.speed = arg
  self
end
end
```

Диски также обрабатываются с помощью построителя диска.

```
class ComputerBuilder...
def disk
  currentDisk = DiskBuilder.new
  yield currentDisk
  @content.disks << currentDisk.content
  return self
end

class DiskBuilder
attr_reader :content
def initialize
  @content = Disk.new
end
def size arg
  @content.size = arg
  self
end
def sata
  @content.interface = :sata
  self
end
def speed arg
  @content.speed = arg
  self
end
end
```

Помимо наилучшего разбиения методов построителей, этот подход позволяет использовать неквалифицированный метод `speed` как для процессора, так и для диска, не внося при этом никакой неоднозначности.

38.6. Последовательность функций с явными аргументами замыканий (Ruby)

В предыдущем примере мы видели преимущества разбиения на несколько построителей выражений (Expression Builder (349)). При таком подходе каждый построитель получается меньшим по размеру и более “сплоченным”. Кроме того, таким образом в разных частях языка, оказывается, можно использовать одно и то же имя (как, например, скорость процессора и диска). Явные аргументы замыкания также позволяют легко использовать DSL в фрагментарном контексте.

Хотя соединение методов в цепочки (Method Chaining (375)) дает нам эти преимущества, получающийся в результате DSL-сценарий может выглядеть довольно грубо. Взаимодействие между вложенными замыканиями и соединением методов в цепочки неизбежно будет идеальным. Большинство DSL в Ruby, с которыми я имел дело, этот стиль не используют.

Вместо этого они применяют последовательность функций (Function Sequence (357)) в пределах каждого замыкания, но для того, чтобы обеспечить несколько построителей, явным образом передают аргумент замыкания. В этом стиле наш сценарий конфигурации компьютера выглядит следующим образом.

```
ComputerBuilder.build do |c|
  c.processor do |p|
    p.cores 2
    p.i386
    p.speed 2.2
  end
  c.disk do |d|
    d.size 150
  end
  c.disk do |d|
    d.size 75
    d.speed 7200
    d.sata
  end
end
```

Наибольшее отличие в сценарии DSL заключается в том, что имеются отдельные инструкции для каждого предложения в DSL. Для каждой инструкции необходимо указать передаваемый объект в качестве получателя вызова метода. Хотя это добавляет текст в инструкцию, в результате получается более привычный для программистов на Ruby стиль кода.

Реализация очень напоминает случай применения соединения методов в цепочки. Здесь также на верхнем уровне имеется построитель компьютера с методом класса, который создает экземпляр и передает его замыканию.

```
class ComputerBuilder...
  attr_reader :content
  def initialize
    @content = Computer.new
  end
  def self.build
    builder = self.new
```

```

yield builder
return builder.content
end

```

Предложение `processor` вводит новый построитель.

```

class ComputerBuilder...
def processor &block
  p = ProcessorBuilder.new
  yield p
  @content.processor = p.content
end

class ProcessorBuilder
attr_reader :content
def initialize
  @content = Processor.new
end
def cores arg
  @content.cores = arg
end
def i386
  @content.type = :i386
end
def speed arg
  @content.speed = arg
end
end
end

```

Реализацию работы с дисками я оставляю читателям для самостоятельной работы.

38.7. Применение вычисления экземпляра (Ruby)

Переход к явным аргументам замыкания дает немало преимуществ, но ценой постоянного упоминания имени аргумента. Ruby предоставляет отличный способ справиться с этим — вычисление экземпляра (с использованием метода `instance_eval`).

Вызванный блок Ruby вычисляется в контексте места определения. В частности, любые неквалифицированные функции (или поля) разрешаются в методы и поля объекта, в котором они были определены. Используя `instance_eval`, можно изменить это поведение по умолчанию, указав другой объект для выполнения блока в его контексте (что означает разрешение всех неквалифицированных методов в методы этого объекта). Следующий код демонстрирует сказанное.

```

class StaticContext < Test::Unit::TestCase
  def identify
    return "in static context"
  end
  def test_demo
    o = OtherObject.new
    assert_equal "in static context", o.use_call {identify}
    assert_equal "in other object", o.use_instance_eval {identify}
  end
end

class OtherObject
  def identify
    return "in other object"
  end

```

```

def use_call &arg
  arg.call
end
def use_instance_eval &arg
  instance_eval &arg
end
end

```

По сути, применение `instance_eval` изменяет то, на что ссылается `self` внутри переданного блока.

Эту возможность можно использовать в нескольких построителях с неквалифицированными вызовами методов в сценарии DSL.

```

ComputerBuilder.build do
  processor do
    cores 2
    i386
    speed 2.2
  end
  disk do
    size 150
  end
  disk do
    size 75
    speed 7200
    sata
  end
end

```

Построитель получает блок, как и ранее, но вместо вызова использует `instance_eval`.

```

class ComputerBuilder...
  def self.build &block
    builder = self.new
    builder.instance_eval &block
    return builder.content
  end
  def initialize
    @content = Computer.new
  end

```

При обработке процессора вновь используется `instance_eval`.

```

class ComputerBuilder...
  def processor &block
    @content.processor = ProcessorBuilder.new.build(block)
  end

class ProcessorBuilder
  def build block
    @content = Processor.new
    instance_eval(&block)
    return @content
  end
  def cores arg
    @content.cores = arg
  end
  def i386
    @content.type = :i386
  end
  def speed arg

```

```

    @content.speed = arg
end
end

```

То же самое относится и к дискам.

```

class ComputerBuilder...
def disk &block
  @content.disks << DiskBuilder.new.build(block)
end

class DiskBuilder
def build block
  @content = Disk.new
  instance_eval(&block)
  return @content
end
def size arg
  @content.size = arg
end
def sata
  @content.interface = :sata
end
def speed arg
  @content.speed = arg
end
end
end

```

Показанный здесь способ использования `instance_eval` сценарием DSL типичен для фрагментарного контекста, когда в обычной программе Ruby размещаются небольшие фрагменты DSL. В случае самодостаточного контекста DSL-сценарий может размещаться в отдельном файле, и применение `instance_eval` позволяет избавиться от предварительного и завершающего шума от настройки переноса области видимости в объект ([Object Scoping \(387\)](#)). Весь файл сценария выглядит следующим образом.

```

computer do
  processor do
    cores 2
    i386
    speed 2.2
  end
  disk do
    size 150
  end
  disk do
    size 75
    speed 7200
    sata
  end
end

```

После этого построитель может обработать весь файл с помощью применения к нему `instance_eval`.

```

class ComputerBuilder...
def load_file aFileName
  load(File.readlines(aFileName).join("\n"))
end
def load aStream
  instance_eval aStream
end

```

```
def computer
  yield
end
```

Использование `instance_eval` кажется таким хорошим приемом, что необходимость в передаче явных аргументов замыканию может удивить. Но, как оказалось, имеется вполне реальный вариант — библиотека построителя Джима Вайриха (Jim Weirich). Это очень хорошая библиотека для создания XML-документов с помощью вложенных замыканий и динамического отклика (*Dynamic Reception* (423)). В первой версии библиотеки Джим использовал `instance_eval`, но позже переключился на явные параметры. Причина этого в том, что программисты привыкли к такому поведению замыкания, как у вызовов; переопределение `self` вызывает путаницу и усложняет необходимое обращение к элементам в статическом контексте.

Что касается меня, то все зависит от того, используете ли вы DSL-сценарий в автономном или фрагментарном стиле. В фрагментарном контексте нужно следовать обычным соглашениям по работе с замыканиями, поэтому переопределение `self` посредством `instance_eval` не является хорошим решением. В случае же автономного сценария DSL стиль кода отличается от обычного кода Ruby; поэтому здесь переопределение не вызывает путаницы и вполне может применяться для снижения шума.

Глава 39

Список литералов

Literal List

Представление выражения языка посредством списка литералов

```
martin.follows("WardCunningham", "bigballof mud", "KentBeck", "neal4d");
```

39.1. Как это работает

Список литералов представляет собой конструкцию языка программирования для формирования структуры данных списка. Многие языки программирования предоставляют специальный синтаксис для списка литералов. Наиболее очевидным примером может служить (`first, second, third`) в Lisp или немного менее элегантное `[first, second, third]` в Ruby. Обычно такие структуры допускают существование вложенных списков. По сути, один из способов рассмотрения программы Lisp — трактовать ее как вложенный список.

Списки литералов обычно используются в вызовах функций. После этого функция получает элементы списка и тем или иным образом их обрабатывает.

Основные языки программирования на базе С не предоставляют возможности использовать вложенные списки. Существуют лiteralные массивы `{1, 2, 3}`, но они зачастую ограничены только константами или литералами, в отличие от общего синтаксиса, который допускает любые символы или выражения.

Одним из способов обхода этой проблемы является применение функций с переменным количеством аргументов наподобие `companions(jo, saraJane, leela)`. В строго типизированном языке программирования для корректной работы вызова с переменным количеством элементов эти элементы должны быть одного типа.

39.2. Когда это использовать

Список литералов хорошо работает, если он вложен в другой элемент, чаще всего — в вызов функции, с использованием грамматики наподобие `parent ::= child*`. Часто элементами списка являются вызовы функций, так что список литералов может сделать осуществимыми вложенные функции (Nested Function (361)). Взглянув на соответствующие примеры, можно увидеть, что списки литералов обычно присутствуют в виде функций

с переменным количеством аргументов. (Эти примеры указывают также на некоторые проблемы при таком комбинировании, в частности в случае строгой типизации.)

Даже если ваш базовый язык программирования имеет встроенный синтаксис для списков литералов, при использовании такого списка в вызове функции обычно лучше прибегать к функциям с переменным числом аргументов, т.е. я предпочитаю использовать companions (`jo, saraJane, leela`), а не `companions ([jo, saraJane, leela])`.

Можно сформировать практически любой DSL, используя только списки литералов — по сути, имитируя Lisp. Очевидно, что это естественный путь для Lisp, но не более чем забавное упражнение на других языках программирования, в которых более естественным будет объединение списков с выражениями других видов.

Глава 40

Ассоциативные массивы литералов

Literal Map

*Представление выражения в виде ассоциативных массивов
литералов*

```
computer(processor(:cores => 2, :type => :i386),  
         disk(:size => 150),  
         disk(:size => 75, :speed => 7200, :interface =>  
              :sata))
```

40.1. Как это работает

Ассоциативный массив литералов — это языковая конструкция, присутствующая во многих языках и позволяющая формировать структуру данных отображения⁶ (также известную как словарь или хеш). Он обычно используется в вызове функции, которая принимает его и обрабатывает.

Самая большая проблема при использовании отображений литералов в динамически типизированных языках — отсутствие способа обращения и обеспечения корректных имен для ключей. В дополнение к тому факту, что вам придется самостоятельно писать код для обработки неизвестных ключей, нет механизма для указания автору сценария DSL, какие ключи являются корректными. Статический язык позволяет избежать этой проблемы путем определения ключей как перечисления определенного типа.

В динамически типизированных языках ключи отображения литералов обычно представляют собой символьные данные (или строки). Символы — это естественный выбор, и они легко поддаются обработке. Некоторые языки предлагают синтаксис ярлыков для упрощения применения символьных ключей. Ruby, например, позволяет заменить `{:cores => 2}` на `{cores: 2}`, начиная с версии 1.9.

Так же, как я рассматриваю вызов функции с переменным количеством аргументов как разновидность списка литералов (Literal List (415)), я рассматриваю вызовы функций с аргументами с ключевыми словами как разновидность отображений литералов. Факти-

⁶ Далее мы будем использовать именно этот термин как более общий по сравнению с ассоциативным массивом. — Примеч. ред.

чески аргументы с ключевыми словами даже лучше, так как они часто позволяют самостоятельно определять корректные ключевые слова. К сожалению, такие аргументы встречаются еще реже, чем синтаксис отображений литералов.

Если у вас есть синтаксис списка литералов, но не отображений, то для представления отображений можно использовать списки — то, что Lisp делает с помощью выражений наподобие `(processor (:cores 2) (:type :i386))`. В других языках этого можно достичь с помощью конструкций типа `processor("cores", 2, "type" "i386")`, рассматривая аргументы как чередующиеся ключи и значения.

Некоторые языки, например Ruby, позволяют опустить разделители для отображений литералов при использовании только одного из них в определенном контексте. Так, строку `processor({:cores=>2, :type=>:i386})` можно записать более кратко: `processor(:cores=>2, :type=>:i386)`.

40.2. Когда это использовать

Отображение литералов представляет собой отличный выбор, когда вам требуется список различных элементов, в котором каждый элемент должен появиться не более одного раза. Конечно, распространенное отсутствие проверки корректности имен ключей раздражает, но в целом такой синтаксис обычно оказывается наиболее подходящим для такого случая. При этом ясно указывается, что каждый подэлемент может появляться не более одного раза, а отображения литералов идеально подходят для обработки вызываемой функцией.

Если у вас нет отображений литералов, можете обойтись списками литералов ([Literal List \(415\)](#)) или использовать вложенные функции ([Nested Function \(361\)](#)) или соединения методов в цепочки ([Method Chaining \(375\)](#)).

40.3. Настройка конфигурации компьютера с помощью списков и отображений (Ruby)

Следуя своей традиции языка сценариев, Ruby предоставляет неплохой синтаксис для списков и отображений литералов. Вот как можно использовать этот синтаксис для настройки конфигурации компьютера.

```
computer(processor(:cores => 2, :type => :i386),
          disk(:size => 150),
          disk(:size => 75, :speed => 7200, :interface => :sata))
```

Здесь используется не только отображение литералов, но и другие технологии. Имеются три функции — `computer`, `processor` и `disk`. Каждая из них получает в качестве аргумента коллекцию: `computer` получает список литералов ([Literal List \(415\)](#)), прочие получают отображение литералов. Я использую перенос области видимости в объект ([Object Scoping \(387\)](#)) класса построителя, который реализует функции. Поскольку это Ruby, я могу применить `instance_eval` для вычисления сценария DSL в контексте экземпляра построителя, что избавит меня от необходимости создавать подкласс.

Начнем с метода `processor`.

```
class MixedLiteralBuilder...
  def processor map
    check_keys map, [:cores, :type]
```

```
    return Processor.new(map[:cores], map[:type])
end
```

Применять отображение литералов просто: я выбираю необходимые элементы из отображения с помощью ключей. Опасность использования отображений наподобие данного заключается в том, что вызывающая функция может случайно указать неверный ключ, так что небольшая проверка не помешает.

```
class MixedLiteralBuilder...
  def check_keys map, validKeys
    bad_keys = map.keys - validKeys
    raise IncorrectKeyException.new(bad_keys)
      unless bad_keys.empty?
  end

  class IncorrectKeyException < Exception
    def initialize bad_keys
      @bad_keys = bad_keys
    end
    def to_s
      "unrecognized keys: #{@bad_keys.join(', ')}"
    end
  end
end
```

Тот же подход используется и для диска.

```
class MixedLiteralBuilder...
  def disk map
    check_keys map, [:size, :speed, :interface]
    return Disk.new(map[:size], map[:speed], map[:interface])
  end
```

Поскольку здесь все значения простые, можно создать объект предметной области и возвращать его в каждой вложенной функции (*Nested Function* (361)). Функция computer может создавать объект компьютера с помощью переменного количества аргументов для нескольких дисков.

```
class MixedLiteralBuilder...
  def computer proc, *disks
    @result = Computer.new(proc, *disks)
  end
```

(* в списке аргументов позволяет использовать переменное количество аргументов в Ruby. В данном случае я могу обращаться ко всем дискам как к массиву disks. Если я вызову другую функцию с *disks, то элементы массива disks будут переданы в нее как отдельные аргументы.)

Для обработки сценария DSL построитель использует `instance_eval`.

```
class MixedLiteralBuilder...
  def load aStream
    instance_eval aStream
  end
```

40.4. Имитация Lisp (Ruby)

Хороший DSL использует одновременно несколько различных технологий. Так, в предыдущем примере наряду с отображениями литералов применялись вложенные

функции ([Nested Function \(361\)](#)) и списки литералов ([Literal List \(415\)](#)). Иногда, однако, интересно воспользоваться только одной методикой, чтобы ощутить границы ее возможностей. Вполне реально написать довольно сложное DSL-выражение с помощью только списков и отображений литералов. Это может выглядеть, например, так.

```
[:computer,
 [:processor, {:cores => 2, :type => :i386}],
 [:disk, {:size => 150}],
 [:disk, {:size => 75, :speed => 7200, :interface => :sata}]
]
```

В этой версии я заменил все вызовы функций списками литералов, в которых первый элемент списка представляет собой имя обрабатываемого элемента, а остальная часть списка содержит аргументы. Этот массив можно обработать путем простого вычисления кода Ruby и передачи его методу, который интерпретирует получившееся выражение.

```
class LiteralOnlyBuilder...
  def load aStream
    @result = handle_computer(eval(aStream))
  end
```

Я обрабатываю каждое выражение путем проверки первого элемента массива и последующей обработки остальных элементов.

```
class LiteralOnlyBuilder...
  def handle_computer anArray
    check_head :computer, anArray
    processor = handle_processor(anArray[1])
    disks = anArray[2..-1].map{|e| handle_disk e}
    return Computer.new(processor, *disks)
  end
  def check_head expected, array
    raise "error: expected #{expected}, got #{array.first}"
      unless array.first == expected
  end
```

Это, по сути, следование форме синтаксического анализатора рекурсивного спуска ([Recursive Descent Parser \(253\)](#)). Правило для компьютера гласит, что он имеет процессор и несколько дисков, и я вызываю метод для его обработки, возвращая вновь создаваемый компьютер.

Обработка процессора проста — нужно взять аргументы из переданного отображения.

```
class LiteralOnlyBuilder...
  def handle_processor anArray
    check_head :processor, anArray
    check_arg_keys anArray, [:cores, :type]
    args = anArray[1]
    return Processor.new(args[:cores], args[:type])
  end
  def check_arg_keys array, validKeys
    bad_keys = array[1].keys - validKeys
    raise IncorrectKeyException.new(bad_keys)
      unless bad_keys.empty?
  end
```

Точно так же выглядит и обработка дисков.

```
class LiteralOnlyBuilder...
  def handle_disk anArray
    check_head :disk, anArray
```

```

check_arg_keys anArray, [:size, :speed, :interface]
args = anArray[1]
return Disk.new(args[:size], args[:speed],
               args[:interface])
end

```

Говоря о таком подходе, следует отметить, что он дает полный контроль над порядком вычисления элементов языка. Здесь выражения для процессора и дисков вычисляются до создания объекта компьютера, но я могу поступать и иначе — так, как захочу. Во многом этот сценарий DSL подобен внешнему DSL, закодированному не с помощью строк, а с помощью синтаксиса коллекции литералов.

Здесь смешаны списки и отображения, но можно ограничиться только списками литералов, которые можно смело назвать имитацией Lisp.

```

[:computer,
 [:processor,
  [:cores, 2,],
  [:type, :i386]],
 [:disk,
  [:size, 150]],
 [:disk,
  [:size, 75],
  [:speed, 7200],
  [:interface, :sata]]]

```

(Надеюсь, вы сообразите, почему я называю этот код имитацией Lisp.)

Все, что я реально здесь делаю, — это заменяю каждое отображение списком двухэлементных подсписков, состоящих из ключа и значения.

Основной загружающий код остается тем же, разбивающим символьное выражение для компьютера на выражения процессора и нескольких дисков.

```

class ListOnlyBuilder...
def load aStream
  @result = handle_computer(eval(aStream))
end
def handle_computer SEXP
  check_head :computer, SEXP
  processor = handle_processor(SEXP[1])
  disks = SEXP[2...-1].map{|e| handle_disk e}
  return Computer.new(processor, *disks)
end

```

Отличие наблюдается в подклассах, где требуется некоторый дополнительный код, эквивалентный просмотру отображений.

```

class ListOnlyBuilder...
def handle_processor SEXP
  check_head :processor, SEXP
  check_arg_keys SEXP, [:cores, :type]
  return Processor.new(select_arg(:cores, SEXP),
                       select_arg(:type, SEXP))
end
def handle_disk SEXP
  check_head :disk, SEXP
  check_arg_keys SEXP, [:size, :speed, :interface]
  return Disk.new(select_arg(:size, SEXP),
                 select_arg(:speed, SEXP),
                 select_arg(:interface, SEXP))
end
def select_arg key, list

```

```
assoc = list.tail.assoc(key)
return assoc ? assoc[1] : nil
end
```

Использование одних лишь списков действительно приводит к более аккуратному сценарию DSL, но список пар в качестве отображения не соответствует стилю Ruby. В любом случае приведенный ранее пример, в котором использовались вызовы функций наряду с коллекциями литералов, существенно точнее соответствует этому стилю.

Однако подход с вложенными списками приводит нас в другое место, где этот стиль выглядит естественно. Как многие читатели уже давно поняли, именно так выглядят Lisp. В этом языке программирования данный сценарий DSL может выглядеть следующим образом.

```
(computer
  (processor
    (cores 2)
    (type i386))
  (disk
    (size 150))
  (disk
    (size 75)
    (speed 7200)
    (interface sata)))
```

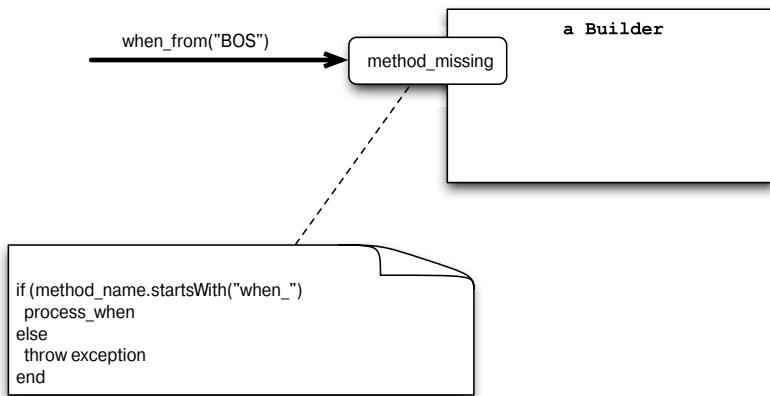
Структура списка гораздо естественнее и понятнее в Lisp. Слова по умолчанию являются символами, а поскольку выражения представляют собой либо атомы, либо списки, нет необходимости в запятых.

Глава 41

Динамический отклик

Dynamic Reception

Обработка сообщений без их определения в получающем классе



Также известен как перекрытие method_missing или doesNotUnderstand

Объект имеет ограниченное количество определенных для него методов. Клиент объекта может попытаться вызвать метод, который в получателе не определен. Статически типизированный язык программирования выявит ошибки во время компиляции и сообщит о них. В результате получить такую ошибку во время выполнения программы невозможно (если не предпринять специальных шагов, чтобы обойти систему типов). В случае динамически типизированного языка программирования можно вызвать несуществующий метод во время выполнения, что обычно приводит к сообщению об ошибке времени выполнения.

Динамический отклик позволяет изменить это поведение, т.е. появляется возможность по-разному отвечать на неизвестные сообщения.

41.1. Как это работает

Многие динамические языки программирования реагируют на вызов неизвестных методов вызовом специального метода обработки ошибок на вершине объектной иерархии. У такого метода нет стандартного имени: в Smalltalk это `doesNotUnderstand`, в Ruby — `method_missing`. Вы можете выполнить собственную обработку неизвестного метода в своем классе. Делая это, вы, по сути, динамически заменяете правила отклика на вызовы методов.

Есть много причин, по которым динамический отклик полезен в программировании в целом. Одним из отличных примеров является поддержка автоматического делегирования другому объекту. Для этого необходимо определить методы, которые вы хотите обрабатывать в исходном приемнике, и использовать динамический отклик для перенаправления всех неизвестных сообщений объекту делегата.

Динамический отклик может применяться в работе DSL по-разному. Один из вариантов использования заключается в преобразовании параметра метода в имя. Хорошим примером этого является динамический поиск в Rails's Active Record. Скажем, у вас есть класс `Person` со свойствами `firstname` и `lastname`. При этом вы можете вызывать `find_by_firstname("martin")` или `find_by_firstname_and_lastname("martin", "fowler")` без явного определения этих методов. Код суперкласса Active Record переопределяет `method_missing` Ruby и проверяет, не начинается ли имя вызываемого метода с `find_by`. Если это так, то он анализирует имя метода, чтобы найти имена свойств, и использует их для построения запросов. Это можно сделать и путем передачи нескольких аргументов, например `find_by("firstname", "martin", "lastname", "fowler")`, но размещение имен свойств в названии метода более удобочитаемо, так как указывает, что бы вы стали делать, если бы такие методы были явно определены.

Метод наподобие `find_by_name` работает, получая имя метода и анализируя его. По сути, вы вкладываете внешний DSL в имя метода. Другой подход состоит в использовании последовательности динамических откликов типа `find_by.firstname("martin").and.lastname("fowler")` или `find_by.firstname.martin.and.lastname.fowler`. В этом случае метод `find_by` вернет построитель выражений (Expression Builder (349)), который можно использовать для создания запроса с помощью соединения методов в цепочки (Method Chaining (375)) и динамического отклика.

Одно из преимуществ этого подхода заключается в избежании размещения различных параметров в кавычках, используя `martin` вместо `"martin"` для снижения шума. Если вы используете перенос области видимости в объект (Object Scoping (387)), то в качестве аргументов можете использовать неквалифицированные символы, например `state :idle` вместо `state :idle`. Это можно сделать путем реализации динамического отклика в суперклассе, так что после вызова метода `state` объекта он перекрывает следующий вызов неизвестного метода для получения имени состояния. Вы можете пойти еще дальше, использовав шлифовку текста (Textual Polishing (469)) для удаления фрагментов “шумной пунктуации”.

41.2. Когда это использовать

Использование динамического отклика для перемещения параметров в имена методов привлекательно по нескольким причинам. Во-первых, он может имитировать то, что

вы делаете с помощью исходного метода, но с меньшими усилиями. Вполне можно представить себе класс человека с методом `find_by_firstname_and_lastname`; применяя динамический отклик, вы предоставляете такой метод, на самом деле его не программируя. Это может значительно сэкономить время, особенно если вы используете много разных комбинаций. Есть, конечно, и другие способы достижения того же. Можно, например, поместить имя атрибута в параметры, как в `find(:firstname, "martin", :lastname, "fowler")`, использовать замыкания, как в `find{|p| p.firstname == "martin" and p.lastname == "fowler"}`, или даже фрагментарно вставить внешний DSL в строку, как в `find("firstname==martin lastname==fowler")`. Однако многие считают, что вложения имен полей в имена методов — наиболее свободный способ выражения вызова.

Еще одним преимуществом замены параметров именами методов является то, что она может обеспечить лучшую согласованность в пунктуации. Выражение наподобие `find.by.firstname.martin.and.lastname.fowler` использует точки в качестве единственного знака пунктуации. Это хорошо, так как при этом нельзя перепутать, когда следует использовать точку, а когда пару скобок или кавычек. Для многих эта согласованность не является такой уж важной. Лично я предпочитаю отделение схемы от данных, поэтому мне больше нравится запись наподобие `find_by.firstname("martin").and.lastname("fowler")`, где имена полей помещены в вызовы методов, а данные — в параметры.

Одна из проблем, связанных с внесением данных в имена методов, заключается в том, что часто языки программирования используют различные кодировки для текста программы и для строковых данных; многие языки программирования допускают только символы ASCII, так что метод не будет работать для не ASCII-имен. Аналогично правила грамматики языка программирования для имен методов могут не допускать применение корректных имен персон.

Прежде всего, важно помнить о том, что динамический отклик окупается только тогда, когда позволяет создавать эти структуры в общем, без какой-либо специальной обработки частных случаев. Это означает, что данный метод стоит применять только тогда, когда можно явно транслировать динамические методы в методы, необходимые для других целей. Хорошими примерами являются условия, потому что они обычно вызывают атрибуты объектов модели предметной области. Метод `find_by_firstname_and_lastname` эффективен потому, что имеется класс `Person` с атрибутами `firstname` и `lastname`. Если нужно писать специальные методы для обработки конкретных случаев динамического отклика, это обычно означает, что от динамического отклика лучше отказаться.

Динамический отклик сопровождается большим количеством проблем и ограничений. Самое большое из них — отсутствие возможности воспользоваться этой методикой в статических языках программирования. Но даже в динамическом языке нужно быть очень осторожными при ее использовании. После переопределения обработчика вызовов неизвестных методов любая ошибка может завести вас в дебри отладки. Стек часто становится совершенно непроницаемым для трассировки.

Существуют и ограничения, накладываемые на то, что можно таким образом выразить. Обычно нельзя написать что-то наподобие `find_by.age.greater_than.2`, так как большинство динамических языков программирования не допускает имя метода “2”. Поэтому приходится прибегать к обходным путям наподобие `find_by.age.greater_than.n2`, но это существенно снижает свободу, ради которой все и затевалось.

Поскольку здесь я рассматриваю, в первую очередь, логические выражения, я должен также указать, что такой способ составления вызовов методов не слишком хорош для по-

строения сложных логических условий. Этот подход вполне работает для чего-то простого наподобие `find_by.firstname("martin").and.lastname("fowler")`, но как только вы перейдете к инструкциям вида `find_by.firstname.like("m*").and.age.greater_than(40).and.not.employer.like("thought*")`, вы тут же пуститесь по дороге, которая приведет вас к созданию сложного синтаксического анализатора в неподходящей для этого среде.

Тот факт, что выражения с использованием динамического отклика плохо работают для сложных условных выражений, не является причиной избегать их для простых случаев. Active Record использует динамический отклик для обеспечения динамического поиска в простых случаях, но сознательно не поддерживает более сложные выражения, поощряя вместо этого использовать другой механизм. Не всем нравится такая непоследовательность, но я думаю, что для разного уровня сложности могут лучше всего подходить различные решения, так что наличие нескольких путей решения одной задачи — это скорее плюс, чем минус.

41.3. Расчет бонусов с помощью анализа имен методов (Ruby)

Рассмотрим в этом примере схему, которая назначает бонусные баллы за те или иные маршруты путешествий. Нашей моделью предметной области является маршрут из элементов, которые могут представлять собой такие действия, как полет, проживание в гостинице, аренда автомобиля и т.д. Мы хотим обеспечить гибкий способ подсчета баллов для часто путешествующих людей, наподобие 300 баллов за перелет из Бостона.

Я покажу, как можно воспользоваться для этого динамическим откликом. Начнем с простого случая одного правила начисления бонусов.

```
@builder = PromotionBuilder.new
@builder.score(300).when_from("BOS")
```

В следующем случае имеется несколько правил начисления бонусов, соответствующих различным видам элементов. Вот как рассчитываются баллы для вылета из определенного аэропорта и для остановки в отеле определенной компании в пределах одного и того же путешествия.

```
@builder = PromotionBuilder.new
@builder.score(350).when_from("BOS")
@builder.score(100).when_brand("hyatt")
```

И наконец, перед вами — составное правило начисления баллов за полет из Бостона рейсом определенной авиакомпании.

```
@builder = PromotionBuilder.new
@builder.score(140).when_from_and_airline("BOS", "NW")
```

41.3.1. Модель

Модель состоит из двух частей: маршрутов и бонусов. Маршрут — это простая коллекция элементов, которые могут представлять собой что угодно. В данном простом примере это только авиарейсы и отели.

```
class Itinerary
  def initialize
    @items = []
  end
  def << arg
```

```

    @items << arg
  end
  def items
    return @items.dup
  end
end

class Flight
  attr_reader :from, :to, :airline
  def initialize airline, from, to
    @from, @to, @airline = from, to, airline
  end
end

class Hotel
  attr_accessor :nights, :brand
  def initialize brand, nights
    @nights, @brand = nights, brand
  end
end

```

Бонусы представляют собой набор правил, каждое из которых содержит начисляемые баллы и список условий.

```

class Itinerary...
  def initialize rules
    @rules = rules
  end

class PromotionRule...
  def initialize anInteger
    @score = anInteger
    @conditions = []
  end
  def add_condition aPromotionCondition
    @conditions << aPromotionCondition
  end

```

Подсчет баллов за маршрут осуществляется путем суммирования баллов, насчитанных каждым из правил.

```

class Itinerary...
  def score_of anItinerary
    return @rules.inject(0) { |sum, r| sum += r.score_of(anItinerary) }
  end

```

Правило расчета баллов для маршрута проверяет, все ли его условия соответствуют маршруту, и если все, то возвращаются начисленные баллы.

```

class PromotionRule...
  def score_of anItinerary
    return (@conditions.all?{|c| c.match(anItinerary)} ) ?
      @score : 0
  end

```

Каждая строка `score` в DSL представляет собой отдельное правило. Так,

```

@builder = PromotionBuilder.new
@builder.score(350).when_from("BOS")
@builder.score(100).when_brand("hyatt")

```

представляет собой одно начисление баллов с двумя правилами. Маршруту может соответствовать как одно из них, так и они оба. В противоположность этому

```
@builder = PromotionBuilder.new
@builder.score(140).when_from_and_airline("BOS", "NW")
```

представляет собой одно правило с двумя условиями. Для начисления баллов должны быть выполнены оба условия.

Для обработки этой ситуации у меня есть объект условия эквивалентности, для которого я могу указать соответствующие имена и значения.

```
class EqualityCondition
  def initialize aSymbol, value
    @attribute, @value = aSymbol, value
  end
  def match anItinerary
    return anItinerary.items.any?{|i| match_item i}
  end
  def match_item anItem
    return false unless anItem.respond_to?(@attribute)
    return @value == anItem.send(@attribute)
  end
end
```

Использование условий равенства в имени метода весьма ограничено. Однако базовая модель позволяет иметь любые виды условий — лишь бы было известно, как выполнять проверку соответствия маршруту. Одни из этих условий могут быть добавлены с помощью DSL, другие — иными средствами, такими как замыкания.

Пример.....

```
rule = PromotionRule.newWithBlock(520) do |itinerary|
  flights = itinerary.items.select{|i| i.kind_of? Flight}
  flights.any?{|f| f.from == "LAX"} and
  flights.any?{|f| f.to == "LAX"} and
  flights.all?{|f| %w[NW CO DL].include?(f.airline)}
end
promotion = Promotion.new([rule])

class BlockCondition
  def initialize aBlock
    @block = aBlock
  end
  def match anItinerary
    @block.call(anItinerary)
  end
end
```

Такая разновидность гибкости может оказаться весьма важной. Этот подход позволяет использовать DSL для обработки простых случаев и предоставляет механизм для обработки более сложных ситуаций.

41.3.2. Постройтель

Базовый построитель включает создающую его коллекцию правил начисления и при необходимости возвращает новый бонусный объект.

```
class PromotionBuilder...
  def initialize
    @rules = []
  end
  def content
    return Promotion.new(@rules)
  end
```

Метод `score` создает одно из этих правил, которое он хранит в переменной контекста (`Context Variable` (187)). Он также создает отдельный построитель для условия.

```
class PromotionBuilder...
  def score anInteger
    @rules << PromotionRule.new(anInteger)
    return PromotionConditionBuilder.new(self)
  end
```

Построитель условия представляет собой класс, который использует динамический отклик. В Ruby динамический отклик осуществляется путем переопределения `method_missing`.

```
class PromotionConditionBuilder...
  def initialize parent
    @parent = parent
  end
  def method_missing(method_id, *args)
    if match = /^when_(\w*)/.match(method_id.to_s)
      process_when match.captures.last, *args
    else
      super
    end
  end
```

Метод `method_missing` проверяет, не начинается ли имя метода с `when_`; если нет — вызов передается суперклассу, который генерирует исключение. В предположении корректного вызова метода из него извлекаются имена атрибутов, проверяется их соответствие аргументам, а затем создаются соответствующие правила.

```
class PromotionConditionBuilder...
  def process_when method_tail, *args
    attribute_names = method_tail.split('_and_')
    check_number_of_attributes(attribute_names, args)
    populate_rules(attribute_names, args)
  end
  def check_number_of_attributes(names, values)
    unless names.size == values.size
      throw "There are %d attribute names but %d arguments" %
        [names.size, values.size]
    end
  end
  def populate_rules names, args
    names.zip(args).each do |name, value|
      @parent.add_condition(EqualityCondition.new(name, value))
    end
  end
```

Этот подход похож на динамические искатели Active Record. Если вам это интересно, можете обратиться к описанию Джамиш Бак (Jamis Buck) по адресу <http://weblog.jamisbuck.org/2006/12/1/under-the-hood-activerecord-base-find-part-3>.

41.4. Расчет бонусов с помощью цепочек вызовов (Ruby)

Теперь я возьму тот же пример и поработаю с ним с помощью цепочек вызовов. Я использую ту же модель и (почти) те же условия примеров. Поскольку DSL теперь иной, иначе формулируются и условия. Вот простой выбор полетов из Бостона.

```
@builder.score(300).when.from.equals.BOS
```

В этом случае я передаю все аргументы в условие как методы, а не как параметры (за исключением баллов, которые остаются параметром, хотя это и не согласуется со всем остальным). Я также указываю в виде метода оператор условия.

Вот случай с двумя отдельными начислениями баллов.

```
@builder.score(350).when.from.equals.BOS
@builder.score(100).when.brand.equals.hyatt
```

А вот и составное условие.

```
@builder.score(170).when.from.equals.BOS.and.nights.at.least._3
```

Составное условие сложнее, чем использованное в предыдущем примере. В данном случае я пользуюсь возможностью применения других операторов, а также демонстрирую способ передачи числового параметра в виде имени метода.

41.4.1. Модель

Модель практически идентична модели из предыдущего примера. Единственное отличие заключается в добавлении дополнительного условия.

```
class AtLeastCondition...
  def initialize aSymbol, value
    @attribute, @value = aSymbol, value
  end
  def match anItinerary
    return anItinerary.items.any?{|i| match_item i}
  end
  def match_item anItem
    return false unless anItem.respond_to?(@attribute)
    return @value <= anItem.send(@attribute)
  end
end
```

41.4.2. Построитель

Основное отличие от предыдущего примера — в построителе. Как и ранее, у меня есть объект построителя, хранящий набор правил и при необходимости производящий бонусный объект.

```
class PromotionBuilder...
  def initialize
    @rules = []
  end
  def content
    return Promotion.new(@rules)
  end
```

Метод `score` добавляет правило в список.

```
class PromotionBuilder...
  def score anInteger
    @rules << PromotionRule.new(anInteger)
    return self
  end
```

Метод `when` возвращает более конкретный построитель для имени атрибута.

```

class PromotionBuilder...
  def when
    return ConditionAttributeBuilder.new(self)
  end

class ConditionAttributeBuilder < Builder
  def initialize parent
    @parent = PromotionConditionBuilder.new(parent)
    @parent.name = self
  end

class Builder
  attr_accessor :content, :parent
  def initialize parentBuilder = nil
    @parent = parentBuilder
  end
end

class PromotionConditionBuilder < Builder
  attr_accessor :name, :operator, :value

```

Для построения условия я создаю небольшое синтаксическое дерево. Каждое условие в выражении состоит из трех частей: имени, оператора и условия. Так что я создаю построитель для каждой из этих частей, так же как и родительский построитель для связывания условий. В результате, создавая построитель имени, я создаю и родитель построителя условия для подготовки дерева.

Постройтель имени атрибута будет искать подходящее имя тестируемого атрибута, так как это имя будет изменяться в зависимости от атрибутов класса модели. Я использую динамический отклик.

```

class ConditionAttributeBuilder...
  def method_missing method_id, *args
    @content = method_id.to_s
    return ConditionOperatorBuilder.new(@parent)
  end

```

Здесь получается имя и возвращается построитель оператора.

Постройтель оператора будет иметь дело только с фиксированным набором операторов, так что здесь можно обойтись без динамического отклика.

```

class ConditionOperatorBuilder < Builder
  attr_reader :condition_class
  def initialize parent
    super
    @parent.operator = self
  end
  def equals
    @content = EqualityCondition
    return next_builder
  end
  def at
    return self
  end
  def least
    @content = AtLeastCondition
    return next_builder
  end
  def next_builder
    return ConditionValueBuilder.new(@parent)
  end

```

Базовое поведение построителя оператора аналогично поведению построителя имени: получить оператор и вернуть новый построитель для последней части (значения). Здесь имеется пара интересных моментов. Во-первых, содержимое этого построителя представляет собой соответствующий класс условия из модели. Во-вторых, метод `at` возвращает просто сам себя — он применяется исключительно для того, чтобы сделать выражение более удобочитаемым.

Окончательный построитель представляет собой построитель значения, который получает значение с помощью динамического отклика.

```
class ConditionValueBuilder < Builder
  def initialize parent
    super
    @parent.value = self
  end
  def method_missing method_id, *args
    @content = method_id.to_s
    @content = @content.to_i if @content =~ /^_\d+/
    @parent.end_condition
  end
end
```

Если значение представляет собой число, придется прибегнуть к хитрости, так как в сценарии DSL для представления числа используется ведущий символ подчеркивания — “`_3`” для представления “3”. (В Ruby `“_3”.to_i` будет переводить строку в целое число, игнорируя ведущее подчеркивание и возвращая значение 3.)

Этот метод, помимо всего прочего, завершает часть выражения, поэтому он должен указать своему родителю на необходимость наполнения модели.

```
class PromotionConditionBuilder...
  def end_condition
    content = @operator.build_content(@name.content, @value.content)
    @parent.add_condition content
    return @parent
  end

class ConditionOperatorBuilder...
  def build_content name, value
    return @content.new(name, value)
  end

class PromotionBuilder...
  def add_condition cond
    current_rule.add_condition cond
  end
  def current_rule
    @rules.last
  end
```

В этот момент я использую наше небольшое синтаксическое дерево и создаю объект условия в модели. В случае составного условия я повторяю этот процесс.

```
class PromotionBuilder...
  def and
    return ConditionAttributeBuilder.new(self)
  end
```

Создание небольших синтаксических деревьев, подобных описанному, — не самый распространенный способ работы внутренних DSL; обычно проще строить модель по хо-

ду работы над сценарием. Однако в случае условных выражений наподобие приведенных выше построение синтаксического дерева может иметь смысл.

Впрочем, в целом я не в восторге от создания выражений с помощью этого подхода. Мне кажется, что как только вы начнете синтаксический анализ последовательности вызовов методов, подобной данной, вы сможете так же просто переключиться на внешний DSL, обеспечивающий большую гибкость. Желание строить синтаксические деревья является признаком того, что внутренний DSL выполняет слишком много работы.

41.5. Уменьшение шума в коде контроллера тайника (JRuby)

Во введении я приводил пример применения Ruby в качестве внутреннего DSL для контроллера тайника. Соответствующий код имел следующий вид.

```
event :doorClosed,      "D1CL"
event :drawerOpened,   "D2OP"
event :lightOn,         "L1ON"
event :doorOpened,     "D1OP"
event :panelClosed,    "PNCL"

command :unlockPanel,  "PNUL"
command :lockPanel,    "PNLK"
command :lockDoor,     "D1LK"
command :unlockDoor,   "D1UL"

resetEvents :doorOpened

state :idle do
  actions :unlockDoor, :lockPanel
  transitions :doorClosed => :active
end

state :active do
  transitions :drawerOpened => :waitingForLight,
               :lightOn => :waitingForDrawer
end

state :waitingForLight do
  transitions :lightOn => :unlockedPanel
end

state :waitingForDrawer do
  transitions :drawerOpened => :unlockedPanel
end

state :unlockedPanel do
  actions :unlockPanel, :lockDoor
  transitions :panelClosed => :idle
end
```

В коде этого примера динамический отклик не используется; применяются только простые вызовы функций. Одним из недостатков данного сценария является, в частности, применение маркера символа Ruby (начальное “:`” в именах). По сравнению с внешним DSL это выглядит шумом. Избавиться от него можно с помощью динамического отклика, получив сценарий наподобие следующего.

```
events do
  doorClosed      "D1CL"
```

```

drawerOpened "D2OP"
lightOn      "L1ON"
doorOpened   "D1OP"
panelClosed  "PNCL"
end

commands do
  unlockPanel "PNUL"
  lockPanel   "PNLK"
  lockDoor    "D1LK"
  unlockDoor  "D1UL"
end

reset_events do
  doorOpened
end

state.idle do
  actions.unlockDoor.lockPanel
  doorClosed.to.active
end

state.active do
  drawerOpened.to.waitingForLight
  lightOn.to.waitingForDrawer
end

state.waitingForLight do
  lightOn.to.unlockedPanel
end

state.waitingForDrawer do
  drawerOpened.to.unlockedPanel
end

state.unlockedPanel do
  panelClosed.to.idle
  actions.unlockPanel.lockDoor
end

```

Стартовой точкой реализации является класс построителя конечного автомата. Этот класс переносит область видимости в объект ([Object Scoping \(387\)](#)) с помощью `instance_eval`. Построение осуществляется в два этапа: сначала выполняется сценарий, а затем — некоторые дополнительные действия.

```

class StateMachineBuilder...
  attr_reader :machine
  def initialize
    @states = {}
    @events = {}
    @commands = {}
    @state_blocks = {}
    @reset_events = []
  end
  def load aString
    instance_eval aString
    build_machine
    return self
  end

```

Для вычисления сценария в построителе есть методы, которые соответствуют основным предложением сценария DSL. Я использую здесь ту же семантическую модель (*Semantic Model* (171)), что и во введении; построитель JRuby наполняет объекты Java.

Сначала выполняется просмотр объявлений событий, что я делаю путем вызова метода `events` построителя конечного автомата и передачи блока, содержащего отдельные объявления событий.

```
class StateMachineBuilder...
  def events &block
    EventBuilder.new(self).instance_eval(&block)
    self
  end
  def add_event name, code
    @events[name] = Event.new(name.to_s, code)
  end

  class EventBuilder < Builder
    def method_missing name, *args
      @parent.add_event(name, args[0])
    end
  end

  class Builder
    def initialize parent
      @parent = parent
    end
  end
end
```

Метод `events` вычисляет блок непосредственно в контексте отдельного построителя, который использует динамический отклик для обработки каждого вызова метода, как объявления события. Для каждого такого объявления я создаю из семантической модели событие и помещаю его в таблицу символов (*Symbol Table* (177)).

Я использую те же базовые методики для команд и сбрасывающих событий. Используя разные построители, я могу поддерживать их относительно простыми, с точно определенной областью ответственности, определяющей, что именно может распознавать данный построитель.

Объявление состояния более интересно. Я вновь использую замыкания для тела объявления, но здесь имеется несколько отличий. Очевидным отличием является то, что я указываю имя с помощью динамического отклика.

```
class StateMachineBuilder...
  def state
    return StateNameBuilder.new(self)
  end
  def addState name, block
    @states[name] = State.new(name.to_s)
    @state_blocks[name] = block
    @start_state ||= @states[name]
  end

  class StateNameBuilder < Builder
    def method_missing name, *args, &block
      @parent.addState(name, block)
      return @parent
    end
  end
end
```

Второе отличие заключается в реализации. Вместо немедленного вычисления вложенного замыкания (*Nested Closure* (403)) я откладываю его в отображение. Откладывая вычисление, я могу не беспокоиться об опережающих ссылках на состояния. Я могу подождать с работой с телами состояний до того момента, пока не будут объявлены все состояния и таблица символов не будет заполнена.

Последнее отличие состоит в том, что я рассматриваю первое состояние сценария как стартовое, для чего использую дополнительную переменную. Значение в нее вносится только в том случае, если в этот момент она пуста. Так гарантируется, что она будет содержать именно первое указанное в сценарии состояние.

Вычисление сценария завершается заполнением всех этих данных. Затем начинается второй этап обработки.

```
class StateMachineBuilder...
  def build_machine
    @state_blocks.each do |key, value|
      if value
        sb = StateBodyBuilder.new(self, @states[key])
        sb.instance_eval(&value)
      end
    end
    @machine = StateMachine.new(@start_state)
    @machine.addResetEvents(
      @reset_events.
      collect{|e| @events[e]}.
      to_java("gothic.model.Event"))
  end

class StateBodyBuilder < Builder
  def initialize parent, state
    super parent
    @state = state
  end
```

Первый шаг этой постобработки состоит в отложенном вычислении тел объявлений состояний — вновь с созданием конкретных построителей и вычислением блоков с помощью `instance_eval`.

Тело может содержать два вида инструкций: действия и переходы. Действия обрабатываются своим методом.

```
class StateBodyBuilder...
  def actions
    return ActionListBuilder.new(self)
  end
  def add_action name
    @state.addAction(@parent.command_at(name))
  end

class ActionListBuilder < Builder
  def method_missing name, *args
    @parent.add_action name
    return self
  end
end

class StateMachineBuilder...
  def command_at name
    return @commands[name]
  end
```

Метод `actions` создает другой построитель, который принимает все вызовы методов как имена команд. Это позволяет указать в одной строке несколько действий, соединенных в цепочку.

В то время как действия используют специальный метод, аналогичный ключевому слову во внешнем DSL, переходы используют динамический отклик.

```
class StateBodyBuilder...
  def method_missing name, *args
    return TransitionBuilder.new(self, name)
  end
  def add_transition event, target
    @state.addTransition(@parent.event_at(event),
                         @parent.state_at(target))
  end

class TransitionBuilder < Builder
  def initialize parent, event
    super parent
    @event = event
  end
  def to
    return self
  end
  def method_missing name, *args
    @target = name
    @parent.add_transition @event, @target
    return @parent
  end
end

class StateMachineBuilder...
  def event_at name
    return @events[name]
  end
  def state_at name
    return @states[name]
  end
```

Здесь неизвестный метод используется для того, чтобы конкретный построитель получал целевое состояние с помощью динамического отклика. Метод `to` используется исключительно для повышения удобочитаемости.

Все это позволяет избавиться от символов “`:`”. Конечно, большой вопрос — стоит ли это вообще делать. На мой взгляд, списки событий и команд получились неплохо. Что же касается состояний, то я доволен ими в меньшей степени. Можно, конечно, использовать гибридный подход с динамическим откликом для того, что мне нравится, и символьные ссылки там, где динамический отклик помочь не в состоянии. Комбинирование методов вообще часто оказывается наилучшим выбором.

Несмотря на то что нам удалось избавиться от символа “`:`”, у нас до сих пор есть кавычки вокруг команд и кодов событий. Для того чтобы избавиться и от них, также можно было бы применить подобную методику.

Глава 43

Аннотации

Annotation

*Данные о программных элементах (таких, как классы и методы),
которые могут обрабатываться в процессе компиляции
и выполнения*

```
@ValidRange(lower = 1, upper = 1000, units = Units.LB)
private Quantity weight;
@ValidRange(lower = 1, upper = 120, units = Units.IN)
private Quantity height;
```

Мы привыкли к классификации данных в наших программах и к определению правил работы этих программ (“клиенты могут быть сгруппированы по регионам и правилам оплаты”). Часто оказывается полезной возможность сделать различные правила элементами самой программы. Языки обычно обеспечивают некоторые встроенные механизмы для этого, например правила управления доступом, которые позволяют помечать классы и методы как открытые или закрытые.

Однако очень часто наши желания в этой области выходят за рамки поддержки языка. Например, мы можем захотеть ограничить диапазон значений, которые может принимать целочисленное поле, отметить методы, которые должны быть выполнены в рамках тестирования, или указать, что класс может быть безопасно сериализован.

Аннотация представляет собой часть информации об элементе программы. Можно получить эту информацию и работать с ней во время выполнения программы или даже во время компиляции, если такая возможность поддерживается средой. Таким образом, аннотации обеспечивают механизм расширения языка программирования.

Я использую здесь термин “аннотация”, так как именно он употребляется в языке программирования Java. Похожий синтаксис есть в .NET, но используемый в нем термин “атрибут” охватывает слишком много других понятий, поэтому я предпочитаю следовать терминологии Java. Однако рассматриваемая здесь концепция имеет более широкий характер, чем синтаксис, и те же преимущества можно получить и без специального синтаксиса такого рода.

42.1. Как это работает

При применении аннотаций имеются два вопроса — их определение и обработка. Хотя оба они зависят от возможностей, изменяющихся в разных языках, определение и обработка аннотаций — это относительно независимые операции, и одна и та же методика обработки может применяться для аннотаций, определенных различными способами. Для соответствия нашей общей модели DSL синтаксис определения аннотаций представляет работу аннотаций как внутренний DSL. Всякий раз они развиваются семантическую модель (*Semantic Model* (171)) путем присоединения данных к модели времени выполнения программы, встроенной в язык. Последующие этапы обработки соответствуют функционированию семантической модели; как и в случае любого DSL, они могут включать выполнение модели и генерацию кода.

42.1.1. Определение аннотации

Наиболее очевидный способ определения аннотации — применение специализированного синтаксиса, имеющегося в ряде языков программирования. Так, в Java можно пометить тестовый метод следующим образом.

```
@test public void softwareAlwaysWorks()
```

В C# то же самое делается так.

```
[Test] public void SoftwareAlwaysWorks()
```

Оба языка программирования допускают наличие параметров в аннотациях, так что вы можете написать код наподобие следующего.

```
class PatientVisit...
    @ValidRange(lower = 1, upper = 1000, units = Units.LB)
    private Quantity weight;
    @ValidRange(lower = 1, upper = 120, units = Units.IN)
    private Quantity height;
```

Использование специализированного синтаксиса — наиболее очевидный, а часто и наиболее простой способ добавления аннотаций. Однако имеются и другие методики, которые можно применять в своей работе.

Одним из самых естественных способов определения аннотации является использование методов класса. Предположим, нужно добавить аннотацию, указывающую диапазон допустимых значений поля. Пусть, например, высота может иметь значения от 1 до 120 (дюймов), а вес — от 1 до 1000 (фунтов) (для простоты ограничимся целочисленными значениями). Мы укажем эти диапазоны в Ruby следующим образом.

```
valid_range :height, 1..120
valid_range :weight, 1..1000
```

Чтобы это сработало, определим метод класса `valid_range`. Этот метод получает два аргумента, имя поля и диапазон, ограничивающий поле. Метод класса может делать с этими данными что угодно. Он может, например, просто добавить их в структуру или создать и сохранить объект валидатора.

Такое использование методов класса может быть почти столь же простым, как и использование специально разработанного синтаксиса. Самая большая проблема здесь в том, что вызову метода класса необходимо имя аннотируемого элемента программы.

Это хотя и приводит к дополнительному многословию, дает программисту свободу в от-делении аннотаций от аннотированных объявлений. Это большой выигрыш для языков, в которых легко поступить таким образом, — при этом оказывается не нужным специ-альный синтаксис аннотаций.

Для подобного применения методов класса нужно помнить о некоторых вопросах. Чтобы аннотации были сохранены, они должны быть выполнены. Приведенный выше пример на языке Ruby выполняется при загрузке кода. Некоторым языкам для этого мо-гут понадобиться дополнительные механизмы. Самый простой способ хранения данных аннотации — в переменных класса, но во многих языках переменные класса совместно используются и самим классом, и его подклассами, что, конечно, никак не испортит си-туацию в конкретном рассматриваемом примере, но может привести к проблемам в дру-гих случаях.

Я описал эту технологию в объектно-ориентированных терминах, но вы можете сде-лать то же самое с любым языком, который позволяет легко представлять свои элементы. Так, можно определить структуру Lisp, отмечающую имена функций некоторыми дан-ными. Такая структура может храниться где угодно, лишь бы в случае необходимости она могла быть найдена при последующей обработке.

Обычная методика, используемая в статически типизированных языках, — примене-ние маркирующего интерфейса. Она предусматривает определение интерфейса без ка-ких-либо методов и реализации. Наличие интерфейса эффективно отмечает класс для последующей обработки. Этот метод работает с классами, но не с методами или полями.

Простую форму аннотации можно обеспечить с помощью соглашения об именова-нии. Это именно то, что было сделано во многих реализациях xUnit, — тестовые методы, которые отмечались как таковые путем начала их названий со слова `test`. Для простых аннотаций этот метод может работать достаточно неплохо, но он ограничен сложностью поддержки нескольких аннотаций и практически не позволяет работать с параметрами.

Во всех этих случаях аннотации обрабатываются встроенными языковыми конструк-циями для создания семантической модели (*Semantic Model* (171)). В дополнение к обычным ограничениям внутренних DSL — синтаксис DSL ограничен синтаксисом базового языка — имеются дополнительные ограничения для аннотаций. При наличии аннотаций семантическая модель должна быть основана на фундаментальном представ-лении самой программы. В объектно-ориентированной программе основополагающим является представление классов, полей и методов. Семантическая модель аннотаций яв-ляется декорацией для этой структуры — вы практически не можете построить полно-стью независимую семантическую модель.

42.1.2. Обработка аннотаций

Аннотации определяются в исходном тексте, но доступны для обработки на более поздних стадиях — в основном во время компиляции, загрузки программы или при обычных операциях времени выполнения.

Обработка во время обычной операции, пожалуй, является самым распространенным случаем. Она предполагает использование аннотаций для управления некоторыми аспек-тами поведения объекта. Простой пример этого — запуск тестовых методов в тестирую-щих средах в стиле xUnit. Эти инструменты позволяют определять тесты как методы в тестовом классе. Не все методы являются тестовыми, поэтому некоторые схемы анно-таций используются для идентификации тестов. Программа запуска тестов находит эти тестовые методы и запускает их.

Отображение базы данных также может работать подобным образом. Программа отображения базы данных опрашивает атрибуты, чтобы выяснить, как поля в программе отображаются на структуры постоянного хранения. Затем эта информация используется для отображения данных.

Такой вид обработки может выполняться как при загрузке программы, так и при обработке. Проверка таких, как показано выше, аннотаций может быть частично обработана во время запуска программы для создания объектов валидаторов, присоединяемых к классам. Эти валидаторы могут затем использоваться для проверки объектов во время выполнения программы.

Такое использование аннотаций времени выполнения соответствует общему подходу выполнения модели DSL. Как и в случае любого DSL, имеется альтернатива в виде генерации кода. Если у нас имеется динамический язык, то такая генерация кода может быть выполнена во время работы — как правило, в процессе загрузки программы. Она может принимать форму генерации новых классов или добавления методов к существующим классам.

Для компилируемых языков программирования генерация кода во время выполнения, как правило, происходит сложнее. Можно прямо во время работы запустить компилятор и связать полученный модуль динамически, но это весьма неудобный способ. Еще один вариант связан с предоставляемыми компилятором возможностями перехвата для обработки аннотаций (как в настоящее время в Java).

Конечно, код может быть сгенерирован и перед компиляцией. Так, для примера с проверкой диапазона мы могли бы генерировать метод проверки либо в основном классе, либо как отдельный объект. После этого при компиляции данный код становится частью программы. Однако такое перемешивание написанного и сгенерированного кода может сбивать с толку и приводить к ошибкам.

Еще один способ работы в случае компилируемых программ — создание байт-кода. При таком подходе компилятор компилирует программу, и уже после мы работаем с байт-кодом для добавления сгенерированных шагов.

Обработка может происходить в нескольких местах с несколькими определениями обработки. Если мы строим веб-приложение и необходимо определить проверки для полей, их неплохо запускать в нескольких местах. Для лучшего времени отклика имеет смысл запускать их в браузере с помощью JavaScript. Но полагаться на эту проверку нельзя, так как пользователь всегда может ее обойти. Поэтому нужно выполнять проверку еще и на сервере. Используя аннотации, можно создать проверку времени выполнения для сервера и сгенерировать код Javascript для проверки в браузере без дублирования кода. Обе проверки могут быть полностью сгенерированы из одной аннотации.

42.2. Когда это использовать

Широкомасштабное использование аннотаций в основных языках программирования — дело все еще относительно новое. Мы все еще выясняем, когда лучше всего их использовать.

Ключевой особенностью аннотаций является то, что они позволяют отделить определение от обработки. Пример с проверкой хорошо это иллюстрирует. Если мы хотим обеспечить допустимый диапазон значений поля, то очевидный способ сделать это — реализовать как часть метода установки значения поля. Проблема в том, что такой способ сочетает в себе определение ограничения с его обеспечением, в данном случае — выполняя проверку при изменении значения.

Во многих ситуациях полезно выполнять проверку ограничений в другое время, например, позволив пользователю заполнить форму, и выполняя проверку только перед ее

отправкой. Чтобы получить такое поведение проверки при отправке, можно использовать, например, общий метод проверки объекта, но и при этом ограничения определяются в тот же момент, когда и проверяются.

Отделив определение и обработку, можно проверить ограничения в разное время, возможно, даже применяя в разные моменты различные подмножества ограничений. Это может также сделать код более понятным, допуская наличие автономных ограничений, так что программист может видеть определение ограничения не загроможденным механизмом выполнения проверок.

Итак, сила аннотаций проявляется там, где имеет смысл отделение определения и обработки данных. Они могут понадобиться, чтобы иметь возможность изменять обработку независимо от определения или делать определение более понятным, обеспечивая его автономность.

Недостатком использования аннотаций является то, что неудобно отслеживать одновременно и определение, и обработку. Если нужно разобраться в них, то аннотации заставят вас выполнять поиск в двух несвязанных местах. Код обработки также носит обобщенный характер, что затрудняет работу с ним.

Следствием сказанного является то, что определение аннотации должно быть декларативным и не должно включать в себя никакого потока логики. Кроме того, оно не должно влечь за собой ни какой-либо связи со временем выполнения логики обработки, ни любого упорядочения обработки аннотаций, связанных с одними и теми же или различными элементами программы.

42.3. Пользовательский синтаксис с обработкой времени выполнения (Java)

Для первого примера применения аннотаций я решил использовать более очевидный случай языка, который имеет пользовательский синтаксис для аннотаций. В данном случае это Java.

Вот как указать допустимый диапазон для целого значения.

```
class PatientVisit...
    @ValidRange(lower = 1, upper = 1000, units = Units.LB)
    private Quantity weight;
    @ValidRange(lower = 1, upper = 120, units = Units.IN)
    private Quantity height;
```

Чтобы этот код работал, необходимо определить тип аннотации наподобие следующего.

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidRange {
    int lower() default Integer.MIN_VALUE;
    int upper() default Integer.MAX_VALUE;
    Units units() default Units.MISSING;
}
```

В системе аннотаций Java тип аннотации сам по себе представляет объект, который содержит только поля, которые должны быть литералами или другими аннотациями.

В результате вся обработка аннотаций выполняется в других местах. Я буду запускать обработку проверки с помощью объектов, проверяющих самих себя.

(Это примечание не совсем относится к данной теме, но я полагаю важным отметить, что наличие объекта, проверяющего самого себя — не всегда правильная стратегия. Проверяя что-либо, вы всегда делаете это в определенном контексте, и этот контекст обычно представляет собой некоторое действие с участием объекта. Этот подход подраз-

зумевает корректность проверки во всех контекстах использования данного кода. Это справедливо далеко не всегда.)

```
class DomainObject...
    boolean isValid() {
        return new ValidationProcessor().isValid(this);
    }
public class PatientVisit extends DomainObject
```

Все, что делает метод объекта предметной области, — это делегирует работу процесору проверки.

```
class ValidationProcessor...
    public boolean isValid(Object arg) {
        for (Field f : arg.getClass().getDeclaredFields())
            for (Annotation a : f.getAnnotations())
                if (doesAnnotationValidationFail(arg, f, a))
                    return false;
        return true;
    }
    public boolean
        doesAnnotationValidationFail(Object obj, Field f,
                                      Annotation a) {
        FieldValidator validator =
            validatorMap().get(a.annotationType());
        if (null == validator) return false;
        return !validator.isValid(obj, f);
    }
    private Map<Class, FieldValidator> validatorMap() {
        Map<Class, FieldValidator> result =
            new HashMap<Class, FieldValidator>();
        result.put(ValidRange.class,
                  new ValidRangeFieldValidator());
        return result;
    }
```

Процессор проверки сканирует класс целевого объекта в поисках аннотаций, выясняя, какие из них являются проверками, сохраняет объект проверки для каждой аннотации и запускает его для выполнения проверки целевого объекта.

Большая часть этого кода нуждается в однократном запуске, так как аннотации не изменяются во время выполнения. Поиск более эффективного способа выполнения этого установочного кода я оставляю вам в качестве самостоятельного задания, но только если вы обещаете выполнить его, лишь окончательно убедившись, что это действительно узкое место с точки зрения производительности программы в целом.

Связь между аннотацией и классом обработки создается с помощью словаря, встроенного в `validatorMap()`. Если у вас есть схема, в которой аннотация может содержать код, то она могла бы реализовать метод `IsValid` самостоятельно. Я мог бы также включить в аннотацию имя класса валидатора в качестве одного из ее полей. Я не сделал этого потому, что обычно предпочитаю (по крайней мере в Java) делать аннотации не зависящими от механизма обработки.

Теперь у меня есть объект проверки диапазона.

```
class ValidRangeFieldValidator...
    public boolean isValid(Object obj, Field field) {
        ValidRange r = field.getAnnotation(ValidRange.class);
        field.setAccessible(true);
        Quantity value;
        try {
```

```

        value = (Quantity)field.get(obj);
    } catch (IllegalAccessException e) {
        throw new RuntimeException(e);
    }
    return (r.units() == value.getUnits())
        && (r.lower() <= value.getAmount())
        && (value.getAmount() <= r.upper());
}

```

42.4. Использование метода класса (Ruby)

Ruby является языком, в котором нет пользовательского синтаксиса аннотаций, но сами аннотации широко применяются. В Ruby мы определяем аннотации с методом класса, вызываемым непосредственно в теле класса.

```

class PatientVisit < Domain Object...
    valid_range :height, 1..120
    valid_range :weight, 1..1000

```

(В примерах Ruby для простоты использованы целые числа.)

Подобный этому код непосредственно в теле класса будет выполнен при загрузке последнего, поэтому он хорошо подходит для инициализации такого рода.

```

class DomainObject...
    @@validations = {}

    def self.valid_range name, range
        @@validations[self] ||= []
        v = lambda do |obj|
            range.include?(obj.instance_variable_get("@#{name.to_s}"))
        end
        @@validations[self] << v
    end

```

Реализация здесь достаточно проста. Я сохраняю валидатор с помощью переменной класса. Эту переменную класса необходимо сделать хешем, индексируемым фактическим классом, поскольку значение переменной класса разделяется всеми подклассами.

Метод `valid_range` при вызове начинает с инициализации при необходимости значения хеша пустым массивом. Затем создается и добавляется в массив получающее единственный аргумент замыкание, которое осуществляет проверку.

Я также добавляю к каждому объекту метод для проверки этого объекта.

```

class DomainObject...
    def valid?
        return @@validations[self.class].all? { |v| v.call(self) }
    end

```

Использование переменной класса с хешем для хранения различных значений для каждого класса на самом деле является способом реализации переменной экземпляра класса. В Ruby я могу сделать это непосредственно следующим образом.

```

class DomainObject...
    class << self; attr_accessor :validations; end

    def self.valid_range name, range
        validations ||= []
        v = lambda do |obj|
            range.include?(obj.instance_variable_get(name))
        end
        validations << v
    end

```

```

    end
  @validations << v
end

class DomainObject...
  def valid?
    return self.class.validations.all? { |v| v.call(self) }
  end

```

42.5. Динамическая генерация кода (Ruby)

Одной из приятных особенностей работы с динамическим языком программирования является возможность добавлять код во время выполнения программы. Я могу воспользоваться ею, чтобы показать дальнейшее усовершенствование обработки аннотаций. На этот раз я хочу обеспечить не просто метод проверки объекта в целом, но и методы проверки отдельных полей. Таким образом, в уже рассмотренном примере я хочу иметь не только метод `valid?`, но и методы `valid_height?` и `valid_weight?` для конкретных полей. Я бы хотел, чтобы эти методы генерировались автоматически, так, чтобы любое поле с аннотацией проверки автоматически получало свой метод проверки.

Приятное заключается в том, что мне не надо модифицировать вызовы аннотаций в классе — они остаются теми же, что и в более простом случае.

```

class PatientVisit...
  not_nil :height, :weight
  valid_range :height, 1..120
  valid_range :weight, 1..1000

```

Для хранения валидаторов я использую подход с переменными экземпляра класса. Отличие состоит в том, что вместо хранения валидаторов в виде простых замыканий я создаю классы валидаторов полей, получающие в качестве аргументов имя поля и замыкание.

```

class DomainObject...
  class << self; attr_accessor :validations; end

  def valid?
    return self.class.validations.all? { |v| v.call(self) }
  end

  class FieldValidator
    attr_reader :field_name
    def initialize field_name, &code
      @field_name = field_name
      @code = code
    end
    def call target
      @code.call target
    end
  end

```

Если я использую метод проверки объекта, все валидаторы работают так же, как и ранее. Дополнительным шагом является следующий метод.

```

class DomainObject...
  def self.define_field_validation_method field_name
    method_name = "valid_#{field_name}?"
    return if self.respond_to? method_name
    self.class_eval do
      define_method(method_name) do

```

```
    return self.class.validations.  
        select{|v| v.field_name == field_name}.  
            all?{|v| v.call(self)}  
    end  
end  
end
```

Этот метод проверяет, определен ли он. Если не определен, для добавления нового метода в класс используется метод `define_method`. Он отбирает те проверки, которые применимы к данному полю, и запускает только их. (Я внес вызов `define_method` в вызов `class_eval`, поскольку `define_method` является закрытым методом. Этого можно избежать, используя `class_eval` со строкой.)

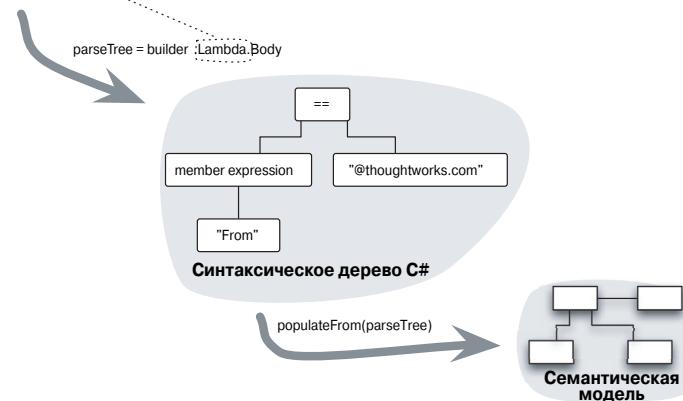
Глава 43

Работа с синтаксическим деревом

Parse Tree Manipulation

Создание синтаксического дерева фрагмента кода для работы с ним с помощью кода обработки DSL

```
var builder = new ImapQueryBuilder(  
    : (q => (q.From == "@thoughtworks.com"));
```



Когда вы пишете код в замыкании, он доступен для некоторого более позднего выполнения. Работая с синтаксическим деревом, можно не только выполнить этот код, но и изучить и модифицировать его синтаксическое дерево.

43.1. Как это работает

Чтобы воспользоваться данным шаблоном, необходима среда программирования, которая поддерживает преобразование фрагмента кода в синтаксическое дерево, с которым можно работать. Это относительно редкая функциональная возможность языка программирования — редкая как в том смысле, что поддерживается только в нескольких

языках программирования, так и в том, что, даже будучи доступной, используется крайне редко. Я не собираюсь проводить подробный обзор инструментов, которые поддерживают эту возможность, но все же упомяну три примера таковых — C# (начиная с версии 3.0), библиотеку Ruby ParseTree и Lisp.

C# и библиотека ParseTree работают во многом схоже. Библиотечный вызов получает фрагмент исходного текста и возвращает структуру данных, представляющую синтаксическое дерево этого фрагмента. В C# вы можете сделать это только с лямбда-выражением. Такое ограничение лямбда-выражениями означает, что вы не можете работать с кодом, состоящим из нескольких инструкций. ParseTree позволяет передать в вызов класс, метод или строку, содержащую код Ruby.

В C# возвращаемая структура данных является иерархией объектов выражений. Это целевые объекты для представления синтаксических деревьев, с иерархией наследования для операторов различного вида. ParseTree возвращает вложенные массивы Ruby с простыми встроенными типами, такими как символы и строки, в качестве листьев.

И в C#, и в Ruby необходимо написать код для обхода дерева и самостоятельной работы с ним. В C# синтаксическое дерево предназначено только для чтения, но вы можете создать его копию и модифицировать ее. Обе библиотеки предоставляют механизм для превращения поддерева в выполняемый код.

Подход Lisp несколько отличается от описанного. Исходный текст Lisp сам по себе по сути представляет сериализованное синтаксическое дерево вложенных списков. Lisp предоставляет синтаксические макросы, которые позволяют изучить любое выражение Lisp и манипулировать им. Стиль программирования на языке программирования Lisp, конечно же, отличается от стиля программирования на C# и Ruby хотя бы тем, что использует макросы, но и здесь можно достичь практически тех же результатов.

Хотя рассматриваемый в данной главе шаблон позволяет вам записать выражение на базовом языке программирования, на это выражение обычно накладываются определенные ограничения, так что записать таким образом все, что вы хотите, невозможно. Как правило на то, с чем вы можете работать в своих выражениях, накладываются определенные ограничения. В таких ситуациях важно быстро выяснить, что вы имеете дело с выражением, с которым не можете справиться. Дело в том, что обычно, обходя синтаксическое дерево, вы знаете, что узлы в дереве соответствуют тому, чего вы от них ожидаете. В случае же применения рассматриваемого шаблона ваше синтаксическое дерево может содержать любую корректную конструкцию базового языка, так что при его обходе придется самостоятельно выполнить некоторые проверки на соответствие ограничениям, накладываемым на синтаксическое дерево.

Зачастую обход всего синтаксического дерева выражения не нужен. В большинстве случаев выполняется обход некоторой части синтаксического дерева. Таким образом, строить полный синтаксический анализатор не нужно. Нужен только анализ фрагментов, необходимых для наполнения семантической модели (*Semantic Model* (171)) и вычисления поддеревьев, как только дальнейшая навигация по дереву станет ненужной.

43.2. Когда это использовать

Шаблон работы с синтаксическим деревом сначала позволяет выразить логику на базовом языке программирования, а затем предоставляет возможность повышенной гибкости при работе с этим выражением. Основной причиной использования указанного шаблона для DSL является желание применять полный спектр возможностей базового языка, а не ограниченного сленга, обычно применяемого для написания конструкций внутреннего DSL.

Однако применение шаблона не ограничивается возможностью выжать из базового языка все до последней капли. В конце концов, одним из преимуществ внутреннего DSL является то, что вы можете смешивать базовый язык с DSL-конструкциями в любых пропорциях. Ключевым же отличием является то, что обычно вы можете управлять только выполнимыми результатами работы языка, но не в состоянии погрузиться в выражения базового языка и манипулировать их структурой.

Но даже при этом имеется не так много примеров необходимости применения рассматриваемого шаблона для DSL. (Как и большинство шаблонов, работа с синтаксическим деревом имеет множество применений вне контекста DSL, которые я не буду здесь рассматривать.) Одним из наилучших примеров поддержки работы с синтаксическим деревом является Linq.

Linq позволяет записывать условия запросов — по сути, логические выражения — с помощью стандартных языков .NET. Эти условия могут затем быть вычислены с помощью структур данных .NET, что является в основном тривиальной задачей. Интересно здесь принятие условия, выраженного на C#, и его превращение в SQL-запрос. Это позволяет писать запросы к базе данных без знания SQL, а также запросы к различным источникам данных. Для этого нужно получить условие C#, превратить его в синтаксическое дерево и при обходе полученного дерева сгенерировать эквивалентный код SQL. По сути, выполняется трансляция исходного текста C# в исходный текст SQL (или исходный текст на некотором другом языке). Рассматриваемый шаблон позволяет использовать знакомый синтаксис для написания условий в ситуациях, когда целевой язык недостаточно хорошо знаком или когда имеется несколько целевых языков.

Другой способ использования шаблона — внесение изменений в синтаксическое дерево для выполнения некоторых полезных манипуляций. Одним из примеров является перенаправление всех вызовов методов определенного объекта на другой объект. Однако полезность такой операции в контексте находящихся в центре внимания данной книги DSL не ясна.

В некоторой степени меня беспокоит то, что работа с синтаксическим деревом является одним из тех методов, сложность которых делает их слишком привлекательными для многих программистов. В результате они могут просто не подумать о других, более простых способах достижения своей цели.

43.3. Генерация запросов IMAP из условий C# (C#)

Некоторые из читателей книги могут быть знакомы с протоколом IMAP, предназначенный для взаимодействия с серверами электронной почты. Если вы используете IMAP, ваша почта остается на сервере и загружается клиентом только для чтения и кэширования. В результате, если вы хотите выполнить поиск в своей электронной почте, он должен выполняться на сервере.

Для поиска с помощью IMAP ваш почтовый клиент отправляет поисковый запрос. Этот запрос, подобно прочим командам IMAP, представляет собой строку. Для записи условий поиска IMAP используется собственный DSL. Я не хочу здесь вникать в его детали (если вас это интересует, обратитесь к [24]) и просто покажу простой пример. Пусть необходимо найти все письма, в которых есть подстрока "entity framework" и которые были отправлены не из домена thoughtworks.com после 23 июня 2008 года. При использовании IMAP этот запрос превращается в команду SEARCH subject "entity framework" sentsince 23-jun-2008 not from "@thoughtworks.com".

DSL для команд поиска IMAP представляет собой хороший предметно-ориентированный язык запросов для электронной почты. Однако мы хотим выражать запросы на C# следующим образом.

```
var threshold = new DateTime(2008, 06, 23);
var builder = new ImapQueryBuilder((q) =>
    (q.Subject == "entity framework") &&
    (q.Date >= threshold) &&
    ("@thoughtworks.com" != q.From));
```

43.3.1. Семантическая модель

Первый шаг состоит в создании семантической модели ([Semantic Model \(171\)](#)) для вывода IMAP. Это простой объект запроса IMAP, который содержит элементы для каждой части запроса. Все эти элементы могут быть логически объединены с помощью логической операции И в единый запрос.

```
class ImapQuery...
internal List<ImapQueryElement> elements =
    new List<ImapQueryElement>();
public void AddElement(ImapQueryElement element) {
    elements.Add(element);
}

interface ImapQueryElement {
    string ToImap();
}
```

Здесь объявлен интерфейс для элементов запроса. Он имеет две реализации: для обработки базовых операторов запроса (`from "@thoughtworks.com"`) и для обработки отрицания (`not`).

```
class BasicElement : ImapQueryElement {
    private readonly string name;
    private readonly object value;
    public BasicElement(string name, object value) {
        this.name = name.ToLower();
        this.value = value;
        validate().AssertOK();
    }

    class NegationElement : ImapQueryElement {
        private readonly BasicElement child;
        public NegationElement(BasicElement child) {
            this.child = child;
        }
    }
}
```

Хотя данный запрос является конъюнкцией, IMAP может использовать обобщенные логические выражения. Это немного сложнее, но большинство запросов к электронной почте могут быть обработаны как конъюнкции. В любом случае для моих иллюстративных целей простой конъюнкции более чем достаточно.

Каждый базовый элемент запроса содержит ключевое слово и значение, отражая способ формирования языка поиска IMAP. Я также добавляю в каждый элемент определенную проверку ошибок, генерируя при их наличии исключения.

```
class BasicElement...
private Notification validate() {
    var result = new Notification();
```

```

if (null == Name)
    result.AddError("Name is null");
if (null == Value)
    result.AddError("Value is null");
if (!stringCriteria.Contains(Name) &&
    !dateCriteria.Contains(Name))
    result.AddError("Unknown criteria: {0}", Name);
if (stringCriteria.Contains(Name) && !(Value is string))
    result.AddError("{0} needs a string argument, got {1}",
                    Name, Value.GetType());
if (dateCriteria.Contains(Name) && !(Value is DateTime))
    result.AddError("{0} needs a DateTime argument, got {1}",
                    Name, Value.GetType());
return result;
}
private readonly static string[] stringCriteria =
    { "subject", "to", "from", "cc" };
private readonly static string[] dateCriteria =
    { "since", "before", "on", "sentbefore",
      "sentsince", "senton" };

class Notification...
public void AssertOK() {
    if (HasErrors) throw new ValidationException(this);
}

```

Применяя этот интерфейс командных запросов, я могу построить модель для своего запроса следующим образом.

```

var expected = new ImapQuery();
expected.AddElement(
    new BasicElement("subject", "entity framework"));
expected.AddElement(
    new BasicElement("since", new DateTime(2008, 6, 23)));
expected.AddElement(new NegationElement(
    new BasicElement("from", "@thoughtworks.com")));

```

Теперь, когда у нас есть семантическая модель, можно генерировать код для команды поиска IMAP. Генерация кода представляет собой просто вывод результата для каждого элемента IMAP.

```

class ImapQuery...
public string ToImap() {
    var result = "";
    foreach (var e in elements) result += e.ToImap();
    return result.Trim();
}

class BasicElement...
public string ToImap() {
    return String.Format("{0} {1} ", name, imapValue);
}
private string imapValue {
    get {
        if (value is string)
            return "\"" + value + "\"";
        if (value is DateTime)
            return imapDate((DateTime)value);
        return "";
    }
}

```

```

private string imapDate(DateTime d) {
    return d.ToString("dd-MMM-yyyy");
}

class NegationElement...
public string ToImap() {
    return String.Format("not {0}", child.ToImap());
}

```

43.3.2. Построение из исходного текста C#

Семантическая модель (*Semantic Model* (171)) позволяет представить и сгенерировать каждую команду поиска для запросов IMAP (или как минимум для использованного здесь подмножества запросов IMAP). Рассмотрим теперь построитель для создания их из исходного текста C#.

Построитель принимает соответствующее лямбда-выражение в конструкторе.

```

class ImapQueryBuilder...
private readonly
    Expression<Func<ImapQueryCriteria, bool>> lambda;
public ImapQueryBuilder(
    Expression<Func<ImapQueryCriteria, bool>> func) {
    lambda = func;
}

```

Чтобы записать выражение в замыкании, нужен некоторый объект, который может действовать, как получатель ключевых слов запроса (*subject*, *sent*, *from*). Этот объект ничего не делает во время выполнения программы; здесь он нужен только для того, чтобы предоставить методы, которые помогут скомпоновать запрос. В результате возвращаемые этими методами значения совершенно не важны, так как в действительности эти методы никогда не будут вызываться.

```

class ImapQueryBuilder...
internal class ImapQueryCriteria {
    public string Subject {get { return ""; } }
    public string To {get { return ""; } }
    public DateTime Sent {get { return DateTime.Now; } }
    public string From {get { return ""; } }
}

```

Для построения запроса воспользуемся отложенным вычислением свойства.

```

class ImapQueryBuilder...
public ImapQuery Content {
    get {
        if (null == content) {
            content = new ImapQuery();
            populateFrom(lambda.Body);
        }
        return content;
    }
    private ImapQuery content;
}

```

Основная работа — рекурсивный обход дерева — выполняется методом *populateFrom*.

```

class ImapQueryBuilder...
private void populateFrom(Expression e) {
    var node = e as BinaryExpression;
    if (null == node)
        throw new BuilderException("Wrong node class", node);
}

```

```

    if (e.NodeType == ExpressionType.AndAlso) {
        populateFrom(node.Left);
        populateFrom(node.Right);
    }
    else
        content.AddElement(new ElementBuilder(node).Content);
}

```

В этот момент я сталкиваюсь с тем, что несмотря на все мое желание позволить клиентам создавать IMAP-запросы на C#, они не могут использовать *любой* код C#. Моя семантическая модель в состоянии обработать только подмножество допустимых выражений C#. Выражение должно состоять из одного или нескольких элементов, которые связаны оператором `&&`. Каждый из узлов элементов должен быть определенным бинарным оператором, у которого один операнд представляет собой ключевое слово — объект условия запроса IMAP. Имеются определенные правила применения операторов к различным ключевым словам. Для строковых ключевых слов (`from`, `subject`, `to`) могут применяться только операторы `==` и `!=`. К ключевым словам, ориентированным на даты (`sent`, `date`), могут применяться любые операторы равенства или сравнения.

Таким образом, я знаю, что единственными элементами, с которыми можно встретиться при обходе синтаксического дерева, являются бинарные выражения, так что если метод `populateFrom` встречает что-либо иное, он генерирует исключение. Если оператор выражения — `&&`, я могу просто рекурсивно обработать дочерние выражения. В случае узла элемента имеется достаточное количество логики для ее вынесения в отдельный класс.

```

class ElementBuilder...
private BinaryExpression node;
public ElementBuilder(BinaryExpression node) {
    this.node = node;
    assertValidNode();
}

```

Такие узлы элементов имеют по два дочерних узла: узел ключевого слова (например, `To`) и узел с некоторым произвольным исходным текстом C#, который возвращает значение, с которым в запросе выполняется сравнение. Я допускаю любой порядок ключевого слова и значения, так как в базовом языке предполагается наличие коммутативности.

Чтобы быть ключевым словом, дочерний узел должен иметь вызов метода экземпляра объекта критерия. Требуется возможность выделить ключевое слово из дочернего узла, так что я создаю метод, получающий дочерний узел и возвращающий ключевое слово, если это узел, или нулевое значение в противном случае.

```

class ElementBuilder...
private string keywordOfChild(Expression node) {
    var call = node as MemberExpression;
    if (null == call) return null;
    if (call.Member.DeclaringType !=
        typeof(ImapQueryBuilder.ImapQueryCriteria))
        return null;
    return call.Member.Name.ToLower();
}

```

Этот метод очень полезен. В первую очередь, он применяется для того, чтобы убедиться в корректности узла элемента, с которым мы имеем дело. Для этого нужно убедиться, что один из его дочерних узлов — узел ключевого слова.

```

class ElementBuilder...
private void assertValidNode() {

```

```

if (null == keywordOfChild(node.Left) &&
    null == keywordOfChild(node.Right))
    throw new BuilderException(
        "Выражение не содержит ключевого слова", node);
if (!isLegalOperator)
    throw new BuilderException(
        "Некорректный оператор", node);
}

```

Проверяется не только наличие узла ключевого слова среди дочерних узлов, но и корректность использованного оператора для найденного ключевого слова.

```

class ElementBuilder...
private bool isLegalOperator {
    get {
        ExpressionType[] dateOperators = {
            ExpressionType.Equal,
            ExpressionType.GreaterThanOrEqual,
            ExpressionType.LessThanOrEqual,
            ExpressionType.NotEqual,
            ExpressionType.GreaterThan,
            ExpressionType.LessThan
        };
        ExpressionType[] stringOperators = {
            ExpressionType.Equal, ExpressionType.NotEqual
        };
        return (isDateKeyword())
            ? dateOperators.Contains(node.NodeType)
            : stringOperators.Contains(node.NodeType);
    }
}
private bool isDateKeyword() {
    return dateKeywords.Contains(keywordMethod());
}
private static readonly string[] dateKeywords =
    { "sent", "date" };
private string keywordMethod() {
    return keywordOfChild(node.Left) ??
        keywordOfChild(node.Right);
}

```

Вы можете заметить, что здесь выполняются дополнительные проверки для ключевых слов, связанных с датами. В случае строковых ключевых слов я полагаюсь на семантическую модель, которая в случае необходимости сообщит, что я пытаюсь создать элемент с неверным ключевым словом. Ключевые слова для дат я должен обрабатывать иначе, поскольку имеется несоответствие между выражением C# и семантической моделью. Если я хочу найти сообщения, отправленные после определенной даты, естественный способ сказать это на C# — что-то наподобие `q.Sent >= aDate`, однако IMAP делает это при помощи выражения `sentsince aDate`. По сути, для определения корректного ключевого слова IMAP необходимо сочетание ключевого слова C# и оператора. В результате понадобится проверка ключевых слов C#, связанных с датой, в построителе, поскольку они являются частью входного DSL, но не семантической модели.

Проверив, что в моем конструкторе имеется корректный узел, я могу упростить последующую логику выделения корректных данных из узла.

Теперь давайте посмотрим на нее. Я начинаю со свойства `Content`, которое отделяет случай простой строки от случая более сложной даты.

```
class ElementBuilder...
    public ImapQueryElement Content {
        get {
            return isDateKeyword() ?
                dateKeywordContent() :
                stringKeywordContent();
        }
    }
```

Для случая строк я создаю базовый элемент запроса, используя соответствующее ключевое слово и значение из другой части. Если использован оператор !=, элемент следует “завернуть” в отрицание.

```
class ElementBuilder...
    private ImapQueryElement stringKeywordContent() {
        switch (node.NodeType) {
            case ExpressionType.Equal:
                return new BasicElement(keywordMethod(), Value);
            case ExpressionType.NotEqual:
                return new NegationElement(
                    new BasicElement(keywordMethod(), Value));
            default:
                throw new Exception("unreachable");
        }
    }
```

Чтобы определить значение, не нужно выполнять анализ соответствующего узла. Вместо этого для получения значения можно просто вернуть содержащееся в нем выражение в C#. Такой подход позволяет помещать в значения моих элементов любой корректный код C# и не обрабатывать его в моем коде навигации.

```
class ElementBuilder...
    private object Value {
        get {
            return (null == keywordOfChild(node.Left))
                ? valueOfChild(node.Left)
                : valueOfChild(node.Right);
        }
    }
    private object valueOfChild(Expression node) {
        return Expression.Lambda(node).Compile().DynamicInvoke();
    }
```

Что касается дат, то несмотря на их большую сложность, я применяю тот же базовый подход. Нужное мне ключевое слово IMAP зависит как от метода ключевого слова в узле, так и от значения оператора. Кроме того, там, где это требуется, я должен генерировать исключения. В качестве первого шага я займусь ключевым словом.

```
class ElementBuilder...
    private ImapQueryElement dateKeywordContent() {
        if ("sent" == keywordMethod())
            return formDateElement("sent");
        else if ("date" == keywordMethod())
            return formDateElement("");
        else throw new Exception("unreachable");
    }
```

При наличии корректного ключевого слова, связанного с датой, я продолжаю работу с типом оператора.

```

class ElementBuilder...
private ImapQueryElement formDateElement(string prefix) {
    switch (node.NodeType) {
        case ExpressionType.Equal:
            return new BasicElement(prefix + "on", Value);
        case ExpressionType.NotEqual:
            return new NegationElement(
                new BasicElement(prefix + "on", Value));
        case ExpressionType.GreaterThanOrEqual:
            return new BasicElement(prefix + "since", Value);
        case ExpressionType.GreaterThan:
            return new NegationElement(
                new BasicElement(prefix + "before", Value));
        case ExpressionType.LessThan:
            return new NegationElement(
                new BasicElement(prefix + "since", Value));
        case ExpressionType.LessThanOrEqual:
            return new BasicElement(prefix + "before", Value);
        default:
            throw new Exception("unreachable");
    }
}

```

Обратите внимание на то, что я воспользовался преимуществом тождественности ключевых слов IMAP, с которыми я работаю. Мой первоначальный код содержал конструкции `switch` для каждого ключевого слова, но я понял, что, если воспользоваться трюком с префиксом, можно избежать дублирования. Код при этом стал немного сложнее, чем я предпочитаю, но я думаю, это нормальная цена за устранение дублирования.

43.3.3. Отступление

На этом можно подводить итоги реализации поиска IMAP, но, прежде чем закончить этот пример, следует кое-что отметить.

Первое, о чем нужно сказать, — это разница между описанием примера и его построением. При описании мне было легче рассмотреть каждую часть реализации отдельно: наполнение семантической модели ([Semantic Model \(171\)](#)) с помощью интерфейса командных запросов, генерация кода IMAP, обход синтаксического дерева. Я думаю, что рассмотрение каждого аспекта по отдельности облегчает понимание.

Однако пример создавался мною не таким же способом. Я выполнил пример в два этапа. Сначала я реализовал поддержку простой конъюнкции базовых элементов, а затем добавил возможность обработки отрицаний. Я записал код всех элементов при первом проходе, а затем расширил и рефакторизовал каждый раздел при добавлении отрицания. Я сторонник последовательного создания программного обеспечения, возможность за возможность, но не думаю, что это наилучший способ объяснить полученный конечный результат. Поэтому не считайте себя глупцом, видя несоответствие структуры конечного результата и моих пояснений.

Еще я хочу заметить, что, хотя описанная методика и приводит в восторг, позволяя воспользоваться самыми дальными тайниками языка, на самом деле я бы не прибег к этому способу. Пожалуй, в качестве альтернативы я бы предпочел соединение методов в цепочки ([Method Chaining \(375\)](#)).

```

class Tester...
var builder = new ChainingBuilder()
    .subject("entity framework")

```

```
.not.from("@thoughtworks.com")
.since(threshold);
```

Вот вся необходимая для этого реализация.

```
class ChainingBuilder...
{
    private readonly ImapQuery content = new ImapQuery();
    private bool currentlyNegating = false;
    public ImapQuery Content {
        get { return content; }
    }
    public ChainingBuilder not {
        get {
            currentlyNegating = true;
            return this;
        }
    }
    private void addElement(string keyword, object value) {
        ImapQueryElement element =
            new BasicElement(keyword, value);
        if (currentlyNegating) {
            element = new NegationElement((BasicElement) element);
            currentlyNegating = false;
        }
        content.AddElement(element);
    }
    public ChainingBuilder subject(string s) {
        addElement("subject", s);
        return this;
    }
    public ChainingBuilder since(DateTime t) {
        addElement("since", t);
        return this;
    }
    public ChainingBuilder from(string s) {
        addElement("from", s);
        return this;
    }
}
```

Эта реализация не так уж и тривиальна, особенно с учетом того, что отрицание заставило меня прибегнуть к переменной контекста (*Context Variable* (187)), но зато она невелика и достаточно проста. В нее требуется добавить методы для поддержки большего количества ключевых слов, но они практически не усложнят реализацию.

Конечно, одной из основных причин простоты является то, что эта структура внутреннего DSL больше похожа на сам запрос IMAP. В самом деле, это действительно просто запрос IMAP, выраженный в виде цепочки методов. Его преимущество по сравнению с использованием самого IMAP сводится к поддержке интегрированной среды разработки. Конечно, некоторые программисты могут предпочесть более C#-образный синтаксис, который обеспечивает работа с синтаксическим деревом, но я должен признать, что я вполне счастлив с IMAP-версией.

Глава 44

Класс таблицы символов

Class Symbol Table

Использование класса и его полей для реализации таблицы символов, обеспечивающее поддержку автозаполнения в интегрированных средах разработки статически типизированных языков программирования

```
public class SimpleSwitchStateMachine  
    extends StateMachineBuilder {  
    Events switchUp, switchDown;  
    States on, off;  
    protected void defineStateMachine() {  
        on.transition(switchDown).to(off);  
        off.transition(switchUp).to(on);  
    }  
}
```

Современные интегрированные среды разработки оснащены множеством мощных полезных возможностей, которые упрощают процесс программирования. Особенно полезным является возможность автозаполнения, основанная на типах. В моих интегрированных средах C# и Java я могу ввести имя переменной и точку и получить список всех методов, которые определены для данного объекта. Даже те программисты, которые, как и я, предпочитают динамически типизированные языки программирования, должны признать важность этого преимущества статически типизированных языков. Работая с внутренним DSL, также не хотелось бы отказываться от этой возможности при вводе имен символов в DSL. Однако наиболее распространенные способы записи символов DSL — с помощью строк или встроенного символьного типа, так что для работы функции автозаполнения просто недостаточно информации.

Класс таблицы символов⁷ позволяет сделать символы в базовом языке статически типизированными, определяя каждый символ в качестве поля построителя выражений (Expression Builder (349)).

⁷ Формально class symbol table следует переводить как “таблица символов класса”, но при этом теряется смысл происходящего — специализированная реализация таблицы символов классом. Поэтому в книге используется название “класс таблицы символов”. — Примеч. пер.

44.1. Как это работает

Данный шаблон основан на написании сценария DSL внутри одного класса построителя выражений (Expression Builder (349)). Этот построитель, как правило, представляет собой подкласс более общего построителя выражений, в котором можно разместить поведение, необходимое для всех сценариев. Постройтель выражения сценария содержит метод для самого сценария и поля для символов. Так, если в вашем DSL есть задачи, и вам надо определить три из них в сценарии, у вас будет объявление поля следующего вида.

```
Tasks drinkCoffee, makeCoffee, wash;
```

Класс Tasks, как и многие другие сущности при работе с DSL, носит нестандартное имя. Здесь вновь удобочитаемость DSL превалирует над обычными правилами стиля кодирования. Определив показанные символы, к ним можно обращаться в сценарии DSL, как к обычным полям; кроме того, интегрированная среда разработки будет предлагать для них автозаполнение, а компилятор — выполнять соответствующие проверки.

Однако просто определить поля недостаточно. Обращаясь к полю в сценарии DSL, я обращаюсь к его содержимому, а не к его определению. Пока я пишу код, интегрированная среда разработки осведомлена и о том, и о другом, но когда я запускаю программу, ссылка на определение поля исчезает, оставляя меня с одним лишь содержимым поля. В обычной жизни это не представляет собой проблемы, но для создания класса таблицы символов необходима ссылка на определение поля во время выполнения программы.

Этого можно достичь путем заполнения каждого поля подходящим объектом до выполнения сценария. Хороший способ сделать это состоит в использовании экземпляра класса в качестве активного сценария — передать код конструктору для заполнения полей и сценария в методе экземпляра. Содержимое полей обычно представляет собой небольшие построители выражений, ссылающиеся на объект лежащей в основе семантической модели (Semantic Model (171)) и содержащие имя поля, чтобы упростить работу с перекрестными ссылками. С точки зрения таблицы символов (Symbol Table (177)) имя поля действует как ключ, а построитель выступает в качестве значения; но иногда требуется другой тип доступа, так что оказывается очень удобно хранить имя поля и в построителе.

Сценарий DSL, как правило, ссылается на поле с применением литерала поля. Для ссылки на задачу wash я могу использовать в сценарии DSL имя поля wash. Однако при обработке сценария DSL нужно, чтобы построители в полях могли обращаться друг к другу. Иногда это будет приводить к поиску полей по имени либо к итеративному обходу всех полей определенного типа. Для этого требуется более сложный код, как правило, с применением механизма отражения. Обычно необходимо не очень большое количество такого кода, так что при достаточной инкапсуляции он не должен доставлять слишком много хлопот.

44.2. Когда это использовать

Главным следствием использования класса таблицы символов является то, что она обеспечивает полную статическую типизацию всех элементов DSL. Основное преимущество этого в том, что такая типизация позволяет интегрированным средам разработки использовать все сложные инструменты, основанные на статической типизации, в частности — автозаполнение. Кроме того, обеспечивается проверка типов в сценарии време-

ни компиляции, которая много значит для большинства программистов (хотя и мало — для меня лично).

Акцентируя внимание на возможностях интегрированных сред разработки, можно прийти к заключению, что этот метод гораздо менее полезен, если нет интегрированной среды разработки, способной использовать преимущества статических типов. Этот метод также мало полезен в динамически типизированных языках.

Недостатком метода является то, что приходится “подгонять” свой DSL, чтобы вписать его в систему типов. В результате классы построителей имеют достаточно странный вид. Кроме того, необходимо размещать сценарии DSL в месте, где они смогут воспользоваться преимуществами описанных возможностей, например в одном и том же классе. Эти ограничения могут сделать DSL более трудными для чтения и использования.

Так что, с моей точки зрения, мы имеем поиск компромисса между ограничениями, накладываемыми на сценарий DSL, и преимуществами поддержки интегрированной средой разработки. Этот компромисс существенно зависит от наличия такой поддержки вющей интегрированной среды разработки.

Если вам требуется поддержка статических типов, можете получить то, что вам нужно, в качестве символов используя перечисления (пример такого подхода можно найти в описании таблиц символов (Symbol Table (177))).

44.3. Статически типизированный класс таблицы символов (Java)

Я использую класс таблицы символов для примера на языке программирования Java из введения. Сценарий DSL располагается в определенном классе.

```
public class BasicStateMachine extends StateMachineBuilder {  
  
    Events      doorClosed, drawerOpened, lightOn, panelClosed;  
    Commands    unlockPanel, lockPanel, unlockDoor;  
    States      idle, active, waitingForLight,  
                waitingForDrawer, unlockedPanel;  
    ResetEvents doorOpened;  
  
    protected void defineStateMachine() {  
        doorClosed. code("D1CL");  
        drawerOpened.code("D2OP");  
        lightOn.     code("L1ON");  
        panelClosed. code("PNCL");  
  
        doorOpened. code("D1OP");  
  
        unlockPanel. code("PNUL");  
        lockPanel.   code("PNLK");  
        lockDoor.    code("D1LK");  
        unlockDoor.  code("D1UL");  
  
        idle  
            .actions(unlockDoor, lockPanel)  
            .transition(doorClosed).to(active)  
            ;  
  
        active  
            .transition(drawerOpened).to(waitingForLight)  
            .transition(lightOn).      to(waitingForDrawer)  
            ;  
    }  
}
```

```

waitingForLight
    .transition(lightOn).to(unlockedPanel)
;

waitingForDrawer
    .transition(drawerOpened).to(unlockedPanel)
;

unlockedPanel
    .actions(unlockPanel, lockDoor)
    .transition(panelClosed).to(idle)
;
}
}

```

Сценарий DSL располагается в собственном классе. Сам по себе он представляет собой один метод, а поля класса — таблицу символов. Я организую все таким образом, что класс сценария DSL представляет собой подкласс построителя — при этом у меня имеется суперкласс построителя, управляющий выполнением сценария. (Подкласс, кроме того, позволяет воспользоваться переносом области видимости в объект (Object Scoping (387)), хотя в данном случае это мне и не нужно.)

```

class StateMachineBuilder...
public StateMachine build() {
    initializeIdentifiers(Events.class, Commands.class,
        States.class, ResetEvents.class);
    defineStateMachine();
    return produceStateMachine();
}
abstract protected void defineStateMachine();

```

Я определяю открытый метод для запуска сценария в суперклассе; он выполняет код настройки полей класса таблицы символов перед запуском сценария. В данном случае запуск DSL-сценария выполняет базовую подготовку информации для конечного автомата, а второй проход фактически создает объекты семантической модели (*Semantic Model (171)*). Таким образом, сценарий запускается в три этапа: инициализация идентификаторов (обобщенный), запуск сценария DSL (конкретный) и генерация модельного конечного автомата (обобщенный).

Первый шаг инициализации идентификаторов необходим, так как любая ссылка на поле в DSL-сценарии обращается к содержимому поля, а не к самому полю. В этом случае подходящими объектами являются конкретные объекты идентификаторов, которые содержат имя идентификатора и ссылаются на объект базовой модели. Реализация этого шага приводит к несколько более “грязному” коду, чем я предпочитаю, поскольку я хочу избежать дублирования кода, написав обобщенный код настройки идентификаторов. Однако никакой обобщенный код не знает о конкретном типе настраиваемого идентификатора, а потому его следует определять динамически.

Надеюсь, все станет немного яснее при рассмотрении примера, в данном случае — класса построителя события (*Events*). Первое, что нужно обсудить, — это имя класса. В любой книге по стилю объектно-ориентированного программирования говорится, что необходимо избегать множественного числа в именах классов, и я согласен с этим советом. Однако в данной ситуации имя во множественном числе в контексте DSL читается лучше, так что это еще один случай нарушения общего правила кодирования для создания хорошего сценария DSL. Именование DSL не отменяет того факта, что это настоящий построитель событий, поэтому в тексте я буду называть его классом построителя событий (аналогично я буду поступать и с родственными ему классами).

Построитель события расширяет общий класс идентификатора.

```
class Identifier...
private String name;
protected StateMachineBuilder builder;
public Identifier(String name,
                  StateMachineBuilder builder) {
    this.name = name;
    this.builder = builder;
}
public String getName() {
    return name;
}

public class Events extends Identifier {
    private Event event;
    public Events(String name, StateMachineBuilder builder) {
        super(name, builder);
    }
    Event getEvent() {
        return event;
    }
}
```

Здесь имеется простое разделение ответственности; класс идентификатора отвечает за общее для всех идентификаторов, а подклассы — за конкретные типы.

Давайте рассмотрим первый этап запуска сценариев — инициализацию идентификаторов. Поскольку должны быть инициализированы многие классы идентификаторов, у меня имеется обобщенный код для решения этой задачи. Таким образом, я могу предоставить список классов, которые являются идентификаторами, и мой код инициализирует все поля этих классов.

```
class StateMachineBuilder...
private void
    initializeIdentifiers(Class... identifierClasses) {
    List<Class> identifierList =
        Arrays.asList(identifierClasses);
    for (Field f : this.getClass().getDeclaredFields()) {
        try {
            if (identifierList.contains(f.getType())) {
                f.setAccessible(true);
                f.set(this, Identifier.create(f.getType(),
                                              f.getName(), this));
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Identifier...
static Identifier create(Class type, String name,
                        StateMachineBuilder builder)
throws NoSuchMethodException, InvocationTargetException,
IllegalAccessException, InstantiationException
{
    Constructor ctor =
        type.getConstructor(String.class,
                           StateMachineBuilder.class);
    return (Identifier) ctor.newInstance(name, builder);
}
```

Такая реализация сложнее, чем я предпочитаю, но зато не нужно писать дублирующие методы инициализации. По сути, я просматриваю каждое поле объекта сценария DSL и, если тип поля является одним из известных типов, инициализирую его с помощью специального статического метода, который находит и вызывает правильный конструктор. В результате после вызова `initializeIdentifiers` все эти поля оказываются заполненными объектами, которые помогут мне построить конечный автомат.

Следующий шаг состоит в выполнении самого сценария DSL. Сценарий выполняется путем создания подходящих промежуточных объектов для хранения всей информации о конечном автомате.

Первым шагом является определение кодов событий и команд.

```
class Events...
public void code(String code) {
    event = new Event(getName(), code);
}
```

Поскольку код имеет всю необходимую для создания объекта события информацию, его можно создать, вызвав метод `code` и поместив его внутрь идентификатора (построитель команды выглядит аналогично).

Построители событий и команд представляют собой вырожденные построители выражений (*Expression Builder* (349)). Построитель состояния немного ближе к обычному построителю, так как он нуждается в нескольких шагах.

Поскольку объект модели состояний не является неизменяемым, его можно создать в конструкторе.

```
class States...
private State content;
private List<TransitionBuilder> transitions =
    new ArrayList<TransitionBuilder>();
private List<Commands> commands =
    new ArrayList<Commands>();

public States(String name, StateMachineBuilder builder) {
    super(name, builder);
    content = new State(name);
}
```

Первое поведение, которое я покажу, — это создание действий. Базовое поведение простое: я прохожу по переданным идентификаторам команд и сохраняю их в построителе состояния.

```
class States...
public States actions(Commands... identifiers) {
    builder.definingState(this);
    commands.addAll(Arrays.asList(identifiers));
    return this;
}
```

Если DSL-сценарий всегда определяет коды перед определением состояний (как это сделал я), можно было бы сэкономить на хранении построителей команд в построителе состояния и вместо этого размещать объекты команд в объекте состояния. Однако это может привести к ошибкам, если состояние будет определено до его кодов действий. Использование построителя в качестве промежуточного объекта обеспечивает работоспособность в любом случае.

Здесь есть определенная хитрость. DSL предполагает, что первое упомянутое состояние является начальным. В результате, сталкиваясь с определением состояния, следует

проверять, не является ли это состояние первым определяемым, и если является, то его нужно сделать начальным состоянием. Однако выяснить, является ли определение состояния первым, может только общий построитель конечного автомата. Поэтому именно он должен принимать решение о том, какое состояние должно быть отмечено как начальное.

```
class StateMachineBuilder...
protected void definingState(States identifier) {
    if (null == start) start = identifier.getState();
}
```

Построитель состояния должен вызвать построитель конечного автомата, чтобы сообщить ему о своем определении, но он не должен знать, что построитель конечного автомата будет делать с этой информацией. Таким образом, построитель состояния фактически выполняет уведомление о событии (поскольку это все, что ему известно), а построитель конечного автомата сам решает, что ему делать с этим событием.

Кроме того, построитель состояния может определять переход. Эта задача несколько сложнее, так как требует нескольких шагов. Я начинаю с метода `transition`, который создает отдельный объект построителя перехода.

```
class States...
public TransitionBuilder transition(Events identifier) {
    builder.definingState(this);
    return new TransitionBuilder(this, identifier);
}
class TransitionBuilder...
private Events trigger;
private States targetState;
private States source;

TransitionBuilder(States state, Events trigger) {
    this.trigger = trigger;
    this.source = state;
}
```

Поскольку мне не требуется упоминать тип построителя переходов в сценарии DSL, я могу дать ему более значащее имя. Единственный метод построителя — `to`, добавляющий его в список построителей переходов построителя исходного состояния.

```
class TransitionBuilder...
public States to(States targetState) {
    this.targetState = targetState;
    source.addTransition(this);
    return source;
}
```

Это элементы, в которых я нуждаюсь для охвата всей конкретной информации в сценарии DSL. Когда сценарий запускается, у меня появляется структура промежуточных данных: построители находятся в полях объекта сценария DSL. Теперь необходимо пройти по этой структуре для получения полносвязной модели конечного автомата.

```
class StateMachineBuilder...
private StateMachine produceStateMachine() {
    assert null != start;
    StateMachine result = new StateMachine(start);
    for (States s : getStateIdentifiers())
        s.produce();
    produceResetEvents(result);
    return result;
}
```

Основная работа здесь заключается в том, чтобы пройти через все построители состояний, заставляя их производить свои связанные объекты модели. Чтобы найти все эти состояния, нужно получить все объекты из полей класса сценария, так что я опять воспользуюсь механизмом отражения, чтобы найти все поля с типом построителя состояний.

```
class StateMachineBuilder...
private List<States> getStateIdentifiers() {
    return getIdentifiers(States.class);
}
private <T extends Identifier>
List<T> getIdentifiers(Class<T> klass) {
    List<T> result = new ArrayList<T>();
    for (Field f : this.getClass().getDeclaredFields()) {
        if (f.getType().equals(klass))
            try {
                f.setAccessible(true);
                result.add((T) f.get(this));
            } catch (IllegalAccessException e) {
                throw new RuntimeException(e);
            }
    }
    return result;
}
```

Для создания своего объекта модели построитель состояния связывает команды и генерирует переходы.

```
class States...
void produce() {
    for (Commands c : commands)
        content.addAction(c.getCommand());
    for (TransitionBuilder t : transitions)
        t.produce();
}

class TransitionBuilder...
void produce() {
    source.getState()
        .addTransition(trigger.getEvent(),
                      getSourceState().getState());
}
```

Последний шаг состоит в генерации сбрасывающих событий.

```
class StateMachineBuilder...
private void produceResetEvents(StateMachine result) {
    result.addResetEvents(getResetEvents());
}
private Event[] getResetEvents() {
    List<Event> result = new ArrayList<Event>();
    for (Events identifier:getIdentifiers(ResetEvents.class))
        result.add(identifier.getEvent());
    return result.toArray(new Event[result.size()]);
}
```

Применение класса и его полей в качестве таблицы символов приводит к несколько запутанному коду, однако этой ценой достигаются такие преимущества, как статическая типизация и поддержка интегрированных сред разработки. Обычно это вполне разумный и оправданный компромисс.

Глава 45

Шлифовка текста

Textual Polishing

Простая замена текста перед серьезной обработкой

`3 hours ago => 3.hours.ago`

Внутренние DSL зачастую легче в разработке, особенно если разработчик плохо владеет синтаксическим анализом. Однако получающийся в результате DSL содержит артефакты базового языка программирования, которые могут усложнять чтение сценария не программистами.

В ходе шлифовки текста используется ряд простых подстановок регулярных выражений для получения более удобочитаемого исходного текста.

45.1. Как это работает

Шлифовка текста — очень простая технология. Она предусматривает выполнение ряда подстановок в сценарии DSL перед его передачей синтаксическому анализатору. Вот простейший пример: многие читатели находят применение точек в вызовах методов отталкивающим. Простая замена пробелов точками может превратить `3 hours ago` в `3.hours.ago`. Более серьезные замены могут превратить `3%` в `percentage(3)`. Результатом шлифовки текста является выражение внутреннего DSL.

Определение шлифовки заключается в простом написании последовательности замен регулярных выражений, что поддерживается большинством языковых сред. Сложнее корректно написать регулярные выражения, чтобы случайно не получить нежелательную замену. Так, пробелы в кавычках, вероятно, не должны быть превращены в точки, но это требование существенно усложняет написание соответствующего регулярного выражения.

Я чаще всего встречал шлифовку текста в динамических языках, в которых текст может быть вычислен во время выполнения. Здесь язык программирования считывает выражение DSL, шлифует его, а затем вычисляет получившийся внутренний код DSL. Однако то же самое можно сделать и с помощью статического языка. В этом случае шлифовку следует выполнить до компиляции DSL-сценария, что добавляет еще один шаг в процесс сборки.

Хотя шлифовка текста в основном применяется с внутренними DSL, иногда она может быть полезной и для внешних DSL. Если некоторые вещи трудно обнаружить с по-

мощью обычной цепочки из лексического и синтаксического анализаторов, может помочь предварительная обработка текста. В качестве примеров можно привести семантические отступы и, возможно, семантические символы новой строки.

Вы можете рассматривать шлифовку текста как простое применение текстовых макросов (Macros (195)) со всеми вытекающими из этого проблемами.

45.2. Когда это использовать

Признаюсь, я очень настороженно отношусь к этой методике. Мне кажется, что если использовать ее немного, от этого будет мало толку, а если интенсивно, то все станет настолько сложным, что, может быть, лучше использовать внешний DSL. Хотя в основе своей концепция повторных замен проста, сделать ошибки в регулярных выражениях очень легко.

В процессе шлифовки текста не может произойти ничего такого, что могло бы изменить синтаксические структуры, так что вы остаетесь привязанными к базовой синтаксической структуре базового языка. Фактически, я считаю, что следует поддерживать исходный текст DSL до шлифовки и получающийся в результате внутренний DSL уверенно распознаваемыми, как один и тот же текст. Получающийся в результате внутренний DSL должен быть как можно более ясным при чтении программистами; шлифовка требуется только для визуального удобства не программистов.

Если вас раздражает зашумленность внутреннего DSL, воспользуйтесь редактором с подсветкой синтаксиса и настройте его так, чтобы раздражающие “шумовые” символы стали окрашенными в цвет, практически сливающимися с фоном. Таким образом, ваши глаза будут их пропускать, и раздражать вас будут только большие интервалы между словами.

Тем, кто все же решил применять шлифовку, я настоятельно рекомендую использовать вместо нее внешний DSL. Научившись создавать синтаксические анализаторы, вы получите гибкость, о которой даже не мечтали при использовании шлифовки, и поддерживать синтаксический анализатор будет гораздо проще, чем последовательность шагов шлифовки.

45.3. Шлифовка правил дисконта (Ruby)

Рассмотрим приложение, применяющее дисконтные правила к заказам. Простое правило может выглядеть следующим образом: если стоимость заказа превышает 30 тысяч долларов, применяется 3%-ная скидка. На внутреннем DSL в Ruby я мог бы записать это с помощью следующего выражения.

```
rule = DiscountBuilder.percent(3).when.minimum(30000).content
```

Неплохо, но для не программиста все равно слишком запутанно. Определенную ясность можно внести, применив перенос области видимости в объект. Если можно поместить выражения в виде строк в отдельный файл, то можно воспользоваться возможностью `instance_eval` Ruby (форма переноса области видимости в объект (`Object Scoping (387)`)) для вычисления каждой строки.

```
Код обработки...
input = File.readlines("rules.rb")
rules = []
input.each do |line|
  builder = DiscountBuilder.new
  builder.instance_eval(line)
```

```
rules << builder.content if builder.has_rule?
end
```

После этого файл правил может содержать строки наподобие следующей.

```
percent (3).when.minimum(30000)
```

С помощью этого метода я также перемещаю вызов `content` (последний метод в соединении методов в цепочки (Method Chaining (375))) в код обработки из видимой пользователю части DSL. Проверка `builder_has_rule?` необходима, так как вычисляются все строки, и строка, являющаяся комментарием, не может быть определением правила. Аналогично, если правило записано некорректно, это приведет к ошибкам, но в данном примере я пренебрегаю этим фактом.

Этот способ может вполне устраивать программистов, но эксперты в предметной области могут предпочесть другую формулировку — что-то наподобие

```
3% if value at least $30000
```

Этот исходный текст можно превратить в приведенный выше фрагмент DSL с помощью шлифовки текста. Такая шлифовка представляет собой ряд текстовых подстановок.

```
class DiscountRulePolisher...
def polish aString
  @buffer = aString
  process_percent
  process_value_at_least
  process_if
  replace_spaces
  return @buffer
end
```

Первое преобразование превращает `3%` в `percent (3)`.

```
class DiscountRulePolisher...
def process_percent
  @buffer = @buffer.gsub(/\b(\d+)\%\s+/, 'percent(\1) ')
end
```

В этом и заключается базовый подход: создать подходящее регулярное выражение, найти соответствующий текст и заменить его тем, что нужно для настоящего внутреннего DSL.

В этом примере различные элементы разделены пробелами, так же, как обычно во внешнем DSL. В результате с обеих сторон от всех регулярных выражений должны быть граничные выражения. В большинстве случаев это `\b` (граница слова), но иногда мне нужно нечто иное (так, в данном примере — `\s+`, поскольку символ `%` не образует границу слова).

Выражение “`at least`” обрабатывается точно так же, хотя и с применением более сложного регулярного выражения.

```
class DiscountRulePolisher...
def process_value_at_least
  @buffer =
    @buffer.gsub(/\bvalue\s+at\s+least\s+\$?(\d+)\b/, 'minimum(\1)')
end
```

Наш эксперт предметной области предпочитает использовать слово `if`, а не `when`. Это проблема в случае обычного внутреннего DSL, так как `if` представляет собой ключевое слово Ruby, но шлифовка спасает положение.

```
class DiscountRulePolisher...
  def process_if
    @buffer = @buffer.gsub(/\bif\b/, 'when')
  end
```

Альтернативным решением является переименование метода `when` во что-то наподобие `my_if` или `_if`. Такое решение позволяет легче увидеть соответствие между шлифуемым текстом и получающимся в результате исходным текстом DSL.

Мой последний шаг состоит в замене пробелов точками для вызова методов, в результате чего внутренний DSL становится корректным исходным текстом Ruby.

```
class DiscountRulePolisher...
  def replace_spaces
    @buffer = @buffer.strip.gsub(/ +/, ".")
  end
```

Все это выглядит не так уж и плохо, но весь приведенный код решает только одну задачу — обработку данного конкретного примера. Чтобы справиться с большим количеством разных ситуаций, код должен быть гораздо сложнее, уродливее и его должно быть очень много. Поэтому вспомните еще раз мой совет: если в вас вселилось желание заняться шлифовкой текста, задумайтесь, не лучше ли применить внешний DSL.

Глава 46

Расширение литералов

Literal Extension

Добавление методов к литералам

`42.grams.flour`

46.1. Как это работает

Литералы, такие как числа и строки, часто становятся хорошей отправной точкой для выражений DSL. Однако традиционно они являются встроенным типами с фиксированными интерфейсами, так что вы не можете их расширять. Однако все больше языков программирования позволяют теперь добавлять методы к классам сторонних производителей с использованием таких методов, как методы расширения C# и открытые классы Ruby. Эта возможность особенно удобна для DSL, так как позволяет начать цепочку методов с литерала.

Как и при использовании большинства цепочек вызовов методов, следует принять одно важное решение — нужно ли использовать построитель выражений (*Expression Builder* (349)). Если построитель выражений не используется, следует убедиться в том, что все промежуточные типы имеют соответствующие свободные методы. Используя построитель выражений, этого можно избежать, но необходимо убедиться, что из построителя можно легко получить основной объект.

Возьмем выражение `42.grams`. Каким должен быть тип результата? Я вижу три основных варианта: число, количество или построитель выражений. В случае числа обычно выбирают одну единицу в качестве канонической, например в случае веса можно использовать килограмм. Тогда `42.grams` дает `0.042`, а `2.oz` — `0.567`.

Здесь следует упомянуть о том, что мой коллега Нил Форд (Neal Ford) называет **метаморфозами типа** (*type transmogrification*). Выражение `42.grams` начинается с целого числа, но превращается в число с плавающей точкой. Это означает, что все дальнейшие методы в цепочке должны быть определены для различных числовых типов.

В случае количества `42.grams` превращается в объект количества с величиной `42` и граммами в качестве единиц. В общем случае я предпочитаю количества простым числам для представления размерных значений; количества лучше отражают мои намерения, а также позволяют мне определить полезное поведение (например, сообщать о проблемах

для выражений наподобие `42.grams + 35.cm`). К сожалению, почти во всех языках программирования отсутствуют встроенные классы количества, но по крайней мере вы можете легко определить их самостоятельно с любыми требующимися свободными методами. Так как величина количества инкапсулирована, проблема метаморфоза типа существенно снижена, поскольку все последующие методы определяются для количества. Однако класс количества все еще содержит свободные методы, которые могут затруднить его понимание.

Последний вариант заключается в использовании построителя выражений, так что `42.grams` даст экземпляр построителя рецепта. В этот момент вы можете использовать один или более построителей выражений и получить полный контроль над работой остальной части выражения. Проблема в том, что необходимо обеспечить простое получение вызывающим кодом объекта от построителя. Это не проблема для выражений наподобие

```
ingredients {
    42.grams.flour
    2.grams.nutmeg
}
```

Но это остается проблемой для выражений наподобие `42.grams + 3.oz`. В большинстве случаев я предпоютаю применять построитель выражения, но, в первую очередь, все зависит от конкретного контекста применения данного шаблона.

46.2. Когда это использовать

Расширение литералов стало популярной иллюстрацией того, как сделать API более свободным, особенно часто применяемой сторонниками языков, которые к этому приспособлены. Ранее в основных объектно-ориентированных языках программирования возможность добавлять методы в классы сторонних производителей не поддерживалась (хотя в Smalltalk это всегда было возможно). И хотя данная возможность предоставляет большую свободу, есть подозрение, что во многом энтузиазм программистов обусловлен просто получением новой игрушки.

В некоторых средах существует серьезная озабоченность тем, что такое добавление методов к литералам раздувает интерфейс литературальных классов. Расширения литералов нужны только в некоторых контекстах, поскольку во многих контекстах они могут сделать интерфейс класса чрезвычайно запутанным. Если вы столкнулись с этой проблемой, взвесьте полезность расширения литералов и сравните преимущества их применения с проблемами, вызываемыми усложнением интерфейса литературального класса. Некоторые языковые среды позволяют ограничить расширения литералов заданным пространством имен, что позволяет устранить описанную проблему.

46.3. Ингредиенты рецепта (C#)

Из скромности я решил не выпячивать свое умение работать со всеми шаблонами в книге и украл этот пример у моего коллеги Нила Форда (Neal Ford), который использовал его в нескольких статьях. Это просто эскиз на языке C#.

```
var ingredient = 42.Grams().Of("Flour");
```

В этом случае я применяю типы доменов, а не построители выражений. Начнем с добавления метода `Grams` к целому числу.

```
namespace dslOrcas.literalExtension {
    public static class RecipeExtensions {
        public static Quantity Grams(this int arg) {
            return new Quantity(arg, Unit.G);
        }
    }
}
```

Обычно в своих примерах я не показываю пространств имен, но в данном случае они имеют значение — метод `Grams` виден только в соответствующем пространстве имен.

Я возвращаю количество, которое представляет собой простую иллюстрацию применения соответствующего шаблона.

```
public struct Quantity {
    private double amount;
    private Unit units;
    public Quantity(double amount, Unit units) {
        this.amount = amount;
        this.units = units;
    }

    public struct Unit {
        public static readonly Unit G = new Unit("g");
        public String name;
        private Unit(string name) {
            this.name = name;
        }
    }
}
```

Хотя количество в данном случае представляет собой написанный мною класс и я могу делать с ним все что хочу, я не считаю, что метод `Of` должен принадлежать этому классу, поскольку метод `Of` является частью DSL для ограниченного применения, в то время как класс количества может быть частью библиотеки общего назначения. Поэтому я вновь прибегну к методу расширения.

```
public static Ingredient Of(this Quantity arg,
                            string substanceName) {
    return new Ingredient(arg,
                          SubstanceRegistry.Obtain(substanceName));
}
```

Код DSL создает объекты ингредиентов.

```
public struct Ingredient {
    Quantity amount;
    Substance substance;
    public Ingredient(Quantity amount, Substance substance) {
        this.amount = amount;
        this.substance = substance;
    }

    public struct Substance {
        private readonly string name;
        public Substance(string name) {
            this.name = name;
        }
    }
}
```

В DSL для именования ингредиентов я использую строки, разрешая их в объекты с помощью реестра, работающего как таблица символов (Symbol Table (177)).

```
private static SubstanceRegistry instance =
    new SubstanceRegistry();
public static void Initialize() {
    instance = new SubstanceRegistry();
```

```
}

private readonly Dictionary<string, Substance>
    values = new Dictionary<string, Substance>();
public static Substance Obtain(string name) {
    if (!instance.values.ContainsKey(name))
        instance.values[name] = new Substance(name);
    return instance.values[name];
}
```

Часть V

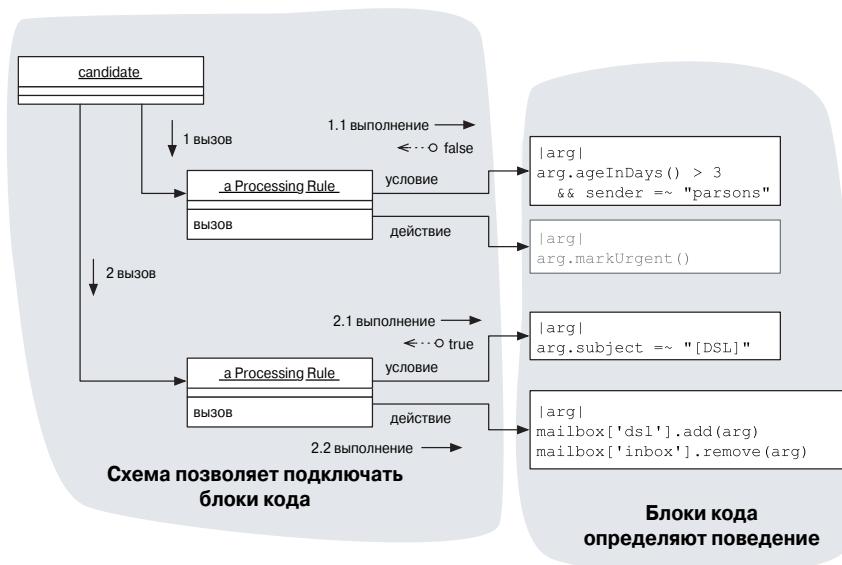
Альтернативные вычислительные модели

Глава 47

Адаптивная модель

Adaptive Model

Размещение блоков кода в структуре данных для реализации альтернативной вычислительной модели



Языки программирования разрабатываются с учетом конкретной вычислительной модели. В случае основных языков программирования это императивная модель с объектно-ориентированной организацией кода. В настоящее время это преобладающий подход, поскольку он представляет собой подходящий компромисс между мощью и понятностью. Однако эта модель не всегда наилучшим образом подходит для той или иной конкретной задачи. И в самом деле, зачастую желание использовать DSL приводит к другим вычислительным моделям.

Адаптивная модель позволяет реализовать альтернативные вычислительные модели в императивном языке. Это можно сделать путем определения модели, в которой связи между элементами представляют поведенческие отношения вычислительной модели. Эта модель обычно требует обращения к разделам императивного кода. Затем вы запускаете модель путем выполнения кода либо над ней (процедурный стиль), либо в рамках самой модели (объектно-ориентированный стиль).

47.1. Как это работает

При написании программного обеспечения мы регулярно строим модели фрагментов реального мира, с которыми работает наша программа. Система каталогизации собирает информацию о продуктах и ценах; веб-сайты средств массовой информации содержат новости, рекламу и дескрипторы, описывающие их совместное размещение. Эти модели могут быть чистыми структурами данных (модели данных) или объединять данные с кодом, который с ними работает (объектные модели). Но даже в объектной модели поток обработки диктуется кодом. Данные, с которыми он работает, могут быть различными, и эти различия могут даже вызывать изменения в деталях обработки, но в целом поток выполнения остается прежним.

Модель состояний тайников, с которой я начал эту книгу, — это зверь совсем иного вида. Общее поведение системы будет существенно изменяться в зависимости от конкретной системы, загруженной в модель состояний. По сути, программой является инстанцирование модели состояний. Конечно, имеется общая семантическая модель (*Semantic Model* (171)) конечного автомата; это постоянный фактор, ограничивающий возможности любого конкретного конечного автомата. Но конфигурация конкретного конечного автомата в самом прямом смысле является выполняемой программой.

Когда модель играет в системе основную поведенческую роль, я называю ее адаптивной. И хотя граница адаптивной модели, как и большинство границ в области программного обеспечения, является размытой, я считаю такую классификацию полезной. На мой взгляд, сутью использования адаптивной модели является чувство, что вы изменяете программу, изменяя экземпляры и их взаимоотношения. Это чувство слаживает границы между кодом и данными, и мы вступаем в мир с новыми возможностями и новыми проблемами. Некоторые сообщества программистов обожают этот мир (в особенности дуальность кода и данных ценится программистами на *Lisp*), но многих разработчиков этот мир одновременно и зачаровывает, и страшит.

Адаптивные модели существуют независимо от DSL, т.е. в вашей системе без DSL может присутствовать адаптивная модель, и вы сможете получить максимальную выгоду от ее использования. Роль DSL здесь заключается в том, чтобы облегчить программирование адаптивной модели путем предоставления языка, на котором вы сможете описать свои намерения более ясно. Это иллюстрируется примерами, которые я использовал для того, чтобы показать разницу между API командных запросов и различными DSL. Одна из наибольших трудностей в применении адаптивной модели — выяснение, что именно она должна делать, и в этом DSL может оказаться неощущимую помощь.

В моих примерах в этой книге в качестве адаптивной модели используется модель объектов в памяти, но адаптивные модели могут принимать различные формы. Адаптивная модель может быть структурой данных, интерпретируемой процедурным кодом. Она часто применяется для хранения модели в базе данных с последующей интерпретацией другими приложениями. Этот стиль часто используется в производственных системах.

Когда мне встречается адаптивная модель, сохраненная в реляционной базе данных, зачастую обнаруживается, что она сопровождается (как правило, сырым) проекционным

редактором (с. 149), обычно использующим для редактирования адаптивной модели формы и поля. При всей работоспособности данного метода есть много преимуществ в использовании вместо него DSL. Предметно-ориентированные языки часто дают более полную картину поведения, хотя это могут сделать и методы визуализации. Возможно, наилучшим аргументом в пользу текстовых DSL является то, что они позволяют легко применять систему управления версиями с адаптивной моделью. Мне всегда не нравится, когда основное поведение системы не находится под управлением надлежащей системы управления версиями кода.

Часто адаптивные модели представлены структурами данных, которые являются хорошо известными структурами графов. В результате при работе с такими моделями оказываются очень полезными учебники по алгоритмам и структурам данных.

47.1.1. Внедрение императивного кода в адаптивную модель

Создавая первый пример конечного автомата, я специально сделал так, чтобы все поведенческие элементы могли быть описаны с помощью простых данных. Действия в конечном автомате представлены путем передачи кода команды. Однако, как правило, адаптивные модели гораздо более тесно взаимодействуют с императивным кодом. В другом конечном автомате я мог бы предпочесть более широкий диапазон выполняемых действиями вещей или наложить ограничивающие условия на переходы. Сделать это в рамках адаптивной модели означает усложнить ее диапазоном императивных выражений, имеющихся в базовом языке программирования. Часто лучшей альтернативой является внедрение кода языка программирования в структуру данных адаптивной модели.

Хорошим примером этого является правило в системе правил вывода (*Production Rule System* (503)). Такое правило состоит из двух частей: логического условия и действия. Зачастую полезно представлять их на базовом языке программирования.

Наиболее естественным способом сделать это является замыкание.

```
rule.Condition = j => j.Start == "BOS";
rule.Action = j => j.Passenger.PostBonusMiles(2000);
```

Замыкания хорошо работают, потому что позволяют легко вставлять произвольные блоки кода в данные структуры. Замыкание является непосредственным объявлением моих намерений. Большой недостаток замыканий в том, что многие языки их не поддерживают. Если ваш язык программирования относится к таким языкам, следует прибегнуть к некоторым обходным путям.

Возможно, самым простым решением является использование *команд* (Command [15]). Для этого я создаю небольшие объекты, “заворачивающие” в себя единственный метод. Мой класс правила использует один из них для состояния, а другой — для действия.

```
class RuleWithCommand {
    public RuleCondition Condition { get; set; }
    public RuleAction Action { get; set; }
    public void Run(Journey j) {
        if (Condition.IsSatisfiedBy(j)) Action.Run(j);
    }
}

interface RuleCondition {
    bool IsSatisfiedBy(Journey j)
}

interface RuleAction {
```

```
    void Run(Journey j);
}
```

Затем я могу получить конкретное правило, создавая подкласс.

```
var rule = new RuleWithCommand();
rule.Condition = new BostonStart();
rule.Action = new PostTwoThousandBonusMiles();

class BostonStart : RuleCondition {
    public bool IsSatisfiedBy(Journey j) {
        return j.Start == "BOS";
    }
}

class PostTwoThousandBonusMiles : RuleAction {
    public void Run(Journey j) {
        j.Passenger.PostBonusMiles(2000);
    }
}
```

Обычно можно уменьшить количество необходимых подклассов, параметризуя команды.

```
var rule = new RuleWithCommand();
rule.Condition = new JourneyStartCondition("BOS");
rule.Action = new PostBonusMiles(2000);

class JourneyStartCondition : RuleCondition {
    readonly string start;
    public JourneyStartCondition(string start) {
        this.start = start;
    }
    public bool IsSatisfiedBy(Journey j) {
        return j.Start == this.start;
    }
}

class PostBonusMiles : RuleAction {
    readonly int amount;
    public PostBonusMiles(int amount) {
        this.amount = amount;
    }
    public void Run(Journey j) {
        j.Passenger.PostBonusMiles(amount);
    }
}
```

Пожалуй, именно этот путь я избрал бы, работая с языком программирования без поддержки замыканий. Другой вариант, который нравится мне гораздо меньше, — использование имени метода и его вызов с помощью механизма отражения.

Я описал применение команд в качестве обходного пути, и если рассматривать их с точки зрения адаптивной модели, то это так и есть. Однако при наполнении адаптивной модели с помощью DSL команды становятся более привлекательным. Во многих ситуациях DSL использует параметры, что естественным образом приводит к параметризованным командам. Использовать всю выразительность замыканий в DSL означает либо применять замыкания во внутреннем DSL, либо внешний код ([Foreign Code \(315\)](#)) — во внешнем DSL. Но последний вариант должен использоваться только в исключительных случаях.

47.1.2. Инструментарий

DSL является ценным инструментом для адаптивной модели, поскольку позволяет программистам настроить экземпляр модели с помощью языка программирования, что делает его поведение более явным. Однако на самом деле DSL не достаточно для работы с адаптивной моделью, когда она становится более сложной. При этом могут пригодиться и другие инструменты.

Зачастую очень сложно отследить, что делает адаптивная модель, так как она использует вычислительную модель, гораздо меньше знакомую программистам. В результате становится очень важной трассировка при выполнении модели. Трассировка должна собрать всю информацию о том, как модель обрабатывает входные данные, и пояснить, почему она сделала то, что сделала. Такая трассировка существенно облегчает поиск ответа на вопрос, почему программа работает именно так, а не иначе.

Когда вы требуете от модели выдать описание ее экземпляра, она может выполнить альтернативную визуализацию самой себя. Графическое описание часто очень полезно. Я видел ряд очень удобных визуализаций с использованием Graphviz — инструмента для автоматического размещения узлов и дуг структуры графа. Хорошим примером может служить диаграмма состояний тайника. Могут также оказаться полезными различные виды отчетов, показывающие, как выглядит модель с разных точек зрения.

Такие визуализации представляют собой простой эквивалент нескольких проекций языковых инструментальных средств. В отличие от упомянутых проекций они не являются редактируемыми (вернее, их можно сделать таковыми слишком высокой ценой). Но даже такие визуализации могут быть чрезвычайно полезными. Вы можете строить их автоматически, как часть процесса сборки, и использовать для проверки своего понимания конфигурации модели.

47.2. Когда это использовать

Адаптивная модель является ключом к использованию альтернативных вычислительных моделей. Она позволяет построить обрабатывающий механизм для альтернативной модели вычислений, который затем можно будет запрограммировать на определенное поведение. Таким образом, как только у вас будет адаптивная модель для системы правил вывода (*Production Rule System (503)*), вы сможете выполнять любой набор правил путем их загрузки в модель. Я обычно советую, чтобы любые альтернативные вычислительные модели, упомянутые в данной книге, реализовывались с помощью адаптивной модели.

Конечно, этот благовидный ответ подразумевает вопрос — когда же следует использовать альтернативные модели вычислений? Это вопрос о том, что же лучше всего подходит для решения нашей задачи. У меня нет однозначного подхода к принятию такого решения. Самое лучшее, что я могу предложить, — это попытаться выразить поведение для различных вычислительных моделей и посмотреть, какая из них упрощает понимание. Такая методика часто означает создание прототипов DSL для управления моделью, так как сама по себе адаптивная модель не может обеспечить достаточной ясности.

В большинстве случаев это предполагает рассмотрение распространенных вычислительных моделей. Шаблоны из этой части книги могут дать вам неплохую отправную точку; если один из них кажется вам соответствующим вашей задаче, испытайтесь его. Существенно реже удается найти совершенно новые, неизвестные ранее вычислительные модели. Зачастую новая реализация может получиться путем постепенного внесения изменений в известную схему. Такая схема может изначально предназначаться для про-

стого хранения данных, но постепенно, с увеличением количества внедренного в нее поведения, вы сможете увидеть, как начинает формироваться адаптивная модель.

У адаптивных моделей есть очень большой недостаток: их может быть очень трудно понять. Я часто сталкивался с ситуациями, когда программисты жалуются на невозможность понять, как работает та или иная адаптивная модель. Создается впечатление встроенной в программу магии, которой пугается множество людей.

Вся жуть проистекает из того факта, что адаптивная модель приводит к неявному поведению. Вы больше не можете рассуждать о том, что делает программа, просто читая ее код. Вместо этого вы должны рассматривать модель конкретной конфигурации, чтобы понять, как ведет себя система. Для многих разработчиков это слишком сложно. Зачастую трудно написать программу, которая бы ясно выражала ваши намерения, но теперь, чтобы сориентироваться, необходимо расшифровать модель данных. Отладка в такой ситуации может быть сплошным кошмаром. Вы можете облегчить себе жизнь, создав подходящий инструментарий, но тогда вы будете тратить время на создание инструментов, а не на работу над поставленной перед вами задачей.

Среди ваших коллег, скорее всего, есть пара программистов, понимающих адаптивные модели. Это фанаты, которые могут невероятно продуктивно их использовать. Все остальные, однако, пребывают в полном замешательстве.

Меня же в замешательство приводит именно этот факт. Я — тот человек, который считает адаптивные модели очень мощным средством. Мне нравится искать их и использовать, и я уверен, что правильно подобранная адаптивная модель может значительно повысить производительность. Но я должен также признать, что они могут быть совершенно чужеродными артефактами для большинства разработчиков. А потому иногда следует отказываться от адаптивной модели — просто потому, что, если в системе будет присутствовать такая магия, ее некому будет сопровождать после увольнения пары понимающих специалистов.

Я надеюсь только на то, что эту проблему удастся решить с помощью DSL. Без предметно-ориентированного языка адаптивную модель трудно программировать, и еще труднее разобраться, как она работает. DSL может во многом превратить неявное поведение в явное, придавая языковую природу конфигурации адаптивной модели. Мне кажется, что с распространением DSL все больше программистов будут комфортно чувствовать себя с адаптивными моделями, и, таким образом, появится возможность массово реализовать преимущества в производительности, которые они обеспечивают.

Глава 48

Таблицы принятия решений

Decision Table

Представление комбинации условных инструкций в табличном виде

Привилегированный клиент	х	х	Да	Да	Нет	Нет
Приоритетный заказ	Да	Нет	Да	Нет	Да	Нет
Международный заказ	Да	Да	Нет	Нет	Нет	Нет
Оплата	150	100	70	50	80	60
Оповещение	Да	Да	Да	Нет	Нет	Нет

The diagram illustrates the relationship between the rows in the decision table and the conditions and outcomes. On the left, there are four arrows pointing from the column headers 'Условия' (Conditions) to the first four rows of the table. On the right, there are three arrows pointing from the last two columns of the table to the column header 'Следствия' (Outcomes).

Тем, у кого есть код, состоящий из нескольких условных инструкций, зачастую весьма трудно разобраться, какие комбинации условий приводят к определенным результатам.

Применяя таблицу принятия решений⁸ и представляя группу условий в виде таблицы, в каждом столбце которой показан результат для конкретного сочетания условий, можно повысить удобочитаемость и облегчить понимание.

48.1. Как это работает

Таблица принятия решений разделена на две части: условия и следствия. В каждой строке условия содержатся состояния данного условия; для простых логических условий каждая ячейка в строке будет содержать значение “истинно” либо “ложно”. Столбцов в таблице столько, сколько необходимо для всех комбинаций условий. Так, для n логических значений в таблице будет 2^n столбцов.

Каждая строка следствия представляет значения одного вывода таблицы. Каждая ячейка представляет собой значение, соответствующее условиям в данном столбце. Так, в случае, представленном на рисунке в начале данной главы, при наличии внутреннего обычного заказа от привилегированного клиента плата составляет 50 долларов, и пред-

⁸ Или *таблицу решений* — в русскоязычной литературе применяются оба термина; см., например, http://ru.wikipedia.org/wiki/Таблица_принятия_решений. — Примеч. пер.

ставитель не оповещается. Таблица принятия решений нуждается только в одном следствии, но их может быть сколько угодно.

Как и в приведенном наброске таблицы, довольно часто используется трехзначная логика, в которой используется третье значение — “все равно”, т.е. этот столбец спрашивлив для любого значения данного состояния. Используя такое значение, можно существенно снизить количество повторений в таблице, что делает ее более компактной.

Ценным свойством таблицы принятия решений является то, что можно определить, все ли перестановки условий в ней присутствуют, и указать пользователю на отсутствующие, если таковые имеются. Может оказаться, что некоторые комбинации условий невозможны, и тогда следствия соответствующих столбцов должны указывать на ошибку. Можно также использовать семантику таблицы, которая рассматривает отсутствующие столбцы как ошибочные.

Таблица может стать гораздо сложнее, если ввести в нее такие вещи, как перечисления, числовые диапазоны или совпадение строк. Каждый такой случай можно рассматривать как логическое значение, но при этом таблица должна знать, что, например, если у нас есть такие условия, как $100 > x > 50$ и $50 \geq x$, то эти условия не могут быть истинны одновременно. В качестве альтернативного решения можно иметь одну строку условия для значения x и позволять пользователю вводить в ячейки диапазоны значений. С таким подходом, как правило, легче работать. Если у нас есть более сложные значения условий, то вычисление всех возможных перестановок может оказаться очень запутанным, и наилучшим решением может быть трактовка всех отсутствующих случаев как ошибочных.

Как обычно, я бы посоветовал построить отдельную семантическую модель (*Semantic Model* (171)) таблицы принятия решений и синтаксический анализатор. В каждом случае нужно решить, насколько обобщенными их следует делать. Вы можете построить модель и анализатор для единственной таблицы принятия решений. У такой таблицы строки условий будут фиксированы в ее коде, как и количество и типы следствий. Обычно значения столбцов остаются настраиваемыми, так что значения следствий для каждого сочетания условий легко изменить.

Более обобщенная таблица принятия решений позволяет настроить типы условий и следствий. Каждому условию необходим некоторый способ указать код вычисления этого условия (имя метода или замыкание). Тип входных данных и каждого следствия в случае строго типизированного языка программирования требуются на этапе компиляции.

Подобные решения необходимы и для синтаксического анализатора. Анализатор может быть создан для фиксированной таблицы принятия решений, даже если он конфигурирует обобщенную семантическую модель. Для большей гибкости для структуры таблицы требуется что-то вроде простой грамматики, чтобы синтаксический анализатор мог правильно интерпретировать входные данные.

Таблицы принятия решений очень просты как для понимания, так и для внесения изменений, и потому особенно хорошо подходят для получения информации от специалистов предметной области. Многие эксперты в предметной области знакомы с электронными таблицами, так что позволить им редактировать данные в электронных таблицах с последующим их импортом в систему — неплохая тактика. Имеется немало способов сделать это; конкретные способы зависят от программ электронных таблиц и вашей платформы. Грубый (но часто эффективный) способ заключается в сохранении таблицы принятия решений в простой текстовой форме наподобие данных, разделенных запятыми (CSV). Как правило, этот метод работает, потому что таблица содержит только значения, без каких-либо формул. Другие подходы предусматривают взаимодействие с электронными таблицами, например запуск и обмен информацией с экземпляром Excel. Электронные таблицы, такие как Excel, которые имеют собственный язык программиро-

вания, могут быть запрограммированы на получение, редактирование и передачу данных таблицы принятия решений в удаленную программу.

48.2. Когда это использовать

Таблицы принятия решений представляют собой очень эффективное средство получения результата множества взаимодействующих условий. Они удобны для программистов и для экспертов в предметной области. Их табличная природа позволяет экспертам работать с ними, используя привычные инструменты электронных таблиц. Самый большой недостаток этого метода заключается в том, что для настройки системы для легкого отображения и редактирования данных требуются немалые усилия. Тем не менее эти усилия, как правило, стоят обеспечиваемых ими коммуникативных преимуществ.

Таблица принятия решений способна справиться только с определенным уровнем сложности — не более того, что можно охватить одним, пусть и весьма сложным, условным выражением. Если вам необходимо объединить несколько видов условных выражений, подумайте о применении системы правил вывода (Production Rule System (503)).

48.3. Вычисление оплаты заказа (C#)

В этом разделе будет рассмотрена таблица принятия решений, работающая с примером, приведенным в начале этой главы.

48.3.1. Модель

Семантическая модель (Semantic Model (171)) в данном случае представляет собой таблицу принятия решений. Я решил создать для этого примера обобщенную таблицу принятия решений, которая может обрабатывать любое количество условий, каждое из которых представляет собой логическое значение трехзначной логики. Для указания входных и выходных типов таблицы принятия решений я использую шаблоны C#. Вот объявление класса и полей.

```
class DecisionTable <Tin, Tout>{
    readonly List<Condition<Tin>> conditions =
        new List<Condition<Tin>>();
    readonly List<Column<Tout>> columns =
        new List<Column<Tout>>();
```

Таблица нуждается в двух видах конфигурации: для условий и столбцов, каждый из которых использует собственный класс. Условия параметризуются входным типом, а столбцы — выходным типом (типом следствий). Начнем с условий.

```
class DecisionTable...
    public void AddCondition(string description,
        Condition<Tin>.TestType test) {
        conditions.Add(new Condition<Tin>(description, test));
    }

    public class Condition<T> {
        public delegate bool TestType(T input);
        public string description { get; private set; }
        public TestType Test { get; private set; }
        public Condition(string description, TestType test) {
            this.description = description;
```

```

        this.Test = test;
    }
}
```

Это позволяет конфигурировать условия для примера, показанного в начале данной главы, с помощью следующего кода.

```

var decisionTable = new DecisionTable<Order, FeeResult>();
decisionTable.AddCondition("Premium Customer",
                           o => o.Customer.IsPremium);
decisionTable.AddCondition("Priority Order",
                           o => o.IsPriority);
decisionTable.AddCondition("International Order",
                           o => o.IsInternational);
```

Входным типом для таблицы принятия решений является заказ. Я не хочу здесь вдаваться в детали, поскольку для данного примера они не имеют ни малейшего значения. Вывод представляет собой специальный класс, который просто “заворачивает” выходные данные.

```

class FeeResult {
    public int Fee { get; private set; }
    public bool ShouldAlertRepresentative { get; private set; }
    public FeeResult(int fee, bool shouldAlertRepresentative) {
        Fee = fee;
        this.ShouldAlertRepresentative =
            shouldAlertRepresentative;
    }
}
```

Следующая часть настройки таблицы состоит в сборе значений столбцов. Здесь я вновь прибегаю к специальному классу для столбцов.

```

class Column <Tresult> {
    public Tresult Result { get; private set; }
    public readonly ConditionBlock Conditions;
    public Column(ConditionBlock conditions, Tresult result) {
        this.Conditions = conditions;
        this.Result = result;
    }
}
```

Столбец состоит из двух частей. Результат представляет собой тип, который обрабатывает следствия. Этот тип — тот же, что и выходной тип самой таблицы принятия решений. Блок условия является специальным классом, который представляет одну комбинацию значений условий.

```

readonly List<Bool3> content = new List<Bool3>();
public ConditionBlock(params Bool3[] args) {
    content = new List<Bool3>(args);
}
```

Я создал трехзначный логический класс для представления значений в условиях. Как он работает, я опишу позже, а пока будем считать, что есть три корректных экземпляра Bool3, соответствующих значениям “истинно”, “ложно” и “все равно”.

Теперь я могу конфигурировать столбцы следующим образом.

```

decisionTable.AddColumn(
    new ConditionBlock(Bool3.X, Bool3.T, Bool3.T),
    new FeeResult(150, true));
decisionTable.AddColumn(
    new ConditionBlock(Bool3.X, Bool3.F, Bool3.T),
    new FeeResult(100, true));
```

```

decisionTable.AddColumn(
    new ConditionBlock(Bool3.T, Bool3.T, Bool3.F),
    new FeeResult(70, true));
decisionTable.AddColumn(
    new ConditionBlock(Bool3.T, Bool3.F, Bool3.F),
    new FeeResult(50, false));
decisionTable.AddColumn(
    new ConditionBlock(Bool3.F, Bool3.T, Bool3.F),
    new FeeResult(80, false));
decisionTable.AddColumn(
    new ConditionBlock(Bool3.F, Bool3.F, Bool3.F),
    new FeeResult(60, false));

class DecisionTable...
    public void AddColumn(ConditionBlock conditionValues,
        Tout consequences) {
        if (hasConditionBlock(conditionValues))
            throw new DuplicateConditionException();
        columns.Add(new Column<Tout>(conditionValues,
            consequences));
    }
    private bool hasConditionBlock(ConditionBlock block) {
        foreach (var c in columns)
            if (c.Conditions.Matches(block)) return true;
        return false;
    }
}

```

Здесь описано, как сконфигурировать таблицу принятия решений, но следующим вопросом является вопрос о том, как она работает. Сердцем таблицы является класс трехзначной логики. Я написал его полиморфно, с использованием различных подклассов для каждого значения.

```

public abstract class Bool3 {
    public static readonly Bool3 T = new Bool3True();
    public static readonly Bool3 F = new Bool3False();
    public static readonly Bool3 X = new Bool3DontCare();
    abstract public bool Matches(Bool3 other);

    class Bool3True : Bool3 {
        public override bool Matches(Bool3 other) {
            return other is Bool3True;
        }
    }

    class Bool3False : Bool3 {
        public override bool Matches(Bool3 other) {
            return other is Bool3False;
        }
    }

    class Bool3DontCare : Bool3 {
        public override bool Matches(Bool3 other) {
            return true;
        }
    }
}

```

Каждый из подклассов `Bool3` имеет свой метод `Matches`, который сравнивает его с другим значением. Аналогично блок условия сравнивает свой список значений `Bool3` с другим блоком условия.

```

public bool Matches(ConditionBlock other) {
    if (content.Count != other.content.Count)
        throw new ArgumentException(
            "Блоки должны быть одного размера");
    for (int i = 0; i < content.Count(); i++)
        if (!content[i].Matches(other.content[i]))
            return false;
    return true;
}

```

Этот метод проверяет соответствие с помощью вызовов `Matches`, но не эквивалентность, так как этот метод не симметричен (т.е. `Bool3.X.Matches(Bool3.T)` истинно, но не наоборот).

Соответствие блоков условий является центральным механизмом. Теперь, когда моя таблица принятия решений настроена, я могу передать ей конкретный заказ для получения величины оплаты.

```

class DecisionTable...
public Tout Run(Tin arg) {
    var conditionValues = calculateConditionValues(arg);
    foreach (var c in columns) {
        if (c.Conditions.Matches(conditionValues))
            return c.Result;
    }
    throw new
        MissingConditionPermutationException(conditionValues);
}
private ConditionBlock calculateConditionValues(Tin arg) {
    var result = new List<bool>();
    foreach (Condition<Tin> c in conditions) {
        result.Add(c.Test(arg));
    }
    return new ConditionBlock(result);
}

```

Теперь мы можем увидеть, как модель таблицы принятия решений настроена и как она работает. Но перед тем как перейти к синтаксическому анализатору, я думаю, имеет смысл показать код, который таблица принятия решений может использовать, чтобы убедиться, что в ней содержатся столбцы для всех перестановок условий.

На верхнем уровне этот код прост. Я пишу функцию для поиска отсутствующих перестановок путем генерации всех возможных перестановок для данного количества условий и проверяю наличие соответствующего столбца.

```

class DecisionTable...
public bool HasCompletePermutations() {
    return missingPermuations().Count == 0;
}
public List<ConditionBlock> missingPermuations() {
    var result = new List<ConditionBlock>();
    foreach(var permutation
        in allPermutations(conditions.Count))
        if (!hasConditionBlock(permutation))
            result.Add(permutation);
    return result;
}

```

Это приводит к вопросу, как можно сгенерировать все возможные перестановки. Я решил, что проще всего сделать это в двумерной матрице, а затем использовать каждый столбец матрицы в качестве перестановки.

```

class DecisionTable...
private List<ConditionBlock> allPermutations(int size) {
    bool[,] matrix = matrixOfAllPermutations(size);
    var result = new List<ConditionBlock>();
    for (int col = 0; col < matrix.GetLength(1); col++) {
        var row = new List<bool>();
        for (int r = 0; r < size; r++) row.Add(matrix[r, col]);
        result.Add(new ConditionBlock(row));
    }
    return result;
}
private bool[,] matrixOfAllPermutations(int size) {
    var result = new bool[size, (int)Math.Pow(2, size)];
    for (int row = 0; row < size; row++)
        fillRow(result, row);
    return result;
}
private void fillRow(bool[,] result, int row) {
    var size = result.GetLongLength(1);
    var runSize = (int)Math.Pow(2, row);
    int column = 0;
    while (column < size) {
        for (int i = 0; i < runSize; i++) {
            result[row, column++] = true;
        }
        for (int i = 0; i < runSize; i++) {
            result[row, column++] = false;
        }
    }
}
}

```

Код для генерации перестановок получился более сложным, чем мне хотелось бы, но, похоже, использование матричной структуры данных облегчает его написание. В подобных ситуациях я предпочитаю применять структуры данных, которые упрощают написание кода, а затем преобразовывать результат в структуру данных, которая нужна мне на самом деле. Это напоминает мне о моей работе инженером, когда задачи, которые трудно решить в обычной системе координат, переводились мною в систему координат, позволяющую решать их проще и быстрее, и решались в ней, после чего я выполнял обратное преобразование.

48.3.2. Синтаксический анализатор

В процессе работы с табличным представлением наподобие рассматриваемого здесь зачастую наилучшим редактором являются электронные таблицы. Имеется масса способов передать данные из электронных таблиц в программу на C#, но я не буду их здесь описывать. Вместо этого я напишу синтаксический анализатор, работающий с простым интерфейсом таблицы.

```

interface ITable {
    string cell(int row, int col);
    int RowCount {get;}
    int ColumnCount {get;}
}

```

Я буду разбирать таблицу в духе трансляции, управляемой разделителями (Delimiter-Directed Translation (213)), но используя строки и столбцы вместо потока токенов с разделителями.

Для этой модели я создал обобщенную таблицу принятия решений, которую я могу использовать с любой таблицей трехзначной логики. Что касается синтаксического анализатора, то я разработаю его специально для данной таблицы. Конечно, можно написать обобщенный синтаксический анализатор таблиц и сконфигурировать его для данного случая, но я оставлю это читателям для развлечения во время бессонницы.

Базовая структура синтаксического анализатора представляет собой командный объект, получающий в качестве входных данных ITable и возвращающий на выходе таблицу принятия решений.

```
class TableParser...
    private readonly DecisionTable<Order, FeeResult>
        result = new DecisionTable<Order, FeeResult>();
    private readonly ITable input;
    public TableParser(ITable input) {
        this.input = input;
    }
    public DecisionTable<Order, FeeResult> Run() {
        loadConditions();
        loadColumns();
        return result;
    }
```

По привычке параметры команде я передаю в конструкторе, а для выполнения работы использую метод Run.

Первый шаг состоит в загрузке условий.

```
class TableParser...
    private void loadConditions() {
        result.AddCondition("Premium Customer",
            (o) => o.Customer.IsPremium);
        result.AddCondition("Priority Order",
            (o) => o.IsPriority);
        result.AddCondition("International Order",
            (o) => o.IsInternational);
        checkConditionNames();
    }
```

Потенциальная проблема здесь заключается в том, что таблица может переупорядочить условия или изменить их без обновления синтаксического анализатора. Поэтому я выполняю простую проверку имен условий.

```
class TableParser...
    private void checkConditionNames() {
        for (int i = 0; i < result.ConditionNames.Count; i++)
            checkRowName(i, result.ConditionNames[i]);
    }
    private void checkRowName(int row, string name) {
        if (input.cell(row, 0) != name)
            throw new ArgumentException("Неверное имя строки");
    }
```

В ходе загрузки условий в действительности данные из таблицы не извлекаются, помимо имен условий для выполнения проверки. Основное назначение таблицы — предоставить условия и следствия для каждого столбца, который загружается на следующем шаге.

```
class TableParser...
    private void loadColumns() {
        for (int col = 1; col < input.ColumnCount; col++) {
            var conditions = new ConditionBlock()
```

```
Bool3.parse(input.cell(0, col)),
Bool3.parse(input.cell(1, col)),
Bool3.parse(input.cell(2, col)));
var consequences = new FeeResult(
    Int32.Parse(input.cell(3, col)),
    parseBoolean(input.cell(4, col))
);
result.AddColumn( conditions, consequences);
}
}
```

Выбирая ячейки из входной таблицы, необходимо преобразовать строки в соответствующие им значения.

```
class Bool3...
public static Bool3 parse (string s) {
if (s.ToUpper() == "Y") return T;
if (s.ToUpper() == "N") return F;
if (s.ToUpper() == "X") return X;
throw new ArgumentException(
String.Format("<{0}> не преобразуется в Bool3", s));
}

class TableParser...
private bool parseBoolean(string arg) {
if (arg.ToUpper() == "Y") return true;
if (arg.ToUpper() == "N") return false;
throw new ArgumentException(
String.Format("<{0}> не логическое значение",
arg));
}
```

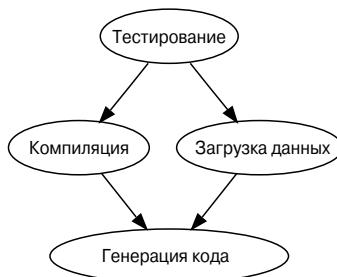

Глава 49

Сеть зависимостей

Dependency Network

Список задач, связанных отношениями зависимости.

Для выполнения некоторой задачи сначала следует выполнить задачи, от которых она зависит



Построение программной системы — обычная задача для разработчиков программного обеспечения. В разных точках этого построения имеются разные задачи, которые вам может потребоваться решить, например скомпилировать программу или выполнить тесты. Если вы хотите запустить тесты, то должны убедиться, что выполнена компиляция последней версии программы. Для успешной же компиляции необходимо выполнить некоторую генерацию кода.

Сеть зависимостей организует функциональность в виде ориентированного ациклического графа (Directed Acyclic Graph — DAG) задач и их зависимостей от других задач. В упомянутом выше случае можно сказать, что задача тестирования зависит от задачи компиляции, а та, в свою очередь, зависит от задачи генерации кода. При запросе на выполнение задачи сначала необходимо найти все задачи, от которых зависит данная, и гарантировать, что они будут выполнены в первую очередь (если это необходимо). Путем перемещения по сети зависимостей можно обеспечить выполнение всех задач, необходимых для успешного выполнения поставленной. Даже если одна задача встречается несколько раз на разных путях зависимостей, все равно можно обеспечить ее однократное выполнение.

49.1. Как это работает

В приведенном выше примере я представил **ориентированное на задачи** описание, в котором сеть представляет собой набор задач с зависимостями между ними. Альтернативным способом является стиль, **ориентированный на продукт**, когда внимание сосредоточивается на продуктах, которые мы собираемся создать, и на зависимостях между ними. Я проиллюстрирую разницу между ними, рассмотрев случай создания программы с помощью некоторой генерации кода с последующей компиляцией. При подходе, ориентированном на задачи, мы говорим, что у нас есть задача генерации кода и задача компиляции, зависящая от задачи генерации кода. При подходе, ориентированном на продукты, мы говорим, что у нас есть выполняемый файл, который создается в процессе компиляции, и некоторые генерируемые исходные файлы. Затем мы указываем зависимости, говоря, что для построения выполняемого файла требуются сгенерированные исходные файлы. Пока что разница между этими подходами может показаться вам слишком незначительной, но, надеюсь, далее она станет для вас понятнее.

Работа сети зависимостей заключается в требовании либо запустить задачу, либо создать продукт. Обычно такой продукт (или задача) называется **целевым**. Затем система находит все предусловия для цели и продолжает поиск предусловий для предусловий... И так до тех пор, пока не будет получен полный список всех транзитивных предусловий, которые должны быть запущены или построены. Затем выполняется вызов всех задач таким образом, чтобы ни одна из них не была вызвана до ее предусловия. Важным свойством этой методики является то, что никакая задача не выполняется более чем один раз, даже если при обходе сети вы сталкиваетесь с ней неоднократно.

Для того чтобы поговорить об этом более предметно, я использую немного больший пример, который к тому же позволит мне избавиться от вездесущего программного обеспечения. Рассмотрим завод по производству магических зелий. Каждое зелье состоит из ингредиентов, которые зачастую составляются из других ингредиентов. Так, для получения живой воды требуется осветленная вода и вытяжка из осьминога (о количествах речь не идет). Чтобы создать вытяжку из осьминога, нам, кроме самого осьминога, потребуется осветленная вода. Чтобы получить ее, нужно высушенное стекло. Мы можем описать связи между этими продуктами в виде ряда зависимостей (рис. 49.1).

```
healthPotion    => clarifiedWater, octopusEssence
octopusEssence => clarifiedWater, octopus
clarifiedWater  => dessicatedGlass
```

В этом случае мы хотим убедиться, что осветленная вода производится при производстве живой воды только один раз, несмотря на то что к ней ведет несколько зависимостей.

Зачастую проще рассматривать все как физические объекты. Так, в данном случае можно представлять себе осветленную воду, как нечто, находящееся в металлическом ковше. Однако та же концепция применима и к информационным продуктам. В этом случае мы могли бы составить производственный план, который включает информацию о том, что необходимо для производства каждого вещества. Нам не понадобится производственный план для осветленной воды до тех пор, пока не нужна сама вода. В конце концов, такой план требует слишком много вычислительных ресурсов, если вся вычислительная техника, которая есть в нашем распоряжении, — это два заколдованных хомяка, передвигающих косточки на счетах.

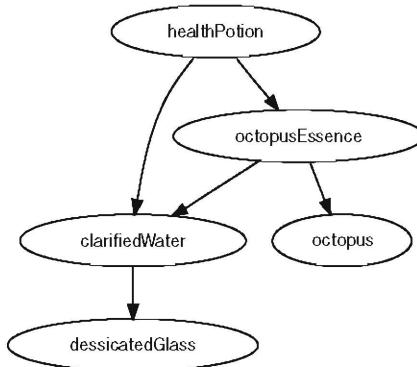


Рис. 49.1. Граф, демонстрирующий зависимости между ингредиентами живой воды

При работе с сетью зависимостей допускаются две основные ошибки, которые могут испортить нам жизнь. Самая серьезная из них — **пропущенное предусловие**, то, что должно было быть сделано, но так и не было сделано. Это серьезная ошибка, которая может привести к неверному результату. Она неприятна еще и потому, что ее может быть трудно обнаружить. Все выглядит корректно работающим, но все данные неправильные, потому что не выполнено предусловие. Другой ошибкой является **ненужное построение**, как, например, повторное составление производственного плана для осветленной воды. В большинстве случаев это приводит лишь к замедлению работы, так как задачи часто являются идемпотентными, но в противном случае можно получить и более серьезные ошибки.

Общей чертой сети зависимостей (в частности, ориентированных на продукты) является то, что каждый продукт отслеживает время последнего обновления. Это может в еще большей степени способствовать сокращению ненужных построений. Когда мы запрашиваем продукт, который должен быть построен, процесс построения выполняется только в том случае, когда время последнего обновления продукта более раннее, чем время обновления любого из предусловий. Чтобы такая технология была работоспособна, первыми должны вызываться предусловия, чтобы в случае необходимости они могли быть перестроены.

Здесь имеется различие между *вызовом* (invoking) задачи и ее *выполнением* (executing). Вызывается каждое транзитивное предусловие, но выполняется оно только при необходимости. Таким образом, если мы вызываем предусловие `octopusEssence`, оно вызывает `octopus` и `clarifiedWater` (которое, в свою очередь, вызывает `dessicatedGlass`). После того как все вызовы закончены, `octopusEssence` сравнивает время последнего изменения планов производства `clarifiedWater` и `octopus` и, если хотя бы одно из этих времен более позднее, чем время последнего изменения плана производства `octopusEssence`, этот план выполняется.

В случае сети, ориентированной на задачи, время последнего изменения часто не используется. Вместо этого каждая задача отслеживает, была ли она выполнена в процессе данного целевого запроса, и выполняется только при первом вызове.

Тот факт, что работа с учетом сохраненных времен последнего изменения оказывает-
ся более легкой, является серьезной причиной для предпочтения ориентации на продукты. Вы можете использовать информацию о времени последнего обновления и в системе, ориентированной на задачи, но для этого каждая задача должна уметь обрабатывать такую информацию. Ориентация на продукты с учетом времени последнего изменения по-
зволяет сети принимать решение о выполнении. Эта возможность не дается даром; она

работает только в том случае, если для неизменных предусловий выход также будет неизменным. Таким образом, все, что может внести изменения в окончательный результат, должно быть объявлено в предусловиях.

Отличие в подходах можно увидеть в системах автоматизации построения. Так, традиционная UNIX-программа `make` ориентирована на продукты (продукты представляют собой файлы), в то время как Java-система `Ant` ориентирована на задачи. Одной из потенциальных проблем ориентированных на продукты систем является то, что не всегда имеется естественный продукт. Хорошим примером этого является выполнение тестов, когда нужно составить что-то вроде отчета об испытаниях. Иногда для корректной работы такой системы приходится создавать псевдорезультат, например пустой файл, который требуется только для того, чтобы можно было воспользоваться временем его последнего изменения.

49.2. Когда это использовать

Сеть зависимостей работает для задач, в которых вы можете разделить вычисления на задания с четко определенными входами и выходами. Способность сети зависимостей выполнять только необходимые задачи делает ее пригодной для ресурсоемких задач или задач, которые должны предпринять некоторые действия для начала работы, например такие, как удаленные операции.

Как и любая альтернативная модель, когда дела идут плохо, сеть зависимостей часто сложна для отладки. Поэтому важно вести журнал вызовов и выполнений, чтобы видеть, что именно происходит в системе. В сочетании с желанием осуществлять выполнение только при необходимости это вынуждает меня посоветовать использовать для сети только относительно крупные задачи.

49.3. Анализ зелий (C#)

В книгах, посвященных программированию, не так уж часто встречаются примеры, посвященные изготовлению магических зелий. Так что, мне кажется, пришло время осветить деятельность предприятий такого рода. Мои эксперты в предметной области говорят мне, что в этом очень конкурентном бизнесе происходит постоянное совершенствование рецептуры. Это приводит к проблеме, связанной с тем, что для контроля качества и получения технических характеристик (профиля) вещества требуется выполнять различные анализы — зачастую очень дорогие и трудоемкие. Поэтому повторять анализ всякий раз при изготовлении зелья не получается, и вместо этого они выполняются только при изменении рецептуры. Кроме того, каждое вещество в производственной цепи может привести к изменениям характеристик других веществ, в которые оно входит, так что при анализе некоторого продукта я должен убедиться в наличии обновленных результатов анализов для всех веществ, которые используют исходное вещество.

Итак, возьмем, например, живую воду. При ее производстве на вход поступает освещенная вода (для которой входом является высушенное стекло). Чтобы проанализировать живую воду, начинать нужно с ее входа (освещенной воды). Если для нее остается действующим рецепт, который я использовал на прошлой неделе, то мне не нужно повторно выполнять анализ освещенной воды. Однако если со времени последнего анализа изменился рецепт для высушенного стекла, то этот анализ следует повторить.

Это типичная сеть зависимостей. Каждое вещество имеет свои входы в качестве предусловий для определения его профиля. Если какие-либо из предусловий вещества со-

держат устаревший профиль, следует выполнить сначала повторный анализ предусловия, а затем — повторный анализ профиля интересующего нас вещества.

Эта сеть зависимостей будет определяться с использованием внутреннего DSL в C#. Вот пример такого сценария.

```
class Script : SubstanceCatalogBuilder {
    Substances octopusEssence, clarifiedWater, octopus,
                dessicatedGlass, healthPotion;
    override protected void doBuild() {
        healthPotion
            .Needs(octopusEssence, clarifiedWater);
        octopusEssence
            .Needs(clarifiedWater, octopus);
        clarifiedWater
            .Needs(dessicatedGlass);
    }
}
```

В этом сценарии использованы такие шаблоны, как Object Scoping (387) и Class Symbol Table (461). О синтаксическом анализе сценария мы поговорим позже, а пока рассмотрим соответствующую семантическую модель (Semantic Model (171)).

49.3.1. Семантическая модель

Структура данных семантической модели (Semantic Model (171)) проста: это граф веществ.

```
class Substance...
    public string Name { get; set; }
    private readonly List<Substance> inputs =
        new List<Substance>();
    private MacGuffinProfile profile;
    private Recipe recipe;

    public void AddInput(Substance s) {
        inputs.Add(s);
    }
```

Каждое вещество имеет рецепт и профиль. Все, что нам нужно о них знать, — это то, что у них есть даты (если я расскажу вам больше, для сохранения тайны мне придется вас убить...).

```
class MacGuffinProfile...
    public DateTime TimeStamp {get; private set;}
    public MacGuffinProfile(DateTime timeStamp) {
        TimeStamp = timeStamp;
    }

class Recipe...
    public DateTime TimeStamp { get; private set; }
    public Recipe(DateTime timeStamp) {
        TimeStamp = timeStamp;
    }
```

Когда запрашивается профиль, в действие вступает сеть зависимостей. Сначала выполняется вызов для каждого входа, так что в результате выполняются вызовы для всех выходов для данного вещества, включая транзитивные. Затем проверяется дата обновления и при необходимости выполняется пересчет.

```

class Substance...
    public MacGuffinProfile Profile {
        get {
            invokeProfileCalculation();
            return profile;
        }
    }
    private void invokeProfileCalculation() {
        foreach (var i in inputs) i.invokeProfileCalculation();
        if (IsOutOfDate)
            profile = profilingService.CalculateProfile(this);
    }
}

```

Поскольку вызов `invoke` для входных параметров осуществляется до своих собственных проверок, можно гарантировать, что для всех входных веществ профиль будет обновлен. Если вещество встречается во входной цепи более одного раза, его вызов будет осуществлен много раз, но его профиль при этом будет пересчитан однократно. Это важно, так как, как вы помните, это занятие весьма дорогостоящее.

В ходе проверки необходимости пересчета используются временные метки профилей и рецептов.

```

class Substance...
    private bool IsOutOfDate {
        get {
            if (null == profile) return true;
            return
                profile.TimeStamp < recipe.TimeStamp ||
                inputs.Any(input =>
                    input.wasUpdatedAfter(profile.TimeStamp));
        }
    }
    private bool wasUpdatedAfter(DateTime d) {
        return profile.TimeStamp > d;
    }
}

```

49.3.2. Синтаксический анализатор

Синтаксический анализатор представляет собой класс таблицы символов (Class Symbol Table (461)) простого вида. Рассмотрим сценарий еще раз.

```

class Script : SubstanceCatalogBuilder {
    Substances octopusEssence, clarifiedWater, octopus,
               dessicatedGlass, healthPotion;
    override protected void doBuild() {
        healthPotion
            .Needs(octopusEssence, clarifiedWater);
        octopusEssence
            .Needs(clarifiedWater, octopus);
        clarifiedWater
            .Needs(dessicatedGlass);
    }
}

```

Сценарий содержится в классе. Поля класса представляют собой различные вещества. Чтобы в сценарии можно было использовать неквалифицированные вызовы функций, я применяю перенос области видимости в объект (Object Scoping (387)). Список веществ составляется в родительском классе.

```

class SubstanceCatalogBuilder...
    public List<Substance> Build() {

```

```
InitializeSubstanceBuilders();
doBuild();
return SubstanceFields.ConvertAll(
    f => ((Substances) f.GetValue(this)).Value);
}
protected abstract void doBuild();
```

Первый шаг построения состоит в заполнении полей экземплярами построителей веществ. Построители веществ для удобочитаемости DSL имеют нестандартное имя `Substances`.

```
class SubstanceCatalogBuilder...
private void InitializeSubstanceBuilders() {
    foreach (var f in SubstanceFields)
        f.SetValue(this, new Substances(f.Name, this));
}
private List<FieldInfo> SubstanceFields {
    get {
        var fields =
            GetType().GetFields(BindingFlags.Instance |
                BindingFlags.NonPublic);
        return Array.FindAll(
            fields,
            f => f.FieldType == typeof (Substances)).ToList();
    }
}
```

Каждый построитель вещества хранит вещество и свободные методы для его заполнения. В данном случае свойства вещества предназначаются и для чтения, и для записи, так что я могу строить значения по ходу дела.

Глава 50

Система правил вывода

Production Rule System

Организация логики посредством набора правил вывода, каждое из которых содержит условие и действие

```
if
    кандидат имеет хорошее происхождение
and
    кандидат – полезный член общества
then
    кандидат достоин интервью

if
    отец кандидата является членом клуба
then
    кандидат имеет хорошее происхождение

if
    кандидат знает английский
and
    кандидат имеет доход 10 тысяч в год
then
    кандидат – полезный член общества
```

Имеется масса ситуаций, которые легко рассматривать как набор условных критериев. Если вы проверяете некоторые данные, то можете рассматривать каждую проверку как условие, приводящее к ошибке, если оно ложно. Принятие на определенную должность также часто можно рассматривать как цепочку условий, успешно пройдя по которой, кандидат получает право занять эту должность. Диагностика поломки также может рассматриваться как ряд вопросов, причем каждый из них может вызывать новые вопросы (и, надеюсь, приводить к обнаружению причины неполадки).

Вычислительная модель системы правил вывода реализует концепцию набора правил, в котором каждое правило имеет условие и последующее действие. Система применяет правила к данным с помощью ряда циклов; каждый цикл выявляет правила, условия которых соответствуют данным, а затем выполняет действия этих правил. Обычно системы правил вывода лежат в основах экспертных систем.

50.1. Как это работает

Базовая структура правил в системе правил вывода довольно проста. У вас есть набор правил, в котором каждое правило имеет условие и действие. Условие представляет собой логическое выражение. Действие может быть чем угодно, но обычно оно ограничено контекстом системы правил вывода. Например, если система правил вывода занимается только проверкой, то соответствующие действия ограничиваются генерацией ошибки, так что каждое действие может лишь указывать, какая ошибка должна быть сгенерирована и какие могут быть предоставлены сопутствующие данные.

Более сложная часть системы правил вывода решает, как следует выполнять правила. В случае обобщенных экспертных систем это может оказаться весьма сложным и запутанным делом. Поэтому имеется целое сообщество, занимающееся экспертными системами, и рынок соответствующего инструментария. Но, как это часто бывает, факт сложности обобщенной системы правил вывода вовсе не означает, что вы не можете построить простую систему правил вывода для отдельных случаев.

Система правил вывода обычно размещает все управление выполнением правила в одном компоненте, который часто называют **механизмом правила, процессором логического вывода или планировщиком**. Простой механизм правила работает посредством выполнения серии циклов логических выводов. Цикл начинается с выполнения всех условий имеющихся правил. Каждое правило, условие которого истинно, называется **активизированным**. Система сохраняет список активизированных правил, именуемый **планом** (agenda). Когда механизм завершает проверку условий правил, он просматривает правила в плане с целью выполнения действий этих правил. Выполнение действия правила называется **запуском** (firing) правила.

Последовательность запуска правил может определяться различными способами. Простейший подход заключается в запуске правил в произвольной последовательности. В этом случае последовательность написания правил не влияет на последовательность запуска, что помогает упростить вычисления. Другой подход заключается в запуске правил всегда в том же порядке, в котором они определены в системе. Хорошим примером этого являются правила фильтрации в системе электронной почты. Вы определяете фильтры с учетом того, что первое же соответствие приведет к обработке письма, и все более поздние правила, которые также могли бы соответствовать этому письму, запущены не будут.

Еще один подход заключается в определении правил с приоритетом, который в области экспертных систем часто называется **особенностью** (salience). В этом случае механизм будет отбирать из плана правила с наивысшим приоритетом, чтобы запустить их первыми. Использование приоритетов часто считается плохим тоном; если вы планируете обильно применять приоритеты, подумайте, действительно ли система правил вывода — подходящая вычислительная модель для вашей задачи и не лучше ли прибегнуть к другой модели.

Еще одной вариацией механизма является выбор, следует ли проверять правила для активизации после запуска каждого правила или лучше запустить все правила из плана перед повторной проверкой. В зависимости от того, как структурированы правила, этот выбор может влиять на поведение системы.

Просматривая базу правил, вы часто обнаруживаете в ней различные группы правил, которые представляют собой логические части общей задачи. В этом случае имеет смысл разделить правила на отдельные **наборы правил** и вычислить их в определенном порядке. Так, для некоторых заданных правил, которые выполняют базовую проверку данных, и для правил, которые определяют уровень квалификации, вы можете предпочесть сна-

чала выполнять набор правил проверки, а затем переходить к правилам определения уровня квалификации только в том случае, если при выполнении первого набора правил ошибок обнаружено не было.

50.1.1. Цепочки выводов

Распространенный случай серии правил проверки представляет собой простейшие системы правил вывода. В этом случае сканируются все правила, и те из них, которые запускаются, сообщают об ошибках или предупреждениях в журнал некоторого вида либо с помощью уведомления `Notification` (205). Одного цикла активизации и запуска достаточно, поскольку действия правил не изменяют состояния данных, с которыми работает система правил вывода.

Однако зачастую действия правил приводят к изменению состояния. В этом случае необходимо заново пересмотреть условия правил, чтобы увидеть, не появились ли среди них такие, которые стали после этого истинными и которые в этом случае следует включить в план. Такое взаимодействие правил известно как **прямое умозаключение** (forward chaining). Вы можете начать с некоторых фактов, с помощью правил вывести дополнительные факты, которые активизируют новые правила, которые позволяют вывести дополнительные факты, и т.д. Механизм останавливается только тогда, когда в плане больше нет правил.

Для простой последовательности, показанной в начале главы, сначала будут проверены все правила. Если два последних правила активизируются и запускаются, то после этого активизируется и запускается первое правило.

Другой подход работает в обратном направлении. В этом случае работа начинается с цели, и база правил изучается для того, чтобы выяснить, какие правила в ней имеют действия, могущие при запуске сделать цель истинной. Затем берем эти правила и, в свою очередь, делаем их подцелями, чтобы найти правила, которые их поддерживают. Такой стиль называется **обратным умозаключением** (backward chaining). Из-за своей сложности он менее распространен в простых системах правил вывода. Поэтому здесь я рассматриваю прямые умозаключения или механизмы без цепочек выводов.

50.1.2. Противоречивые выводы

Одним из больших преимуществ правил является то, что каждое из них можно сформулировать независимо, и система правил вывода выводит следствия из них. Но нет в мире совершенства, и мы сталкиваемся с проблемой — что делать, если получатся цепочки умозаключений, которые противоречат друг другу? Представим, например, правило членства в клубе по реконструкции военных сражений времен войны за независимость, которые гласят, что, чтобы присоединиться к свободолюбивой американской революционной армии, вы должны быть старше 18 лет, гражданином США, иметь свою имитацию мушкета и т.д. В разделе 4.7 есть правило, которое гласит, что британские граждане могут вступить в клуб, но только на стороне армии жестоких тиранов. Эта система правил вывода работает в течение нескольких лет, пока в клуб не прихожу я со своим двойным гражданством. Теперь одно правило гласит, что я могу присоединиться к революционной армии, а другое — что ни в коем случае не могу.

Наибольшая опасность заключается в том, что мы можем не заметить этого вообще. Если следствие этих правил заключается в изменении логического значения `isEligibleForRevolutionaryArmy`, то победит то правило, которое будет запущено последним. Если последовательность запусков или приоритеты правил не определены, это может привести к неправильному выводу или даже к различным выводам в зависимости от скрытых свойств последовательности выполнения правил.

Существуют два основных пути работы с противоречивыми правилами. Один из них заключается в такой разработке структуры правил, которая позволит избежать противоречий. Это предполагает гарантию того, что способ выполнения правил позволяет избежать противоречий, возможно, выбором способа, которым правила обновляют данные, или путем организации наборов правил, или с помощью приоритетов. В нашем примере мы могли бы начать с установления всех условий как ложных и позволить им изменяться только на истинные, но не наоборот. Такое соглашение заставит тех, кто не хочет видеть британцев на стороне революционеров, переписать правила. Но нужно быть осторожными, потому что ошибка может вкрасться в правило, которое потенциально в состоянии разрушить весь проект.

Альтернативный подход состоит в записи всех выводов таким образом, чтобы противоречия допускались, что позволит их обнаруживать. В этом случае вместо логического значения для права занять пост в революционной армии необходимо создать отдельный объект `fact`, ключ которого — `eligibilityForRevolutionaryArmy`, а значение — логическая величина. Как только выполнение правил будет закончено, следует просмотреть все факты с интересующим нас ключом и выяснить, нет ли объектов с одним и тем же ключом, но разными значениями. Шаблон *Observation* [11] — один из способов обработки ситуаций такого рода.

В общем случае нужно быть осторожными и не допустить циклов в системе правил вывода, когда несколько правил расположены так, что одно из них заставляет бесконечно выполнять другое. Это может произойти как с противоречивыми правилами, которые спорят друг с другом, так и с правилами, которые попадают в цикл с положительной обратной связью.

Инструменты, предназначенные специально для разработки систем правил вывода, имеют свои методы для решения таких проблем.

50.1.3. Шаблоны в структуре правила

Хотя я встречал разные базы правил, я не могу сказать, что проводились какие-либо разумные исследования вопроса организации систем правил вывода. Однако те немногие базы правил, которые я видел, следовали некоторым общим шаблонам в структуре правил.

Распространенным и одновременно простым случаем являются проверки. Их простота связана с тем, что все правила обычно имеют простое следствие — генерацию ошибки проверки некоторого вида. В проверках мало цепочек выводов (если таковые вообще имеются). Я подозреваю, что большинство людей, которые серьезно работают с системами правил вывода, не считают проверки системой правил из-за их простоты. И конечно же, мне представляется излишним использовать некоторый специализированный инструментарий для таких целей. Однако этот вид простой структуры хорошо подходит для самостоятельного написания.

Определение права участвовать в боях на стороне революционной армии несколько сложнее. При работе с базой правил такого вида вы пытаетесь оценить, подходит ли кандидат по одному или нескольким параметрам. Это может быть система оценки, которая определяет право на получение страховки, или определение системы скидок, которая распространяется на некоторый заказ. В этом случае правила могут быть структурированы как последовательность шагов, в которой правила более низкого уровня ведут к выводам на более высоком уровне. Избежать противоречий можно путем сохранения всех положительных выводов, возможно, с некоторым специальным способом их отмены.

Еще более сложной оказывается система диагностики, когда выполняется поиск основной причины некоторых проблем. Здесь гораздо больше шансов столкнуться с противоречиями, а потому оказывается достаточно важным применение шаблона наподобие *Observation* [11].

50.2. Когда это использовать

Система правил вывода является естественным выбором, когда у вас есть поведение, которое лучше всего выражается в виде набора инструкций “если то”. В действительности написание потока управления такого рода часто оказывается хорошей отправной точкой для развития системы правил вывода.

Большой опасностью системы правил вывода является ее соблазнительность. Небольшой пример прост для понимания и демонстрации не программистам. Но простая демонстрация не в состоянии показать, как с увеличением размера система правил вывода становится все сложнее, в особенности при использовании цепочек выводов. Это может сделать отладку крайне трудной.

Данная проблема часто усугубляется наличием инструментов для механизма правил. Такие инструменты легко “притянуть за уши” для использования во многих неподходящих местах; при этом понимание того, насколько трудно модифицировать систему, приходит тогда, когда она становится слишком большой. Таким образом, это еще один аргумент в пользу самостоятельного построения чего-то простого, что вы сможете самостоятельно настроить для своих нужд, и изучения при этом предметной области и того, как система правил вывода сможет в нее вписаться. После такого изучения вы сможете оценить, стоит ли заменять свою простую систему профессиональным инструментом.

Я не говорю, что механизмы правил — всегда плохая идея, хотя я еще не встречал такого, который бы достаточно хорошо работал. Важно то, что вы должны относиться к ним с осторожностью и отдавать себе отчет в том, что именно вы получаете при их использовании.

50.3. Проверка для членства в клубе (C#)

Проверки представляют собой хороший простой пример системы правил вывода, так как они достаточно распространены и обычно не включают цепочки выводов. В качестве примера возьмем первый этап рассмотрения заявки на вступление в некоторый мнимый викторианский английский клуб. Для обработки заявок я буду использовать два отдельных набора правил. Первый будет выполнять проверку основных данных заявителя, чтобы убедиться в правильности заполнения формы. Второй набор правил будет оценивать право заявителя на интервью, и об этом я расскажу во втором примере.

Вот несколько правил проверки.

- Должно быть указано гражданство
- Должно быть указано учебное заведение
- Требуется положительный годовой доход

50.3.1. Модель

В описываемой здесь модели — две части. Первая, очень простая, часть — данные о персоне, с которыми будут работать правила. Это простой класс данных со свойствами для различной интересующей нас информации.

```
class Person...
    public string Name          { get; set; }
    public University? University { get; set; }
    public int? AnnualIncome   { get; set; }
    public Country? Nationality { get; set; }
```

Перейдем теперь к части обработки правил. Базовая структура механизма проверок представляет собой список правил проверки.

```
class ValidationEngine...
List <ValidationRule> rules = new List<ValidationRule>();

interface ValidationRule {
    void Check(Notification note, Person p);
}
```

Для запуска этого механизма следует выполнить каждое из правил и сохранить результат с применением уведомления (`Notification (205)`).

```
class ValidationEngine...
public Notification Run(Person p) {
    var result = new Notification();
    foreach (var r in rules) r.Check(result, p);
    return result;
}
```

Основное правило проверки принимает предикат и сообщение для записи в случае невыполнения предиката.

```
class ExpressionValidationRule : ValidationRule {
    readonly Predicate<Person> condition;
    readonly string description;
    public ExpressionValidationRule(
        Predicate<Person> condition, string description) {
        this.condition = condition;
        this.description = description;
    }
    public void Check(Notification note, Person p) {
        if (! condition(p))
            note.AddError(String.Format(
                "Проверка '{0}' не прошла.", description));
    }
}
```

Затем можно создать и запустить правила с использованием интерфейса командных запросов с помощью кода наподобие приведенного далее.

```
engine = new ValidationEngine();
engine.AddRule(
    p => p.Nationality != null, "Отсутствует гражданство");
var tim = new Person("Tim");
var note = engine.Run(john);
```

50.3.2. Синтаксический анализатор

В этом примере использован простой внутренний DSL. Для правил я прибегаю к непосредственному применению лямбда-выражений C#. Мой сценарий DSL начинается следующим образом.

```
class ExampleValidation : ValidationEngineBuilder {
    protected override void build() {
        Validate("Годовой доход в наличии")
            .With(p => p.AnnualIncome != null);
        Validate("Годовой доход положителен")
            .With(p => p.AnnualIncome >= 0);
```

Перенос области видимости в объект (**Object Scoping** (387)) позволяет определять проверки с помощью простого вызова метода `Validate`, соединенного в цепочку с методом для определения предиката.

Одновременно с построителем создается и механизм. Метод `Validate` создает дочерний построитель правила для хранения информации правила.

```
abstract class ValidationEngineBuilder {
    public ValidationEngine Engine { get; private set; }
    protected ValidationEngineBuilder() {
        Engine = new ValidationEngine();
        build();
    }
    abstract protected void build();
    protected WithParser Validate(string description) {
        return new ValidationRuleBuilder(description, Engine);
    }
}

class ValidationEngine...
public void AddRule(Predicate<Person> condition,
                    string errorMessage) {
    rules.Add(new ExpressionValidationRule(condition,
                                             errorMessage));
}
interface WithParser {
    void With(Predicate<Person> condition);
}
```

Последовательный интерфейс здесь выглядит излишним, так как у построителя правила имеется только один метод. Но мне кажется, что имя интерфейса помогает передать, что ищет синтаксический анализатор.

50.3.3. Развитие DSL

Эти проверки вполне справляются с передаваемой им логикой, но, надеюсь, написав несколько выражений для сравнения с нулевым значением, вы задумаетесь, нет ли лучшего способа такой проверки. Если такие проверки достаточно распространены в вашем приложении, можете поместить логику проверки на нулевое значение в правиле, так что все, что потребуется сделать в сценарии, — это указать свойство, которое не должно быть нулевым.

Один из способов сделать это (который работает во многих языках программирования) — написать имя свойства в виде строки и использовать механизм отражения для проверки. Вот как эта строка выглядит в сценарии DSL.

```
MustHave("University");
```

Для поддержки описанного способа работы следует расширить модель и синтаксический анализатор.

```
class ValidationEngineBuilder...
protected void MustHave(string property) {
    Engine.AddNotNullRule(property);
}

class ValidationEngine...
public void AddNotNullRule(string property) {
    rules.Add(new NotNullValidationRule(property));
}

class NotNullValidationRule : ValidationRule {
```

```

readonly string property;
public NotNullValidationRule(string property) {
    this.property = property;
}
public void Check(Notification note, Person p) {
    var prop = typeof(Person).GetProperty(property);
    if (null == prop.GetValue(p, null))
        note.AddError("Отсутствует значение {0}", property);
}

```

Здесь следует подчеркнуть, что изменять семантическую модель (*Semantic Model* (171)) для реализации поддержки не нужно. Вместо этого можно легко разместить код в построителе.

```

class ValidationEngineBuilder...
protected void MustHaveALT(string property) {
    PropertyInfo prop = typeof(Person).GetProperty(property);
    Engine.AddRule(p => prop.GetValue(p, null) != null,
        String.Format("Должно иметься свойство {0}",
            property));
}

```

Часто по привычке такая логика размещается в построителе, но я призываю вас не поддаваться ей. Если я помешу логику в семантическую модель, она будет в состоянии гораздо лучше использовать информацию, так как будет знать, что она делает. Такое решение, например, позволяет семантической модели генерировать код для проверки в случае, если вы захотите встроить в форму Javascript. Но даже без подобной необходимости лично я предпочитаю размещать в семантической модели как можно больше логики. Это не требует больших трудозатрат по сравнению с размещением в построителе, но зато при этом знания о правилах размещаются там, где они наиболее полезны.

При всех преимуществах у строкового аргумента имеются недостатки, особенно в средах со статической типизацией наподобие C#. Было бы лучше получить имя свойства с помощью некоторого механизма C#, чтобы можно было воспользоваться автозаполнением и статическими проверками.

К счастью, в C# это можно сделать с помощью лямбда-выражений. Сценарий DSL для проверки равенства нулевому значению выглядит следующим образом.

```
MustHave(p => p.Nationality);
```

Эту возможность я также реализую в модели с простым вызовом из построителя.

```

class ValidationEngineBuilder...
protected void
    MustHave<T>(Expression<Func<Person, T>> expression) {
        Engine.AddNotNullRule(expression);
    }

class ValidationEngine...
public void
    AddNotNullRule<T>(Expression<Func<Person, T>> e) {
        rules.Add(new NotNullValidationRule<T>(e));
    }

class NotNullValidationRule<T> : ValidationRule {
    readonly Expression<Func<Person, T>> expression;
    public NotNullValidationRule(
        Expression<Func<Person, T>> expression) {
            this.expression = expression;
    }
}

```

```

public void Check(Notification note, Person p) {
    var lambda = expression.Compile();
    if (lambda(p) == null)
        note.AddError("Отсутствует значение {0}", expression);
}
}

```

Я использовал здесь не только само лямбда-выражение, но и его текст, который понадобится мне при выводе сообщения об ошибке при неудачной проверке.

50.4. Правила избрания: расширение членства в клубе (C#)

В предыдущем примере были показаны правила проверки ввода в форму заявки на получение членства в нашем вымышленном клубе. В этом же примере рассматриваются правила избрания, в связи с чем он немного сложнее, так как включает некоторые прямые умозаключения. Правило верхнего уровня гласит, что кандидат считается достойным интервью, если он имеет хорошее происхождение и является полезным членом общества.

```

class ExampleRuleBuilder : EligibilityEngineBuilder {
    protected override void build() {
        Rule("Интервью, если устраивает происхождение и польза")
            .When(a => a.IsOfGoodStock && a.IsProductive)
            .Then(a => a.MarkAsWorthyOfInterview());
    }
}

```

Как и в предыдущем примере, я использую внутренний DSL с переносом области видимости в объект (*Object Scoping* (387)) с применением суперкласса, но подробности синтаксического анализа я опишу позже. Существуют различные правила для определения того, что же такое хорошее происхождение и что значит быть полезным членом общества. Есть два способа отвечать критерию хорошего происхождения — быть сыном члена клуба или иметь подходящее воинское звание.

```

Rule("Отец - член клуба означает хорошее происхождение")
    .When(a => a.Candidate.Father.IsMember)
    .Then(a => a.MarkOfGoodStock());
Rule("Воинское звание означает хорошее происхождение")
    .When(a => a.IsMilitarilyAccomplished)
    .Then(a => a.MarkOfGoodStock());
Rule("Надо быть как минимум капитаном")
    .When(a => a.Candidate.Rank >= MilitaryRank.Captain)
    .Then(a => a.MarkAsMilitarilyAccomplished());
Rule("Оксфорда или Кембриджа вполне достаточно")
    .When(a => a.Candidate.University == University.Cambridge
          || a.Candidate.University == University.Oxford)
    .Then(a => a.MarkOfGoodStock());

```

Эти правила иллюстрируют важное свойство системы правил вывода — правила открыты в том смысле, что, например, можно легко добавить новые правила, которые гласят, что значит иметь хорошее происхождение. Я могу добавить новые правила, не изменения тех, которые уже имеются в системе. Недостатком же является то, что в базе правил нет единого места, где я точно могу найти *все* условия. Один из способов справиться с этим — создание инструмента, способного найти все правила, которые имеют следствием вызов *MarkOfGoodStock*.

Другое требование к кандидату на собеседование — быть полезным членом общества. Так, в высшем английском обществе интересуются, сколько вы зарабатываете.

```

Rule("Полезный англичанин")
    .When(a => a.Candidate.Nationality == Country.England
          && a.Candidate.AnnualIncome >= 10000)
    .Then(a => a.MarkAsProductive());
Rule("Полезный шотландец")
    .When(a => a.Candidate.Nationality == Country.Scotland
          && a.Candidate.AnnualIncome >= 20000)
    .Then(a => a.MarkAsProductive());
Rule("Полезный американец")
    .When(a => a.Candidate.Nationality == Country.UnitedStates
          && a.Candidate.AnnualIncome >= 80000)
    .Then(a => a.MarkAsProductive());
Rule("Полезный военный")
    .When(a => a.IsMilitarilyAccomplished
          && a.Candidate.AnnualIncome >= 8000)
    .Then(a => a.MarkAsProductive());

```

Английский клуб, естественно, предпочитает англичан, но за хорошую сумму за английского джентльмена может сойти даже американец.

Глядя на используемые здесь шаблоны, мы видим, что список правил определяется с помощью последовательности функций (*Function Sequence* (357)). Предоставление подробной информации для каждого правила использует соединение методов *When* и *Then* в цепочки (*Method Chaining* (375)), а вложенные замыкания (*Nested Closure* (403)) применены для записи содержимого условия и действия каждого правила.

50.4.1. Модель

Модель принятия в члены клуба подобна модели проверки наличия данных, но немного сложнее для того, чтобы обрабатывать различные следствия и прямые умозаключения. Первая часть представляет собой структуру данных, используемую для отчета о результатах выполнения логики.

```

class Application...
    public Person Candidate { get; private set; }
    public bool IsWorthyOfInterview { get; private set; }
    public void MarkAsWorthyOfInterview()
    {
        IsWorthyOfInterview = true;
    }
    public bool IsOfGoodStock { get; private set; }
    public void MarkOfGoodStock() { IsOfGoodStock = true; }
    public bool IsMilitarilyAccomplished { get; private set; }
    public void MarkAsMilitarilyAccomplished()
    {
        IsMilitarilyAccomplished = true;
    }
    public bool IsProductive { get; private set; }
    public void MarkAsProductive() { IsProductive = true; }
    public Application(Person candidate)
    {
        this.Candidate = candidate;
        IsOfGoodStock = false;
        IsWorthyOfInterview = false;
        IsMilitarilyAccomplished = false;
        IsProductive = false;
    }
}

```

Подобно рассматривавшемуся ранее классу *Person*, это в основном простой класс данных, но у него несколько необычная структура. Все его свойства — логические, изначально имеющие значение *false*, которое может быть изменено только на *true*, но не обратно на *false*. Это обеспечивает структуру системы правил, позволяющую избежать незамеченных противоречий.

Каждое правило получает пару замыканий: одно — для условия, другое — для следствия, а также текстовое описание.

```
class EligibilityRule...
    public string Description { get; private set; }
    readonly Predicate<Application> condition;
    readonly Action<Application> action;
    public EligibilityRule(string description,
                           Predicate<Application> condition,
                           Action<Application> action)
    {
        this.Description = description;
        this.condition = condition;
        this.action = action;
    }
```

Я могу загрузить правила в список.

```
class EligibilityRuleBase {
    private List<EligibilityRule> initialRules =
        new List<EligibilityRule>();
    public List<EligibilityRule> InitialRules {
        get { return initialRules; }
    }
    public void AddRule(string description,
                        Predicate<Application> condition,
                        Action<Application> action)
    {
        initialRules.Add(new EligibilityRule(description,
                                              condition,
                                              action));
    }
```

Запуск механизма несколько сложнее из-за прямых умозаключений. Основной цикл проверяет правила, помещает те из них, которые могут быть активизированы, в план, запускает правила из плана, а затем проверяет, нет ли новых правил, которые могут быть активизированы.

```
class EligibilityEngine...
    public void Run() {
        activateRules();
        while (agenda.Count > 0) {
            fireRulesOnAgenda();
            activateRules();
        }
    }
```

Я использую некоторые дополнительные структуры данных для отслеживания запуска правил.

```
class EligibilityEngine...
    public readonly EligibilityRuleBase ruleBase;
    List<EligibilityRule> availableRules =
        new List<EligibilityRule>();
    List<EligibilityRule> agenda =
        new List<EligibilityRule>();
    List<EligibilityRule> firedLog =
        new List<EligibilityRule>();
    readonly Application application;
    public EligibilityEngine(EligibilityRuleBase ruleBase,
                            Application application) {
        this.ruleBase = ruleBase;
```

```

    this.application = application;
    availableRules.AddRange(ruleBase.InitialRules);
}
}

```

Я копирую правила из базы правил в список доступных правил. Когда правило активизируется, я удаляю его из этого списка (поэтому оно не может быть активизировано вновь) и вношу в план.

```

class EligibilityEngine...
private void activateRules() {
    foreach (var r in availableRules)
        if (r.CanActivate(application))
            agenda.Add(r);
    foreach (var r in agenda)
        availableRules.Remove(r);
}

class EligibilityRule...
public bool CanActivate(Application a) {
    try {
        return condition(a);
    } catch(NullReferenceException) {
        return false;
    }
}
}

```

Я “отправляю” нулевые ссылки в `CanActivate`, рассматривая их, просто как неудачи активизации. Это позволяет мне при создании правила записать условное выражение вида `anApplication.Candidate.Father.IsMember` без необходимости делать проверки на нулевые значения — соответствующая ответственность перенесена в модель.

Запуская правило, я удаляю его из плана и помещаю в журнал запущенных правил. Позже я смогу использовать этот журнал для диагностических целей.

```

class EligibilityEngine...
private void fireRulesOnAgenda() {
    while (agenda.Count > 0) {
        fire(agenda.First());
    }
}
private void fire(EligibilityRule r) {
    r.Fire(application);
    firedLog.Add(r);
    agenda.Remove(r);
}

class EligibilityRule...
public void Fire(Application a) {
    action(a);
}
}

```

50.4.2. Синтаксический анализатор

Как я говорил ранее, для построителя правила избрания я использую структуру, подобную применявшейся для правил проверки — с переносом области видимости в объект (*Object Scoping* (387)) с использованием суперкласса.

```

abstract class EligibilityEngineBuilder {
    protected EligibilityEngineBuilder() {
        RuleBase = new EligibilityRuleBase();
    }
}

```

```

        build();
    }
    public EligibilityRuleBase RuleBase { get; private set; }
    abstract protected void build();
}

```

Я определяю правило как последовательность функций (Function Sequence (357)), состоящую из вызовов Rule.

```

class EligibilityEngineBuilder...
protected WhenParser Rule(string description) {
    return new EligibilityRuleBuilder(RuleBase, description);
}

class EligibilityRuleBuilder : ThenParser, WhenParser{
    EligibilityRuleBase RuleBase;
    string description;
    Predicate<Application> condition;
    Action<Application> action;
    public EligibilityRuleBuilder(EligibilityRuleBase ruleBase,
                                  string description) {
        this.RuleBase = ruleBase;
        this.description = description;
    }
}

```

Я использую соединение методов в цепочки (Method Chaining (375)) для дочернего построителя правила с последовательными интерфейсами для работы остальной части правила. Сначала вызывается When.

```

class EligibilityEngineBuilder...
interface WhenParser {
    ThenParser When(Predicate<Application> condition);
}

class EligibilityRuleBuilder...
public ThenParser When(Predicate<Application> condition) {
    this.condition = condition;
    return this;
}

```

Затем следует вызов Then.

```

class EligibilityEngineBuilder...
interface ThenParser {
    void Then(Action<Application> action);
}

class EligibilityRuleBuilder...
public void Then(Action<Application> action) {
    this.action = action;
    loadRule();
}
private void loadRule() {
    RuleBase.AddRule(description, condition, action);
}

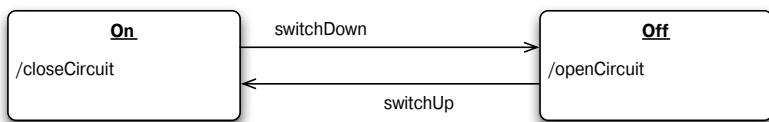
```


Глава 51

Конечный автомат

State Machine

*Модель системы как набора явных состояний
с переходами между ними*



Многие системы реагируют на входные сигналы по-разному, в зависимости от некоторых внутренних свойств. Иногда полезно классифицировать эти различные внутренние состояния и описывать различия в реакции и то, что заставляет систему переходить из одного состояния в другое. Для описания и, возможно, для управления этим поведением может использоваться конечный автомат.

51.1. Как это работает

Конечный автомат — распространенная сущность как в программном обеспечении, так и в дискуссиях о программировании. Именно поэтому я использовал его в примере, открываящем эту книгу. Степень применения конечного автомата варьируется в зависимости от ситуации, равно как и формы используемых автоматов.

Давайте рассмотрим другой пример, не столь простой и ясный, как во введении. Рассмотрим систему обработки заказов. Создав заказ, я могу свободно добавлять в него элементы или удалять их или полностью отменить заказ. В какой-то момент я должен оплатить заказ. Как только я это сделаю, заказ будет передан для выполнения и отправки оплаченного товара. Но и в этот момент я еще смогу добавить или удалить элементы заказа или отменить весь заказ. Но как только товар будет отправлен, я не смогу сделать ничего из вышеперечисленного.

Все сказанное можно описать с помощью диаграммы переходов между состояниями, показанной на рис. 51.1.

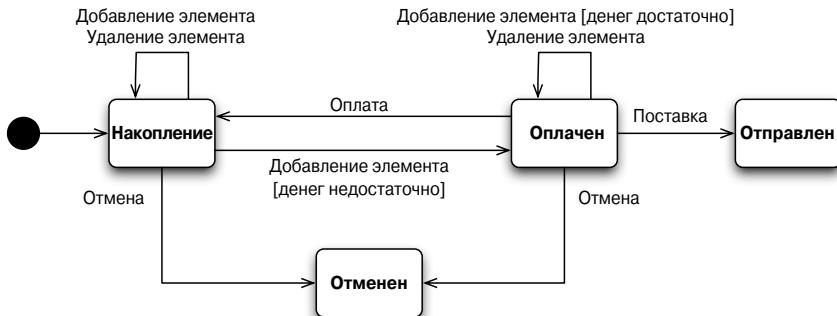


Рис. 51.1. UML-диаграмма конечного автомата обработки заказа

Сейчас мне нужно пояснить, что же такое “состояние”. В общем случае мы часто говорим о состоянии объекта как сочетании значений его свойств. В приведенном примере удаление элемента из заказа изменяет его состояние. Однако диаграмма состояния конечного автомата не отражает все возможные его состояния, ограничиваясь только небольшим их количеством. Это состояния, которые являются интересными с точки зрения модели в том плане, что они влияют на поведение системы. Об этом меньшем множестве состояний я буду говорить как о **состояниях автомата**. Так, при удалении элемента из заказа изменяется состояние заказа, но не состояние его автомата.

Такая модель состояний представляет собой полезный способ рассмотрения поведения заказа, но это не означает, что мы хотим иметь модель конечного автомата в своей программе. Модель может помочь нам понять, что при вызове метода `cancel` нам необходимо убедиться, что мы находимся в соответствующем состоянии. Но это может быть просто простая проверяющая инструкция в методе `cancel`.

Аналогично отслеживание состояния автомата заказа может выполняться с помощью соответствующего поля заказа, но может и полностью выводиться. Вы можете определить, находитесь ли вы в состоянии “оплачено”, сравнивая сумму платежа с общей стоимостью заказа. Диаграмма может оставаться полезным средством визуализации работы заказа, но не обязательной моделью, которая будет проявляться в программном обеспечении.

Конечные автоматы, как и другие альтернативные вычислительные модели, имеют несколько разновидностей, у которых есть много общих элементов, но при этом имеются и значительные различия. Начну с общих элементов. Суть конечного автомата — в понятии множества состояний, в которых может находиться этот автомат. Затем мы можем определить несколько переходов из каждого состояния, в котором каждый переход вызывается событием и приводит автомат в целевое состояние. Это целевое состояние часто (но не обязательно) представляет собой другое состояние, отличное от исходного. Результатирующее поведение машины представляет собой определение состояний и событий, вызывающих переходы между состояниями.

На рис. 51.1 показано схематическое представление такой сети. Состояние накопления имеет четыре определенных для него перехода. Они гласят, что если автомат находится в этом состоянии и получает событие отмены, он переходит в состояние “отменен”. Событие оплаты приводит в состояние “оплачен”, добавление или удаление элемента приводит обратно в то же состояние накопления. Добавление и удаление элемента представляют собой разные переходы, хотя они и ведут в одно и то же целевое состояние.

Общим вопросом, связанным с конечными автоматами, является вопрос о реакции на событие, не определенное для состояния, в котором в настоящий момент находится

конечный автомат. В зависимости от приложения, такие события могут либо рассматриваться как ошибки, либо игнорироваться.

На рис. 51.1 также представлено еще одно понятие — защищенного перехода. Когда в состоянии “оплачено” конечный автомат получает событие добавления элемента, переход зависит от того, достаточно ли денег. Логические условия для переходов не должны перекрываться, так как в противном случае конечный автомат не сможет решить, в какое целевое состояние он должен перейти. Защищенные переходы — это необязательный атрибут конечных автоматов. Так, во вступительном примере их нет.

На рис. 51.1 описаны несколько состояний автомата и события, которые вызывают перемещение автомата между ними. Но это все еще пассивная модель, так как она не вызывает каких-либо действий, которые приводят к изменениям в системе. Чтобы применить адаптивную модель (*Adaptive Model* (478)) к конечному автомату, нужен способ привязать к автомата действия. За время существования конечных автоматов для этого придумано несколько схем. Есть два места, к которым логично привязать действия: переходы и состояния. Привязка действий к переходу означает, что действие выполняется при переходе. Привязка действий к состоянию чаще всего означает, что действие выполняется при входе в это состояние. Однако можно встретить и действия, связанные с выходами из состояния. Некоторые конечные автоматы допускают внутренние действия, которые вызываются при получении события в этом состоянии — как при переходе из состояния в это же состояние, но без вызова действия, связанного со входом в состояние.

Различные подходы к привязке действий годятся для разных задач (и разных программистов). Я не даю каких-то строгих рекомендаций, кроме как использовать наиболее простой способ, пригодный для моделирования требуемого поведения. Многие реализации методов конечных автоматов стремятся к максимальной выразительности автомата, такие как очень выразительная модель конечного автомата, используемая UML. Но небольшие конечные автоматы, подходящие для DSL, зачастую хорошо работают с существенно более простыми моделями.

51.2. Когда это использовать

Мной овладевает ужасное чувство, когда все, что я могу сказать, — это то, что вы должны использовать конечный автомат, когда интересующее вас поведение напоминает конечный автомат, т.е. когда у вас есть ощущение перемещения из состояния в состояние, вызываемое событиями. Во многих отношениях наилучший способ понять, подходит ли для вас конечный автомат, — попытаться набросать на бумаге эскиз и, если он хорошо выглядит, попробовать его в действии.

Имеется одна опасная область, в которой может пригодиться капля теории языков, которую я приводил ранее (“Регулярные, контекстно-свободные и контекстно-зависимые грамматики”, с. 112). Вспомните, что конечные автоматы ограничены синтаксическим анализом регулярных грамматик, т.е. они не могут обрабатывать соответствие произвольно вложенных скобок. Вы можете столкнуться с этой проблемой, если в интересующем вас поведении имеется что-либо подобное.

51.3. Контроллер тайника (Java)

Часто в этой книге, начиная с введения, я использовал в качестве примера простой конечный автомат. Для всех случаев, в которых он упоминался, я использовал одну семантическую модель (*Semantic Model* (171)), описанную во введении. Поведение со-

стояния не использует защищенных переходов и связывает очень простые действия со входом в состояние. Действия просты в том смысле, что они не включают выполнения блока произвольного кода, а только отправляют сообщения с числовым кодом. Это упрощает модель конечного автомата и DSL для управления им (что очень важно для такого рода примеров).

Часть VI

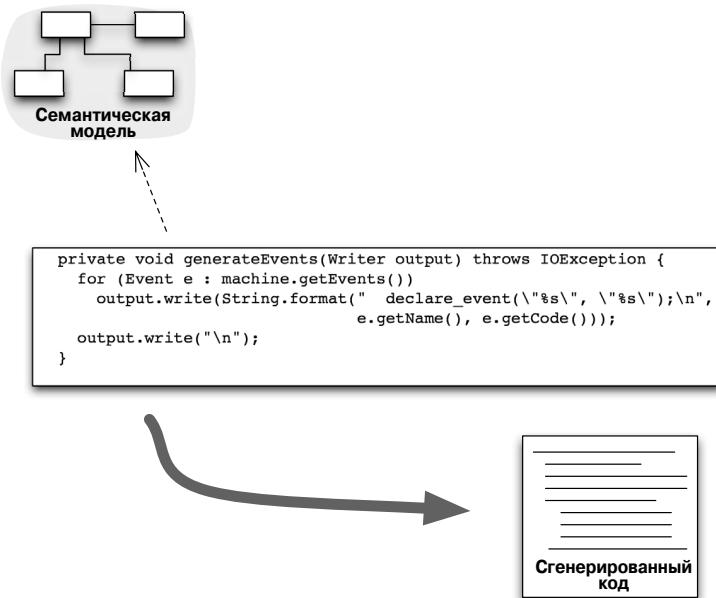
Генерация кода

Глава 52

Генерация с помощью преобразователя

Transformer Generation

Генерация кода путем написания преобразователя, который обходит входную модель и генерирует вывод



52.1. Как это работает

Генерация с помощью преобразователя включает написание программы, которая получает на входе семантическую модель ([Semantic Model \(171\)](#)), а на выходе дает исходный текст для целевой языковой среды. Я предлагаю рассматривать преобразователи

как управляемые вводом и управляемые выводом. Управляемые выводом преобразования начинаются с требующегося вывода и погружаются во вход для сбора данных, в которых они нуждаются. Преобразование, управляемое вводом, проходит по входным структурам данных и генерирует вывод.

В качестве примера рассмотрим генерацию веб-страницы на основе каталога продукции. Подход, управляемый выводом, начинает со структуры веб-страницы, возможно, с помощью следующей подпрограммы.

```
renderHeader();
renderBody();
renderFooter();
```

Преобразование, управляемое вводом, начинает с просмотра и обхода входных структур данных примерно таким образом.

```
foreach (prod in products) {
    renderName(prod);

    foreach (photo in prod.photos) {
        renderPhoto(photo);
    }
}
```

Часто преобразователи используют сочетание двух указанных подходов. Я регулярно попадаю в ситуации, когда внешняя логика управляет выходом, но вызывает подпрограммы, которые являются в большей степени управляемыми входными данными. Внешняя логика описывает структуру выходного документа, разделяя его на логические разделы, в то время как внутренняя часть генерирует вывод на основании конкретного вида входных данных. В любом случае я считаю полезным рассматривать каждую процедуру преобразования как управляемую вводом либо выходом и ясно сознавать, какой именно подход используется.

Многие преобразования могут проследовать непосредственно от семантической модели к целевому исходному тексту, но в случае более сложных преобразований может оказаться полезным разбиение на несколько шагов. Двухступенчатое преобразование, например, обходит входную модель и создает выходную. Эта выходная модель является именно моделью, а не исходным текстом, но более ориентированной на генерацию вывода. Затем на втором шаге выполняются обход этой выходной модели и генерация окончательного исходного текста. Многоступенчатое преобразование полезно, когда преобразование сложно или когда необходимо получить несколько исходных текстов из одних и тех же входных данных. В последнем случае на первой стадии преобразования можно создать единую выходную модель с общими элементами. Разница между выходными исходными текстами может заключаться в различных вторых стадиях преобразования.

При многоступенчатом подходе можно также смешивать разные методы, например использовать генерацию с помощью преобразователя для первого этапа и шаблонную генерацию (Templated Generation (529)) — для второго.

52.2. Когда это использовать

Одноступенчатая генерация с помощью преобразователя является хорошим выбором, если выходной исходный текст связан со входной моделью простым взаимоотношением, а большая часть выходного текста генерируется. В этом случае преобразователь достаточно легко написать, и не нужно применять шаблонные инструменты.

Генерация с помощью преобразователя в несколько этапов может быть очень полезной при более сложных взаимоотношениях между входом и выходом, поскольку каждый этап может обрабатывать различные аспекты проблемы.

Используя генерацию, осведомленную о модели (Model-Aware Generation (545)), обычно можно наполнить модель простой последовательностью вызовов, которые легко генерировать с помощью преобразователя.

52.3. Контроллер тайника (Java генерирует C)

Генерации, осведомленной о модели (Model-Aware Generation (545)), часто сопутствует генерация с помощью преобразователя, так как четкое разделение сгенерированного и статического кодов позволяет любой части сгенерированного кода включать очень небольшой статический код. Так что в данном случае я буду генерировать код для контроллера тайника из примера для генерации, осведомленной о модели. Чтобы вам не пришлось зря листать страницы, я приведу здесь код, который мне нужно сгенерировать.

```
void build_machine() {  
  
    declare_event("doorClosed", "D1CL");  
    declare_event("drawerOpened", "D2OP");  
    declare_event("lightOn", "L1ON");  
    declare_event("doorOpened", "D1OP");  
    declare_event("panelClosed", "PNCL");  
  
    declare_command("lockDoor", "D1LK");  
    declare_command("lockPanel", "PNLK");  
    declare_command("unlockPanel", "PNUL");  
    declare_command("unlockDoor", "D1UL");  
  
    declare_state("idle");  
    declare_state("active");  
    declare_state("waitingForDrawer");  
    declare_state("unlockedPanel");  
    declare_state("waitingForLight");  
  
    /* Тело состояния idle */  
    declare_action("idle", "unlockDoor");  
    declare_action("idle", "lockPanel");  
    declare_transition("idle", "doorClosed", "active");  
  
    /* Тело состояния active */  
    declare_transition("active", "lightOn",  
                      "waitingForDrawer");  
    declare_transition("active", "drawerOpened",  
                      "waitingForLight");  
  
    /* Тело состояния waitingForDrawer */  
    declare_transition("waitingForDrawer", "drawerOpened",  
                      "unlockedPanel");  
  
    /* Тело состояния unlockedPanel */  
    declare_action("unlockedPanel", "unlockPanel");  
    declare_action("unlockedPanel", "lockDoor");  
    declare_transition("unlockedPanel", "panelClosed",  
                      "idle");  
  
    /* Тело состояния waitingForLight */
```

```

declare_transition("waitingForLight", "lightOn",
                  "unlockedPanel");

/* Переходы сбрасывающего события */
declare_transition("idle", "doorOpened", "idle");
declare_transition("active", "doorOpened", "idle");
declare_transition("waitingForDrawer", "doorOpened",
                  "idle");
declare_transition("unlockedPanel", "doorOpened",
                  "idle");
declare_transition("waitingForLight", "doorOpened",
                  "idle");
}

```

Вывод, который мне предстоит сгенерировать, имеет очень простую структуру, очевидную из способа построения внешней подпрограммы генератора.

```

class StaticC_Generator...
public void generate(PrintWriter out) throws IOException {
    this.output = out;
    output.write(header);
    generateEvents();
    generateCommands();
    generateStateDeclarations();
    generateStateBodies();
    generateResetEvents();
    output.write(footer);
}
private PrintWriter output;

```

Это типичный код внешней подпрограммы управляемого выводом преобразователя. Можно просто пройти по всем этим шагам в порядке их поступления. В качестве заголовка выводится статический текст, который должен располагаться в верхней части исходного файла.

```

class StaticC_Generator...
private static final String header =
    "#include \"sm.h\"\n" +
    "#include \"sm-pop.h\"\n" +
    "\nvoid build_machine() {\n";

```

Я создаю генератор с конечным автоматом, с которым он будет работать.

```

class StaticC_Generator...
private StateMachine machine;
public StaticC_Generator(StateMachine machine) {
    this.machine = machine;
}

```

Сначала я использую его для генерации объявлений событий.

```

class StaticC_Generator...
private void generateEvents() throws IOException {
    for (Event e : machine.getEvents())
        output.printf(" declare_event(\"%s\", \"%s\");\n",
                      e.getName(), e.getCode());
    output.println();
}

```

Объявления команд и состояний столь же просты.

```

class StaticC_Generator...
private void generateCommands() throws IOException {
    for (Command c : machine.getCommands())
        output.printf(" declare_command(\"%s\", \"%s\");\n",
                      c.getName(), c.getCode());
    output.println();
}

private void generateStateDeclarations() throws IOException {
    for (State s : machine.getStates())
        output.printf(" declare_state(\"%s\");\n", s.getName());
    output.println();
}

```

Далее я генерирую тела (действия и переходы) для каждого состояния. В этом случае я должен объявить все состояния до переходов, так как иначе я получу сообщение об ошибке из-за наличия опережающей ссылки.

```

class StaticC_Generator...
private void generateStateBodies() throws IOException {
    for (State s : machine.getStates()) {
        output.printf(" /* Тело состояния %s */\n",
                      s.getName());
        for (Command c : s.getActions()) {
            output.printf(" declare_action(\"%s\", \"%s\");\n",
                          s.getName(), c.getName());
        }
        for (Transition t : s.getTransitions()) {
            output.printf(
                " declare_transition(\"%s\", \"%s\", \"%s\");\n",
                t.getSource().getName(),
                t.getTrigger().getName(),
                t.getTarget().getName());
        }
        output.println();
    }
}

```

Здесь также продемонстрирован переход к стилю управления входом. Код, порожденный в каждом конкретном случае, следует структуре входной модели. Это нормально, так как не имеет значения, в каком порядке я объявляю действия и переходы. Этот код также показывает генерацию комментариев с использованием динамических данных.

Наконец я генерирую сбрасывающие события.

```

class StaticC_Generator...
private void generateResetEvents() throws IOException {
    output.println(" /* Переходы сбрасывающего события */");
    for (Event e : machine.getResetEvents())
        for (State s : machine.getStates())
            if (!s.hasTransition(e.getCode())) {
                output.printf(
                    " declare_transition(\"%s\", \"%s\", \"%s\");\n",
                    s.getName(),
                    e.getName(),
                    machine.getStart().getName());
            }
}

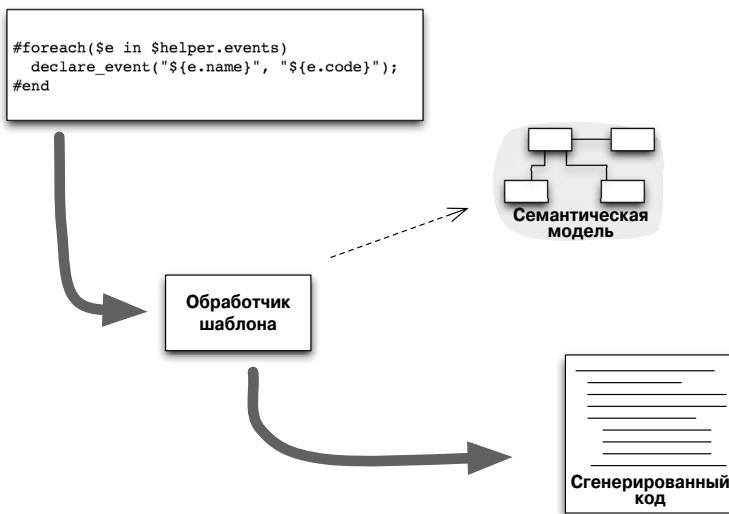
```


Глава 53

Шаблонная генерация

Templated Generation

Генерация вывода путем написания выходного файла вручную и размещения шаблонных меток-идентификаторов для генерации переменных частей



53.1. Как это работает

Базовая идея, лежащая в основе шаблонной генерации, заключается в написании выходного файла с включением меток-идентификаторов для всех варьирующихся частей. Затем с этим шаблонным файлом работает обработчик шаблонов, который и заполняет метки-идентификаторы для создания реального выходного файла.

Шаблонная генерация — очень старая технология, знакомая всем, кто, например, работал с возможностями электронной почты в текстовом редакторе. Она очень распространена в веб-разработке и применяется многими веб-сайтами с динамическим содержимым.

жимым. В этом случае шаблоном является весь документ, но шаблонная генерация может работать и с более мелкими контекстами. Старая добрая функция `printf` из стандартной библиотеки языка программирования C является примером применения шаблонной генерации для вывода одной строки. В контексте генерации кода я обычно использую шаблонную генерацию, когда весь документ является шаблоном, но пример `printf` напоминает нам, что шаблонная генерация и генерация с помощью преобразователя (*Transformer Generation* (523)) могут быть очень перемешаны. Текстовые макро-процессоры — еще один старый добрый инструмент разработки программного обеспечения — представляют собой разновидность шаблонной генерации.

При использовании шаблонной генерации имеются три основных компонента: обработчик шаблонов, шаблон и контекст. **Шаблон** представляет собой исходный текст выходного файла, в котором динамические части представлены маркерами-идентификаторами. Последние включают ссылки на контекст, который будет использоваться для заполнения динамических элементов в процессе генерации. **Контекст**, таким образом, выступает в качестве источника динамических данных — по сути, моделью данных для шаблонной генерации. Контекст может быть простой структурой данных или сложным программным контекстом; различные инструменты используют контексты разных видов. **Обработчик шаблонов** является инструментом, который связывает шаблон и контекст для генерации выхода. Управляющая программа будет выполнять шаблонную программу с определенным контекстом и шаблоном для получения выходного файла и может работать с одним и тем же шаблоном с разными контекстами, генерируя разные выходные файлы.

Наиболее общий вид обработчика шаблонов позволяет помещать в маркерах-идентификаторах произвольные выражения базового языка программирования. Это распространенный механизм, используемый такими инструментами, как JSP и ASP. Подобно любому виду внешнего кода (*Foreign Code* (315)) он должен использоваться с осторожностью, иначе структуру базового кода может испортить шаблон. Я настоятельно рекомендую ограничиться простыми вызовами функций внутри маркеров-идентификаторов, желательно с использованием встроенного помощника (*Embedment Helper* (537)).

Поскольку шаблонам файлов часто свойственно быть совершенно запутанными из-за большого количества вносимого кода, многие обработчики шаблонов не позволяют использовать в маркерах произвольный код. Такие инструменты предоставляют специализированные **языки шаблонов** для использования в маркерах вместо кода основного языка. Такие языки шаблонов, как правило, весьма ограничены, так что маркеры остаются достаточно простыми, а структура шаблонов сохраняет ясность. Простейший вид языка шаблонов рассматривает контекст как отображение и предоставляет выражения для поиска значений в этом отображении и их вставки в выходной файл. Для простых шаблонов такого механизма достаточно, однако очень часто нужны существенно большие возможности.

Как правило, в сложных ситуациях необходима возможность генерации выхода для элементов коллекции. Обычно для этого нужна некоторая итеративная структура, такая как цикл. Другая часто требуемая возможность — условная генерация, когда в зависимости от значения в контексте могут генерироваться различные результаты. Очень часто в исходных шаблонах встречается дублирование кода, что свидетельствует о необходимости некоторого механизма подпрограммы в самом языке шаблона.

Я не собираюсь разбираться в том, как различные шаблонные системы обрабатывают такие ситуации, хотя это весьма интересное и поучительное развлеченье. Мои же общие рекомендации минимальны, так как сила шаблонной генерации прямо пропорциональна простоте, с которой можно представить себе выходной файл, посмотрев на шаблон.

53.2. Когда это использовать

Мощь шаблонной генерации проявляется в том, что можно взглянуть на файл шаблона и сразу понять, как будет выглядеть сгенерированный результат. Это особенно полезно, когда имеется достаточно много статического содержимого, в то время как динамическое содержимое встречается относительно редко и оно достаточно простое.

Таким образом, первый признак того, что вы можете использовать шаблонную генерацию, — большое количество статического содержимого в созданном файле. Чем больше доля статического содержимого, тем выше вероятность того, что будет довольно просто применить шаблонную генерацию. Второе, что требуется рассмотреть, — сложность создаваемого динамического содержимого. Чем больше используется циклов, условных конструкций и прочих возможностей языка шаблонов, тем труднее представить, как будет выглядеть результат обработки файла шаблона. В этом случае имеет смысл подумать о применении генерации с помощью преобразователя (Transformer Generation (523)) вместо шаблонной генерации.

53.3. Генерация конечного автомата тайника с помощью вложенных условных конструкций (Velocity и Java, генерирующие C)

Генерации кода для вложенных условных конструкций в конечном автомате — хороший пример, когда статический вывод относительно велик, а динамическая часть достаточно проста, так что в этой ситуации показано применение шаблонной генерации. В данном примере я буду генерировать код, который рассматривается в примере для шаблона Model Ignorant Generation (557). Чтобы вы могли почувствовать, чего мы хотим добиться, взгляните на весь выходной файл.

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include "sm.h"
#include "commandProcessor.h"

#define EVENT_doorClosed "D1CL"
#define EVENT_drawerOpened "D2OP"
#define EVENT_lightOn "L1ON"
#define EVENT_doorOpened "D1OP"
#define EVENT_panelClosed "PNCL"
#define STATE_idle 1
#define STATE_active 0
#define STATE_waitingForDrawer 3
#define STATE_unlockedPanel 2
#define STATE_waitingForLight 4
#define COMMAND_lockDoor "D1LK"
#define COMMAND_lockPanel "PNLK"
#define COMMAND_unlockPanel "PNUL"
#define COMMAND_unlockDoor "D1UL"

static int current_state_id = -99;

void init_controller() {
    current_state_id = STATE_idle;
```

```
}

void hard_reset() {
    init_controller();
}

void handle_event_while_idle (char *code) {
    if (0 == strcmp(code, EVENT_doorClosed)) {
        current_state_id = STATE_active;
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}
void handle_event_while_active (char *code) {
    if (0 == strcmp(code, EVENT_lightOn)) {
        current_state_id = STATE_waitingForDrawer;
    }
    if (0 == strcmp(code, EVENT_drawerOpened)) {
        current_state_id = STATE_waitingForLight;
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}
void handle_event_while_waitingForDrawer (char *code) {
    if (0 == strcmp(code, EVENT_drawerOpened)) {
        current_state_id = STATE_unlockedPanel;
        send_command(COMMAND_unlockPanel);
        send_command(COMMAND_lockDoor);
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}
void handle_event_while_unlockedPanel (char *code) {
    if (0 == strcmp(code, EVENT_panelClosed)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}
void handle_event_while_waitingForLight (char *code) {
    if (0 == strcmp(code, EVENT_lightOn)) {
        current_state_id = STATE_unlockedPanel;
        send_command(COMMAND_unlockPanel);
        send_command(COMMAND_lockDoor);
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}
```

```

    }

void handle_event(char *code) {
    switch(current_state_id) {
        case STATE_idle: {
            handle_event_while_idle (code);
            return;
        }
        case STATE_active: {
            handle_event_while_active (code);
            return;
        }
        case STATE_waitingForDrawer: {
            handle_event_while_waitingForDrawer (code);
            return;
        }
        case STATE_unlockedPanel: {
            handle_event_while_unlockedPanel (code);
            return;
        }
        case STATE_waitingForLight: {
            handle_event_while_waitingForLight (code);
            return;
        }
        default: {
            printf("Невозможное состояние");
            exit(2);
        }
    }
}
}

```

Здесь я буду использовать обработчик шаблонов Apache Velocity, который представляет собой распространенный и простой в понимании и использовании обработчик шаблонов, доступный для Java и C#.

Я могу рассматривать общий файл как сегменты с динамическим содержимым, которые следует генерировать. Каждый сегмент управляется коллекцией элементов, которую я должен итеративно обойти для генерации кода этого сегмента.

Я начну с рассмотрения того, как можно генерировать определения событий, такие как `#define EVENT_doorClosed "D1CL"`. Если вы поймете, как это работает, разобраться во всем остальном не составит для вас труда.

Начну с кода шаблона.

```

Файл шаблона...
#define EVENT_doorClosed "D1CL"

```

К сожалению, один из источников путаницы заключается в том, что как препроцессор C (сам по себе являющийся разновидностью шаблонной генерации), так и Velocity используют символ "#" для указания шаблонной команды. `#foreach` является командой Velocity, в то время как `#define` — командой препроцессора C. Velocity игнорирует все нераспознанные команды, поэтому `#define` он рассматривает просто как текст.

`#foreach` представляет собой директиву Velocity для перебора элементов коллекции. Она поочередно получает каждый из элементов из `$helper.events` и выполняет тело директивы, в котором `$e` заменено этим элементом. Другими словами, это типичная конструкция в стиле “для каждого”.

`$helper.events` представляет собой ссылку на контекст шаблона. Я использую встроенный помощник (`Embedment Helper` (537)) и, таким образом, размещаю в контексте Velocity только сам помощник, в данном случае — экземпляр `SwitchHelper`. Помощник инициализируется конечным автоматом, а свойство `events` обеспечивает к нему доступ.

```
class SwitchHelper...
    private StateMachine machine;

    public SwitchHelper(StateMachine machine) {
        this.machine = machine;
    }
    public Collection<Event> getEvents() {
        return machine.getEvents();
    }
```

Каждое событие представляет собой объект семантической модели (`Semantic Model` (171)). В результате я могу использовать свойство `code` непосредственно. Однако создание константы для ссылки в коде требует немного большей работы, поэтому я размещаю некоторый код в помощнике.

```
class SwitchHelper...
    public String eventEnum(Event e) {
        return String.format("EVENT_%s", e.getName());
    }
```

Конечно, нет абсолютной необходимости использовать константу. Можно было бы использовать сам код события. Я генерирую константы, потому что предпочитаю, чтобы даже сгенерированный код был удобочитаемым.

Как это обычно бывает, команды используют точно такой же механизм, как и события, поэтому я оставлю этот код вашему воображению.

Для создания событий мне нужно отсортировать целочисленные константы.

```
Файл шаблона...
#foreach ($s in $helper.states)
#define $helper.stateEnum($s) $helper.stateId($s)
#end

class SwitchHelper...
    public Collection<State> getStates() {
        return machine.getStates();
    }
    public String stateEnum(State s) {
        return String.format("STATE_%s", s.getName());
    }
    public int stateId(State s) {
        List<State> orderedStates =
            new ArrayList<State>(getStates());
        Collections.sort(orderedStates);
        return orderedStates.indexOf(s);
    }
```

Некоторым читателям могут не понравиться моя генерация и сортировка списка состояний каждый раз, когда мне нужен идентификатор. Будьте уверены, что, если бы это вызывало проблемы с производительностью, я кэшировал бы отсортированный список, но таких проблем мой подход не вызывает.

Теперь, когда сгенерированы все объявления, я могу генерировать условные конструкции. Сначала идет внешняя конструкция `switch` для текущего состояния.

Файл шаблона...

```

void handle_event(char *code) {
    switch(current_state_id) {
        #foreach ($s in $helper.states)
            case $helper.stateEnum($s):
                handle_event_while_$s.name (code);
                return;
        }
    #end
    default:
        printf("Невозможное состояние");
        exit(2);
    }
}

```

Затем следует внутренняя конструкция `switch` для входящего события. Я разбил код на несколько отдельных функций.

Файл шаблона...

```

#foreach ($s in $helper.states)
void handle_event_while_$s.name (char *code) {
    #foreach ($t in $helper.getTransitions($s))
        if (0 == strcmp(code, $helper.eventEnum($t.trigger))) {
            current_state_id = $helper.stateEnum($t.target);
        #foreach ($a in $t.target.actions)
            send_command($helper.commandEnum($a));
        #end
    }
    #end
}
#end

```

Чтобы получить переходы для каждого состояния, мне нужны переходы, определенные в семантической модели, и переходы сбрасывающего события.

```

class SwitchHelper...
public Collection<Transition> getTransitions(State s) {
    Collection<Transition> result =
        new ArrayList<Transition>();
    result.addAll(s.getTransitions());
    result.addAll(getResetTransitions(s));
    return result;
}

private Collection<Transition>
getResetTransitions(State s) {
    Collection<Transition> result =
        new ArrayList<Transition>();
    for (Event e : machine.getResetEvents()) {
        if (!s.hasTransition(e.getCode()))
            result.add(new Transition(s, e, machine.getStart()));
    }
    return result;
}

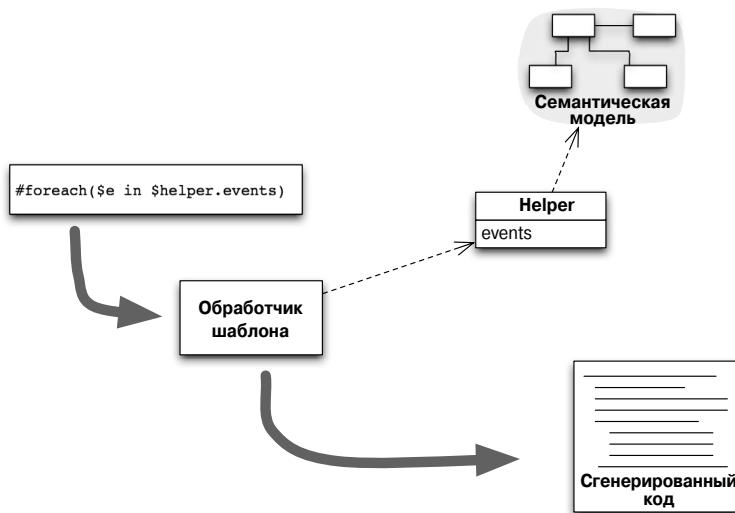
```


Глава 54

Встроенный помощник

Embedment Helper

Объект, минимизирующий код шаблонной системы, предоставляя все необходимые функции шаблонному механизму



Многие системы позволяют расширить возможности простого представления путем встраивания в это представление кода общего назначения, чтобы можно было выполнять действия, которые иначе невозможны. Примерами могут служить вставка кода в шаблоны веб-страниц, размещение кода действий в файлах грамматик или маркеры-идентификаторы с кодом в шаблоны генерации кода. Этот механизм внешнего кода (*Foreign Code (315)*) общего назначения существенно увеличивает возможности представления, в который он встраивается, при этом не усложняя само базовое представление. Однако распространенной проблемой при применении внешнего кода является то, что он ухудшает ясность и понимание представления, в которое встраивается.

Встроенный помощник перемещает весь сложный код во вспомогательный класс, оставляя в базовом представлении только простые вызовы методов. Это позволяет представлению оставаться доминирующим и сохранять его ясность.

54.1. Как это работает

Основная идея встроенного помощника сходна с применением рефакторинга. Создайте встроенный помощник, обеспечьте его видимость для базового представления, перенесите весь код из представления во встроенный помощник — и в представлении останутся только простые вызовы методов.

В этой технологии имеется один потенциально сложный технический аспект, который заключается в переносе объекта в область видимости при обработке представления. Большинство систем предоставляют для этого тот или иной механизм, который необходим и для вызова библиотек, но иногда он несколько запутан и хаотичен.

После того как вы сделаете встроенный помощник видимым, любой код, который представляет собой нечто большее, чем простой вызов метода, должен быть перемещен во встроенный помощник, так что в базовом представлении остаются только простые вызовы методов.

Это приводит к еще одному осложнению, которое не связано с технической стороной дела: как сделать понятным то, что делает код во встроенном помощнике? Ключ к этому, как и в случае любой абстракции, — в тщательном именовании методов, чтобы названия методов ясно отражали их назначение, не раскрывая при этом конкретную реализацию. Это столь же фундаментальный навык любого хорошего программиста, как и именование методов и функций в любом контексте.

Встроенный помощник часто используется вместе с шаблонной генерацией (*Templated Generation* (529)), и, когда вы сталкиваетесь с таким сочетанием, часто возникает вопрос “Должен ли встроенный помощник генерировать вывод?” Я часто сталкивался с безапелляционным утверждением: встроенные помощники не должны генерировать вывод ни при каких обстоятельствах. Я не согласен с такой категоричностью. Конечно, в случае генерации вывода во встроенном помощнике имеются проблемы — этот вывод не виден из шаблона. Так как весь смысл шаблонной генерации в том, чтобы вы могли по шаблону видеть, что будет получаться в результате, такая скрытая генерация, несомненно, представляет собой проблему.

Однако я думаю, что эта проблема должна быть соразмерена со сложностью сохранения вывода в шаблоне и наличием более сложных конструкций внешнего кода (*Foreign Code* (315)), которые могут понадобиться, чтобы избежать генерации в помощнике. Исключительный компромисс в каждом конкретном случае свой, и, хотя я и считаю, что желательно избегать генерации во встроенном помощнике, я не думаю, что это всегда лучше, чем любые другие альтернативы.

54.2. Когда это использовать

Я с большим подозрением отношусь к шаблонам, о которых кто-то говорит, что они должны использоваться всегда, но встроенный помощник — один из тех шаблонов, о которых я говорю точно так же, кроме разве что действительно тривиальных случаев. В свое время мне пришлось пересмотреть очень много исходных текстов с внешним кодом (*Foreign Code* (315)), и я могу с уверенностью говорить об огромной разнице в случае использования встроенного помощника и при его отсутствии. Без него очень трудно понять

базовое представление. Так, большое количество внешнего кода в действиях файла грамматики практически не позволяет изучить саму грамматику.

Сохранение ясности базового представления является главной причиной применения встроенного помощника, но есть и еще одно преимущество с точки зрения используемого инструментария. В особенности оно становится очевидным при использовании сложных интегрированных сред разработки. В этом случае внедренный код нельзя редактировать с помощью инструментов, которыми оснащена интегрированная среда разработки, но если перенести его во встроенный помощник, то все возможности среды станут доступными. Даже минимальные возможности простых текстовых редакторов наподобие подсветки обычно не работают должным образом со встроенным кодом.

Тем не менее есть ситуация, когда встроенный помощник применять не стоит: если вы используете классы, которые действуют как естественное хранилище для соответствующей информации. Примером может служить применение шаблонной генерации (*Templated Generation* (529)) с семантической моделью (*Semantic Model* (171)). В этом случае большая часть поведения, которую вы бы разместили во встроенном помощнике, может быть частью самой семантической модели, если, конечно, это не делает семантическую модель слишком сложной.

54.3. Состояния тайника (Java и ANTLR)

Возможно, самый простой способ объяснить, как работает встроенный помощник, — это показать, как он выглядит, когда вы его *не* используете. Для этого я возьму файл ANTLR-грамматики, почти такой же, какой был использован в примере для встраиваемой трансляции (*Embedded Translation* (305)). Я не буду приводить весь файл грамматики, ограничившись здесь несколькими правилами:

```
machine   : eventList resetEventList commandList state*;
eventList : 'events' event* 'end';
event     : name=ID code=ID
{
    events.put($name.getText(),
    new Event($name.getText(), $code.getText()));
};
state     : 'state' name=ID
{
    obtainState($name);
    if (null == machine)
        machine =
            new StateMachine(states.get($name.getText()));
}
actionList[$name]?
transition[$name]*
'end'
;
transition [Token sourceState]
    : trigger = ID '=>' target = ID
{
    states.get($sourceState.getText())
        .addTransition(events.get($trigger.getText()),
                      obtainState($target));
};
```

Помимо кода в действиях, мне нужно отдельно создать таблицы символов (*Symbol Table* (177)), а также определить некоторые функции общего назначения, которые по-

могут избежать дублирования кода, такие как `obtainState`. Я делаю это в разделе `members` файла грамматики.

При таком встроенным коде файлы грамматики могут содержать больше строк Java, чем самой грамматики DSL. А вот, для сравнения, как это выглядит с применением встроенного помощника.

```
machine   : eventList resetEventList commandList state*;
eventList : 'events' event* 'end';
event    : name=ID code=ID {helper.addEvent($name, $code);};
state    : 'state' name=ID {helper.addState($name);}
          actionList[$name]?
          transition[$name]*
          'end';
transition [Token sourceState]
           : trigger = ID '=>' target = ID
             {helper.addTransition($sourceState,
                                     $trigger, $target);};
```

Разница заключается в переносе кода во встроенный помощник. Для этого первым шагом является размещение объекта помощника в генерируемом синтаксическом анализаторе. ANTLR позволяет мне сделать это путем объявления поля в разделе `members`.

```
@members {
    StateMachineLoader helper;
//...
```

Это действие помещает поле в генерируемый класс синтаксического анализатора. Я настраиваю его видимость на уровне пакета, так что с ним можно работать с помощью другого класса. Его можно было бы сделать закрытым и обеспечить методы доступа, но я не думаю, что этим стоит заниматься в данном случае.

В общем потоке выполнения программы у меня есть класс загрузчика, который управляет синтаксическим анализом. Он хранит результирующий конечный автомат, а при создании я передаю ему объект читателя в качестве параметра.

```
class StateMachineLoader...
private Reader input;
private StateMachine machine;

public StateMachineLoader(Reader input) {
    this.input = input;
}
```

Метод `run` выполняет синтаксический анализ и заполняет поле `machine`.

```
class StateMachineLoader...
public StateMachine run() {
    try {
        StateMachineLexer lexer =
            new StateMachineLexer(new ANTLRReaderStream(input));
        StateMachineParser parser =
            new StateMachineParser(new CommonTokenStream(lexer));
        parser.helper = this;
        parser.machine();
        machine.addResetEvents(resetEvents.toArray(new Event[0]));
        return machine;
    } catch (IOException e) {
        throw new RuntimeException(e);
    } catch (RecognitionException e) {
        throw new RuntimeException(e);
    }
}
```

Синтаксический анализ ANTLR инициируется строкой `parser.machine`. Помощник настраивается в предыдущей строке. В данном случае в роли помощника выступает класс загрузчика. Загрузчик достаточно прост, так что лучшим решением представляется добавить в него вспомогательное поведение, а не создавать для этого отдельные классы.

Затем я добавляю в помощник методы для обработки различных вызовов. Я не буду приводить здесь их все. Вот лишь один из них — для добавления события.

```
class StateMachineLoader...
void addEvent(Token name, Token code) {
    events.put(name.getText(), new Event(name.getText(),
        code.getText()));
}
```

Чтобы свести объем кода в файле грамматики к минимуму, я передаю необходимую информацию в токене, откуда ее извлекает помощник.

При использовании генератора синтаксических анализаторов (*Parser Generator* (277)) меня часто беспокоит один вопрос: должен ли я использовать для встроенного помощника названия, ориентированные на события или на команды? В данном случае я использовал командно-ориентированные имена `addEvent` и `addState`. Имена, ориентированные на события, имели бы вид наподобие `eventRecognized` и `stateNameRecognized`. Аргументом в пользу имен, ориентированных на события, является то, что они не подразумевают каких-либо действий помощника, предоставляя ему решать, что именно делать. Это особенно удобно, если вы используете с одним и тем же синтаксическим анализатором разные помощники, которые выполняют разные действия в качестве реакции на анализ. Проблемой в случае имен, ориентированных на события, является то, что вы не можете сказать, что именно происходит, опираясь только на саму грамматику. В случае, когда я использую грамматику только для единственного вида деятельности, я предпочел бы быть в состоянии, читая грамматику, черпать из имен информацию о том, что происходит на каждом шаге.

В данном примере я использовал в качестве встроенного помощника отдельный объект. Другой подход заключается в использовании суперкласса. Настройка `superClass` в ANTLR позволяет мне установить любой класс в качестве суперкласса для генерируемого синтаксического анализатора. Затем я могу использовать в качестве суперкласса встроенный помощник и разместить в нем все необходимые данные и функции. Преимуществом этого подхода является то, что я могу написать просто `addEvent`, а не `helper.addEvent`.

54.4. Должен ли помощник генерировать HTML (Java и Velocity)

Распространенное правило, которое я часто слышу, — что встроенный помощник не должен генерировать никакого вывода. Я не считаю это полезным правилом, но чувствую, что неплохо бы привести конкретный пример, что к тому же может быть хорошим способом изучения компромиссов в этом вопросе. Пример на самом деле не связан с DSL, так как предполагает создание HTML, но одни и те же принципы спасают меня от мучительного поиска какого-то искусственного и надуманного примера.

Предположим, у нас есть коллекция объектов `person`, и мы хотим распечатать их имена в виде неупорядоченного списка. Каждый человек может иметь адрес электронной почты или URL. Если у него есть URL, то следует сделать его имя ссылкой, указывающей на URL; если есть адрес электронной почты, то следует сделать ссылку `mailto:`; если нет ни URL, ни электронной почты, никаких ссылок быть не должно. При использовании Velocity в качестве обработчика шаблонов приводят к следующему коду.

```
<ul>
#foreach($person in $book.people)
#if( $person.getUrl() )
<li><a href="$person.url">$person.fullName</a></li>
#elseif( $person.getEmail() )
<li><a href="mailto:$person.getEmail()">$person.fullName</a></li>
#else
<li>$person.fullName</li>
#end
#end
</ul>
```

Проблема в том, что теперь в файл шаблона встроена логика. Эта логика может мешать восприятию макета, т.е. делать то, с чем призван бороться встроенный помощник. Вот как выглядит шаблон при использовании встроенного помощника, генерирующего выходной текст.

Файл шаблона...

```
<ul>
#foreach($person in $book.people)
<li>$helper.render($person)</li>
#end
</ul>

class PageHelper...
public String render(Person person) {
    String href = null;
    if (null != person.getEmail())
        href = "mailto:" + person.getEmail().toString();
    if (null != person.getUrl())
        href = person.getUrl().toString();
    if (null != href)
        return String.format("<a href = \"%s\">%s</a>",
                           href, person.getFullName());
    else
        return person.getFullName();
}
```

Перемещая логику во встроенный помощник, я облегчаю восприятие шаблона ценой того, что часть получающегося HTML в шаблоне не отображается.

Но прежде чем поставить вас перед выбором “все или ничего”, я должен отметить, что часто в таких ситуациях можно найти золотую середину. В нашем случае во встроенный помощник можно перенести только часть логики, не включающую генерацию выходной информации.

Файл шаблона...

```
<ul>
#foreach($person in $book.people)
#if( $helper.hasLink($person) )
<li><a href = "$helper.getHref($person)">
                $person.fullName</a></li>
#else
<li>$person.fullName</li>
#end
#end
</ul>

class PageHelper...
public boolean hasLink(Person person) {
    return (null != person.getEmail()) ||
```

```
        (null != person.getUrl());
    }
public String getHref(Person person) {
    if (null != person.getUrl())
        return person.getUrl().toString();
    if (null != person.getEmail())
        return "mailto:" + person.getEmail().toString();
    throw new IllegalArgumentException("Ссылки нет");
}
```

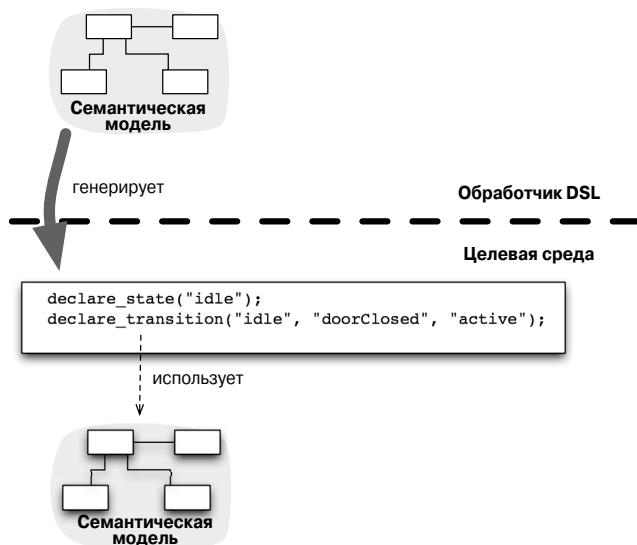
Лично я считаю, что перемещение некоторой генерации вывода во встроенный помощник является вполне разумным выбором. Чем сложнее логика и сам шаблон, тем больше преимуществ дает перемещение генерации вывода во встроенный помощник. Самое большое возражение против этого относится к случаю, когда с шаблоном и кодом работают разные люди. В такой ситуации внесение изменений требует координации их работы.

Глава 55

Генерация, осведомленная о модели

Model-Aware Generation

*Генерация кода с явной имитацией семантической модели DSL,
так что сгенерированный код разделен на обобщенный
и конкретный*



При генерации кода вы встраиваете в него семантику сценария DSL. С помощью генерации, осведомленной о модели, вы дублируете некоторую форму семантической модели (**Semantic Model (171)**) в сгенерированном коде, чтобы сохранить в нем разделение обобщенного и конкретного кода.

55.1. Как это работает

Наиболее важным аспектом генерации, осведомленной о модели, является то, что она сохраняет принцип разделения обобщенного и конкретного. Конкретная форма, которую модель принимает в сгенерированном коде, гораздо менее важна. Вот почему я предпочитаю говорить, что сгенерированный код содержит симулякр семантической модели (*Semantic Model* (171)).

Это симулякр модели по многим причинам. Как правило, код генерируется из-за ограничений целевой среды. Эти ограничения часто усложняют выражение семантической модели больше, чем вам хотелось бы. В результате требуется множество компромиссов, которые делают семантическую модель менее эффективной в смысле выражения намерений системы. Однако важно понимать, что пока удается разделять обобщенное и конкретное, все это не так уж и важно.

Так как симулякр представляет собой самостоятельную версию семантической модели, можно и должно строить и тестировать эту модель без использования генерации кода. Убедитесь, что модель имеет простой API для наполнения. Генерация кода будет позже создавать код конфигурации, который будет вызывать этот API. Вы сможете выполнять проверку симулября модели с использованием тестовых сценариев, которые также будут использовать этот API. Это позволяет создавать, тестировать и совершенствовать базовое поведение целевой среды с помощью генерации кода. Все это можно делать с помощью относительно простого тестового наполнения модели, которое должно быть проще для понимания и отладки.

55.2. Когда это использовать

Использование генерации, осведомленной о модели, имеет много преимуществ по сравнению с использованием генерации, игнорирующей модель (*Model Ignorant Generation* (557)). Симулябр модели без генерации проще создавать и тестировать, потому что при этом вы не должны повторно запускать и осмысливать генерацию кода. Поскольку сгенерированный код состоит из вызовов API симулябра, его намного проще генерировать, а значит, проще создавать и поддерживать генератор.

Основная причина не использовать генерацию, осведомленную о модели, — ограничения целевой среды. Обычно они сводятся к тому, что выразить даже симулябр модели оказывается слишком сложно либо у симулябра возникают проблемы производительности во время выполнения.

Во многих случаях DSL используется как интерфейс к существующей модели. Если вы генерируете код для работы с моделью, значит, вы используете генерацию, осведомленную о модели.

55.3. Конечный автомат тайника (C)

За примером генерации, осведомленной о модели, я вновь обращусь к конечному автомата тайника, с которого начинал эту книгу. Представим теперь ситуацию, когда мы работаем с системой безопасности из устройств с поддержкой Java, и тут приходит новая партия, которая программируется только на C. В результате нам нужно сгенерировать код на C на основе существующей семантической модели Java.

Здесь я не буду детально распространяться о самой генерации кода; если вас интересует эта тема, обратитесь к главе, посвященной генерации с помощью преобразователя (Transformer Generation (523)). Здесь я сосредоточусь на том, как может выглядеть окончательный код (как сгенерированный, так и рукописный) при использовании генерации, осведомленной о модели.

Имеется много способов реализации такой модели в С. По сути, я реализую ее как структуру данных с процедурами, которые обходят эту структуру с целью получения требующегося поведения. Каждый физический контроллер управляет только одним устройством, так что мы можем хранить структуру как статические данные. Кроме того, я избегаю выделения памяти из кучи и распределяю всю необходимую мне память с самого начала.

Я построил структуру данных как набор вложенных записей и массивов. Вершиной структуры является контроллер.

```
typedef struct {
    stateMachine *machine;
    int currentState;
} Controller;
```

Заметьте, что я представляю текущее состояние как целое значение. Как вы увидите, я использую ссылки на целочисленные значения в симулякре модели для представления всех разнообразных связей между различными частями модели.

Конечный автомат содержит массивы для состояний, событий и команд.

```
typedef struct {
    State states[NUM_STATES];
    Event events[NUM_EVENTS];
    Command commands[NUM_COMMANDS];
} stateMachine;
```

Размеры различных массивов устанавливаются с помощью макроопределений.

```
#define NUM_STATES          50
#define NUM_EVENTS           50
#define NUM_TRANSITIONS      30
#define NUM_COMMANDS         30
#define NUM_ACTIONS          10
#define COMMAND_HISTORY_SIZE 50
#define NAME_LENGTH           30
#define CODE_LENGTH            4
#define EMPTY                  -1
```

События и команды обладают именами и кодами.

```
typedef struct {
    char name[NAME_LENGTH + 1];
    char code[CODE_LENGTH + 1];
} Event;

typedef struct {
    char name[NAME_LENGTH + 1];
    char code[CODE_LENGTH + 1];
} Command;
```

Структура состояния хранит действия и переходы. Действия представляют собой целые значения, соответствующие командам, а переходы — пары целочисленных значений для запускающего события и целевого состояния.

```
typedef struct {
    int event;
```

```

    int target;
} Transition;

typedef struct {
    char name[NAME_LENGTH + 1];
    Transition transitions[NUM_TRANSITIONS];
    int actions[NUM_COMMANDS];
} State;

```

Многие программисты на С предпочли бы использовать арифметику указателей, а не индексы для навигации по структурам массивов, но я предпочитаю избегать арифметических операций над указателями, чтобы не запутывать тех читателей, которые не знакомы с C (не говоря уж о том, что мой C никогда не был достаточно хорош). Далее, я считаю, что генерированный код должен быть удобочитаемым, даже если он никогда не будет редактироваться, ведь даже в таком случае он, скорее всего, будет использоваться для отладки. Чтобы сделать его удобочитаемым, вы должны понимать свою целевую аудиторию, например тех, кто будет заниматься отладкой. Даже если вы как создатель генератора хорошо знакомы с арифметикой указателей и широко ее применяете, вам все равно следует использовать ее в генерированном коде с осторожностью, если те, кто будут читать генерированный код, не такие специалисты в этой области, как вы.

Чтобы закончить со структурами данных, я объявляю конечный автомат и контроллер как статические переменные, и это означает, что имеется только по одному их экземпляру.

```

static stateMachine machine;
static Controller controller;

```

Все эти данные определения сделаны в одном .c-файле. Таким образом, я могу инкапсулировать структуру данных за пакетом внешних объявлений функций. При этом конкретный код знает об этих функциях, и ничего не знает о структурах данных. Это тот случай, когда незнание — настоящеe блаженство.

Инициализируя конечный автомат, я помещаю нулевой байт в первые символы строк, делая их таким образом пустыми.

```

void init_machine() {
    int i;
    for (i = 0; i < NUM_STATES; i++) {
        machine.states[i].name[0] = '\0';
        int t;
        for (t = 0; t < NUM_TRANSITIONS; t++) {
            machine.states[i].transitions[t].event = EMPTY;
            machine.states[i].transitions[t].target = EMPTY;
        }
        int c;
        for (c = 0; c < NUM_ACTIONS; c++)
            machine.states[i].actions[c] = EMPTY;
    }
    for (i=0; i < NUM_EVENTS; i++) {
        machine.events[i].name[0] = '\0';
        machine.events[i].code[0] = '\0';
    }
    for (i=0; i < NUM_COMMANDS; i++) {
        machine.commands[i].name[0] = '\0';
        machine.commands[i].code[0] = '\0';
    }
}

```

Чтобы объявить новое событие, я ищу первое пустое событие и вставляю в него данные.

```
void declare_event(char *name, char *code) {
    assert_error(is_empty_event_slot(NUM_EVENTS - 1),
                 "Список событий полон");
    int i;
    for (i = 0; i < NUM_EVENTS; i++) {
        if (is_empty_event_slot(i)) {
            strncpy(machine.events[i].name, name, NAME_LENGTH);
            strncpy(machine.events[i].code, code, CODE_LENGTH);
            break;
        }
    }
    int is_empty_event_slot(int index) {
        return ('\0' == machine.events[index].name[0]);
    }
}
```

Конструкция `assert_error` представляет собой макрос, который проверяет переданное ему условие и, если оно ложно, вызывает соответствующую функцию для сообщения об ошибке.

```
#define assert_error(test, message) \
    do { if (!(test)) sm_error(#message); } while (0)
```

Обратите внимание на то, что я завернул в макрос блок `do-while`. Это выглядит странно, но предотвращает неприятности при использовании данного макроса в конструкции `if`.

Команды объявлены точно таким же образом, так что соответствующий код я опущу.

Состояния объявлены через ряд функций. Первая просто объявляет имя состояния.

```
void declare_state(char *name) {
    assert(is_empty_state_slot(NUM_STATES - 1));
    int i;
    for (i = 0; i < NUM_STATES; i++) {
        if (is_empty_state_slot(i)) {
            strncpy(machine.states[i].name, name, NAME_LENGTH);
            break;
        }
    }
    int is_empty_state_slot(int index) {
        return ('\0' == machine.states[index].name[0]);
    }
}
```

Объявление действий и переходов немного сложнее, так как требуется найти идентификатор действия по его имени. Вот как это выглядит для действий.

```
void declare_action(char *stateName, char *commandName) {
    int state = stateID(stateName);
    assert_error(state >= 0, "Неопознанное состояние");
    int command = commandID(commandName);
    assert_error(command >= 0, "Неопознанная команда");
    assert_error(EMPTY ==
                machine.states[state].actions[NUM_ACTIONS-1],
                "Слишком много действий у состояния");
    int i;
    for (i = 0; i < NUM_ACTIONS; i++) {
        if (EMPTY == machine.states[state].actions[i]) {
```

550 Часть VI. Генерация кода

```
        machine.states[state].actions[i] = command;
        break;
    }
}

int stateID(char *stateName) {
    int i;
    for (i = 0; i < NUM_STATES; i++) {
        if (is_empty_state_slot(i)) return EMPTY;
        if (0 == strcmp(stateName, machine.states[i].name))
            return i;
    }
    return EMPTY;
}

int commandID(char *name) {
    int i;
    for (i = 0; i < NUM_COMMANDS; i++) {
        if (is_empty_command_slot(i)) return EMPTY;
        if (0 == strcmp(name, machine.commands[i].name))
            return i;
    }
    return EMPTY;
}
```

Обработка переходов выполняется аналогично.

```
void declare_transition (char *sourceState, char *eventName,
                        char *targetState)
{
    int source = stateID(sourceState);
    assert_error(source >= 0,
                 "Неопознанное исходное состояние");
    int target = stateID(targetState);
    assert_error(target >= 0,
                 "Неопознанное целевое состояние");
    int event = eventID_named(eventName);
    assert_error(event >= 0, "unrecognized event");
    int i;
    for (i = 0; i < NUM_TRANSITIONS; i++) {
        if (EMPTY ==
            machine.states[source].transitions[i].event) {
            machine.states[source].transitions[i].event = event;
            machine.states[source].transitions[i].target = target;
            break;
        }
    }
    int eventID_named(char *name) {
        int i;
        for (i = 0; i < NUM_EVENTS; i++) {
            if (is_empty_event_slot(i)) break;
            if (0 == strcmp(name, machine.events[i].name))
                return i;
        }
        return EMPTY;
    }
}
```

Теперь я могу использовать эти объявления для определения полного конечного автомата, в данном случае — знакомого конечного автомата для мисс Грант.

```

void build_machine() {
    declare_event("doorClosed", "D1CL");
    declare_event("drawerOpened", "D2OP");
    declare_event("lightOn", "L1ON");
    declare_event("doorOpened", "D1OP");
    declare_event("panelClosed", "PNCL");

    declare_command("lockDoor", "D1LK");
    declare_command("lockPanel", "PNLK");
    declare_command("unlockPanel", "PNUL");
    declare_command("unlockDoor", "D1UL");

    declare_state("idle");
    declare_state("active");
    declare_state("waitingForDrawer");
    declare_state("unlockedPanel");
    declare_state("waitingForLight");

    /* Тело состояния idle */
    declare_action("idle", "unlockDoor");
    declare_action("idle", "lockPanel");
    declare_transition("idle", "doorClosed", "active");

    /* Тело состояния active */
    declare_transition("active", "lightOn",
                      "waitingForDrawer");
    declare_transition("active", "drawerOpened",
                      "waitingForLight");

    /* Тело состояния waitingForDrawer */
    declare_transition("waitingForDrawer", "drawerOpened",
                      "unlockedPanel");

    /* Тело состояния unlockedPanel */
    declare_action("unlockedPanel", "unlockPanel");
    declare_action("unlockedPanel", "lockDoor");
    declare_transition("unlockedPanel", "panelClosed",
                      "idle");

    /* Тело состояния waitingForLight */
    declare_transition("waitingForLight", "lightOn",
                      "unlockedPanel");

    /* Переходы сбрасывающего состояния */
    declare_transition("idle", "doorOpened", "idle");
    declare_transition("active", "doorOpened", "idle");
    declare_transition("waitingForDrawer", "doorOpened",
                      "idle");
    declare_transition("unlockedPanel", "doorOpened",
                      "idle");
    declare_transition("waitingForLight", "doorOpened",
                      "idle");
}
}

```

Этот код наполнения порожден генератором кода (см. раздел “Контроллер тайника (Java генерирует С)” на с. 525).

Теперь я должен показать код, который выполняет работу конечного автомата. В данном случае это функция, которая вызывается для обработки события с указанным кодом.

```
void handle_event(char *code) {
    int event = eventID_with_code(code);
```

```

// Неизвестное событие игнорируется
if (EMPTY == event) return;

int t = get_transition_target(controller.currentState,
                             event);
// Из этого состояния нет переходов - игнорируем
if (EMPTY == t) return;

controller.currentState = t;
int i;
for (i = 0; i < NUM_ACTIONS; i++) {
    int action =
        machine.states[controller.currentState].actions[i];
    if (EMPTY == action) break;
    send_command(machine.commands[action].code);
}
} int eventID_with_code(char *code) {
int i;
for (i = 0; i < NUM_EVENTS; i++) {
    if (is_empty_event_slot(i)) break;
    if (0 == strcmp(code, machine.events[i].code))
        return i;
}
return EMPTY;
}
int get_transition_target(int state, int event) {
int i;
for (i = 0; i < NUM_TRANSITIONS; i++) {
    if (EMPTY == machine.states[state].transitions[i].event)
        return EMPTY;
    if (event ==
        machine.states[state].transitions[i].event) {
        return machine.states[state].transitions[i].target;
    }
}
return EMPTY;
}
}

```

Так выглядит рабочая модель конечного автомата. Есть несколько моментов, которые хотелось бы отметить. Во-первых структура данных несколько примитивна и предполагает обход массива в поисках различных кодов и названий. При определении конечного автомата это, пожалуй, не является проблемой, но при работе конечного автомата было бы лучше заменить линейный поиск хеш-функцией. Поскольку конечный автомат хорошо инкапсулирован, это достаточно легко сделать, и я оставляю реализацию этой идеи в качестве упражнения для читателя. Изменение таких деталей реализации модели не влияет на интерфейс функций конфигурации, которые определяют новые конечные автоматы. Это важный результат инкапсуляции.

Модель не включает понятие сбрасывающего события. Различные сбрасывающие события, которые определены через сценарии DSL и семантическую модель Java, в конечном автоматах на С превратились просто в дополнительные переходы конечного автомата. Это делает его проще, и является примером типичного компромисса, когда я предпочитаю простоту работы четкому указанию намерений. В истинной семантической модели (*Semantic Model (171)*) я предпочитаю точно указывать намерения, но в генерированной для целевой среды модели я ценю ясное указание намерения не так высоко.

Я мог бы пойти дальше в упрощении работы конечного автомата, удаляя все имена событий, команд и состояний. Эти имена используются при настройке автомата и не используются во время выполнения. Так что я мог бы использовать временные таблицы поиска, которые стали бы не нужны и могли бы быть уничтожены после полного определения конечного автомата. Действительно, функции объявлений могли бы использовать просто целые числа, что-то вроде `declare_action(1, 2);`. Хотя это совершенно неудобочитаемо, можно аргументировать применение такого способа работы тем, что в любом случае этот код просто генерируется. Впрочем, я предпочитаю сохранять имена, так как считаю, что даже сгенерированный код должен быть удобочитаемым (и что еще более важно, это позволяет получить больше полезной диагностики, когда дела идут плохо). Но если бы с памятью в целевой среде было тяжело, я бы запросто этим пожертвовал.

55.4. Динамическая загрузка конечного автомата (C)

Генерация кода на C в приведенном выше примере означает, что для того, чтобы создать новый конечный автомат, нужно выполнить компиляцию получившегося кода. Использование генерации, осведомленной о модели, позволяет также создать конечный автомат во время работы, управляя генерацией кода с помощью другого файла.

В этом случае я могу выразить поведение конкретного конечного автомата с помощью текстового файла наподобие следующего.

```
config_machine.txt...
event doorClosed D1CL
event drawerOpened D2OP
event lightOn L1ON
event doorOpened D1OP
event panelClosed PNCL
command lockDoor D1LK
command lockPanel PNLK
command unlockPanel PNUL
command unlockDoor D1UL
state idle
state active
state waitingForDrawer
state unlockedPanel
state waitingForLight
transition idle doorClosed active
action idle unlockDoor
action idle lockPanel
transition active lightOn waitingForDrawer
transition active drawerOpened waitingForLight
transition waitingForDrawer drawerOpened unlockedPanel
transition unlockedPanel panelClosed idle
action unlockedPanel unlockPanel
action unlockedPanel lockDoor
transition waitingForLight lightOn unlockedPanel
transition idle doorOpened idle
transition active doorOpened idle
transition waitingForDrawer doorOpened idle
transition unlockedPanel doorOpened idle
transition waitingForLight doorOpened idle
```

Этот файл можно сгенерировать из семантической модели (Semantic Model (171)) Java.

```
class StateMachine...
public String generateConfig() {
```

554 Часть VI. Генерация кода

```
StringBuffer result = new StringBuffer();
for(Event e : getEvents()) e.generateConfig(result);
for(Command c : getCommands()) c.generateConfig(result);
for(State s : getStates())
    s.generateNameConfig(result);
for(State s : getStates())
    s.generateDetailConfig(result);
generateConfigForResetEvents(result);
return result.toString();
}

class Event...
public void generateConfig(StringBuffer result) {
    result.append(String.format("event %s %s\n",
        getName(),
        getCode()));
}

class Command...
public void generateConfig(StringBuffer result) {
    result.append(String.format("command %s %s\n",
        getName(),
        getCode()));
}

class State...
public void generateNameConfig(StringBuffer result) {
    result.append(String.format("state %s\n",
        getName()));
}
public void generateDetailConfig(StringBuffer result) {
    for(Transition t : getTransitions())
        t.generateConfig(result);
    for(Command c : getActions())
        result.append(String.format("action %s %s\n",
            getName(),
            c.getName()));
}
```

Для запуска конечного автомата можно легко интерпретировать файл config_machine с применением трансляции, управляемой разделителями (Delimiter-Directed Translation (213)) с использованием простых функций обработки строк из стандартной библиотеки C.

Общая функция для создания конечного автомата открывает файл и поочередно интерпретирует каждую строку.

```
void build_machine() {
    FILE * input = fopen("machine.txt", "r");
    char buffer[BUFFER_SIZE];
    while (NULL != fgets(buffer, BUFFER_SIZE, input)) {
        interpret(buffer);
    }
}
```

Стандартная функция C strtok позволяет разбить строку на токены, разделенные пробельными символами. Я могу извлечь первый токен, а затем вызвать функцию для интерпретации строки данного вида.

```
#define DELIMITERS " \t\n"

void interpret(char * line) {
    char * keyword;
    keyword = strtok(line, DELIMITERS);
    if (NULL == keyword) return; // Пустые строки
    if ('\#' == keyword[0]) return; // Комментарии
```

```
if (0 == strcmp("event", keyword))
    return interpret_event();
if (0 == strcmp("command", keyword))
    return interpret_command();
if (0 == strcmp("state", keyword))
    return interpret_state();
if (0 == strcmp("transition", keyword))
    return interpret_transition();
if (0 == strcmp("action", keyword))
    return interpret_action();
sm_error("Unknown keyword");
}
```

Каждая конкретная функция извлекает необходимые токены и вызывает соответствующие статические функции `declare_*`, которые я определил в предыдущем примере. Я приведу функцию интерпретации событий; остальные функции практически такие же, как и эта.

```
void interpret_event() {
    char *name = strtok(NULL, DELIMITERS);
    char *code = strtok(NULL, DELIMITERS);
    declare_event(name, code);
}
```

(Повторные вызовы `strtok` с `NULL` в качестве первого аргумента приводят к дальнейшему извлечению токенов из той же строки, которая использовалась в предыдущем вызове `strtok`.)

Я не рассматриваю этот текстовый формат как DSL, поскольку я разработал его для простой интерпретации, а не для чтения людьми. Удобочитаемость очень полезна, так как существенно облегчает отладку. Тем не менее в данном случае удобочитаемость человеком отошла на второе место, уступив первенство простоте интерпретации.

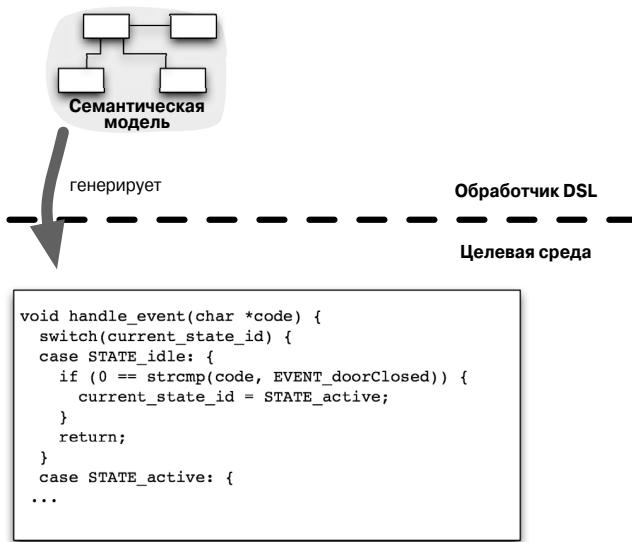
Цель этого примера — проиллюстрировать, что генерация кода для статического целевого языка не означает, что вы не можете использовать интерпретацию времени выполнения. С помощью генерации, осведомленной о модели, я могу скомпилировать обобщенную модель конечного автомата вместе с очень простым интерпретатором. После этого мой генератор кода будет генерировать текстовый файл, который затем будет интерпретироваться. Это позволяет использовать для контроллера язык программирования C, при этом перекомпиляция для внесения изменений в конечный автомат не требуется. Генерируя файл таким образом, чтобы максимально облегчить интерпретацию в доступной среде, можно свести расходы на интерпретатор к минимуму. Я мог бы, конечно, пойти дальше и написать на C полный обработчик DSL, но это привело бы к повышению требований к обработке и существенному увеличению количества программирования на C. В зависимости от конкретной ситуации такой подход может оказаться вполне приемлемым решением, и нам больше не понадобится генерация, осведомленная о модели.

Глава 56

Генерация, игнорирующая модель

Model Ignorant Generation

Вся логика жестко прошивается в сгенерированном коде, так что явное представление семантической модели отсутствует



56.1. Как это работает

Одним из преимуществ генерации кода является то, что она позволяет создавать код, слишком часто повторяющийся для того, чтобы писать его вручную. Это позволяет создавать реализации, которые при работе вручную невозможно было бы написать из-за дублирования кода. В частности, генерация кода позволяет получать поведение, которое обычно представлено структурами данных, и кодировать его в потоке управления.

Для использования генерации, игнорирующей модель, можно начать с написания реализации определенного сценария DSL в целевой среде. Я предпочитаю начинать с очень простого, минимального сценария. Такая реализация должна быть очень четкой и ясной, но может свободно смешивать обобщенный и конкретный код, а кроме того, можно не беспокоиться о повторах в определенных элементах, так как в конечном итоге они будут генерироваться. Это означает, что я не должен задумываться о сложных структурах данных, предпочитая в основном процедурный код и простые структуры.

56.2. Когда это использовать

Целевые среды часто включают языки с ограниченными возможностями в плане структурирования программ и построения хороших моделей. В таких ситуациях не представляется возможным использовать генерацию, осведомленную о модели (*Model-Aware Generation* (545)), поэтому единственным вариантом оказывается генерация, игнорирующая модель. Вторая основная причина использования генерации, игнорирующей модель, заключается в том, что при использовании генерации, осведомленной о модели, получается реализация, требующая слишком большого количества ресурсов. Кодирование логики в потоке управления может снизить требования к памяти или повысить производительность. Если это достаточно критичные моменты, то генерация, игнорирующая модель, позволит вам достичь желаемого.

В целом, однако, я предпочитаю по возможности использовать генерацию, осведомленную о модели. Как правило, при ее применении легче генерировать код, так что программу генерации проще понять и модифицировать. Я уже говорил, что генерация, игнорирующая модель, зачастую дает код, который легче понимать. Обратный эффект заключается в том, что при всей простоте генерируемого кода написать генерирующий код может оказаться намного сложнее.

56.3. Конечный автомат тайника как вложенные условные конструкции (C)

Я опять обращаюсь к конечному автомату тайника, который использовался во введении. Одна из классических реализаций конечного автомата использует вложенные условные конструкции, которые позволяют вычислить очередной шаг с помощью условных выражений, основываясь на текущем состоянии и полученном событии. В этом примере я покажу, как выглядит реализация контроллера мисс Грант при применении такой технологии. Чтобы увидеть, как можно сгенерировать этот код, обратитесь к примеру в описании шаблона *Templated Generation* (529).

Имеется два условия, которые необходимо обработать: входящее событие и текущее состояние. Я начну с обработки текущего состояния.

```
#define STATE_idle 1
#define STATE_active 0
#define STATE_waitingForDrawer 3
#define STATE_unlockedPanel 2
#define STATE_waitingForLight 4

void handle_event(char *code) {
    switch(current_state_id) {
        case STATE_idle: {
            handle_event_while_idle (code);
            return;
        }
    }
}
```

```

case STATE_active: {
    handle_event_while_active (code);
    return;
}
case STATE_waitingForDrawer: {
    handle_event_while_waitingForDrawer (code);
    return;
}
case STATE_unlockedPanel: {
    handle_event_while_unlockedPanel (code);
    return;
}
case STATE_waitingForLight: {
    handle_event_while_waitingForLight (code);
    return;
}
default: {
    printf("in impossible state");
    exit(2);
}

```

При тестировании состояния используется статическая переменная, хранящая текущее состояние.

```

#define ERROR_STATE -99
static int current_state_id = ERROR_STATE;
void init_controller() {
    current_state_id = STATE_idle;
}

```

Каждая вспомогательная функция выполняет дальнейшую обработку полученного события. Вот как выглядит обработка для состояния active.

```

void handle_event_while_active (char *code) {
    if (0 == strcmp(code, EVENT_lightOn)) {
        current_state_id = STATE_waitingForDrawer;
    }
    if (0 == strcmp(code, EVENT_drawerOpened)) {
        current_state_id = STATE_waitingForLight;
    }
    if (0 == strcmp(code, EVENT_doorOpened)) {
        current_state_id = STATE_idle;
        send_command(COMMAND_unlockDoor);
        send_command(COMMAND_lockPanel);
    }
}

```

Прочие вспомогательные функции очень похожи, поэтому я не буду здесь их приводить.

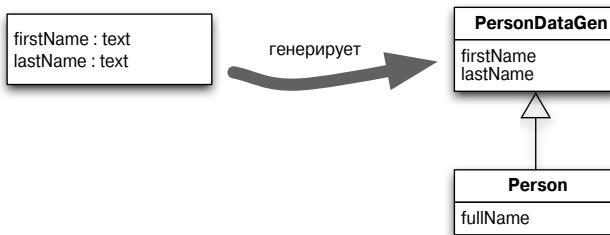
Для написания вручную для разных конечных автоматов этот код содержит слишком много повторяющихся фрагментов. Однако, будучи сгенерированным, он оказывается весьма простым для понимания.

Глава 57

Отделение генерируемого кода с помощью наследования

Generation Gap⁹

Отделение генерируемого кода от не генерируемого с помощью наследования



Одной из трудностей при генерации кода является то, что сгенерированный и рукописный коды должны рассматриваться и обрабатываться по-разному. Сгенерированный код никогда не должен редактироваться вручную, в противном случае вы не сможете безопасно генерировать его заново.

Описываемый в этой главе метод позволяет разделить сгенерированную и написанную вручную части кода, поместив их в разные классы, связанные отношением наследования.

Эта модель впервые была описана покойным Джоном Влиссидесом (John Vlissides). В его формулировке рукописный класс являлся подклассом сгенерированного класса. Мое описание немного иное; мне очень жаль, что я не могу обсудить с Джоном свой подход к проблеме...

⁹ Непереводимая игра слов, основанная на том, что слово “generation” имеет два значения: “генерация” и “поколение” (“generation gap” в переводе с английского означает “конфликт поколений”). В данной главе речь идет как о генерации кода, так и о поколениях классов, связанных наследованием. — Примеч. пер.

57.1. Как это работает

Базовая форма для отделения генерируемого кода с помощью наследования включает генерацию суперкласса, который Влиссидес называет основным (*core class*), и подкласс, написанный вручную. Таким образом, всегда можно переопределить любой аспект сгенерированного кода в подклассе. Рукописный код может легко вызывать любые сгенерированные методы, а сгенерированный — написанные вручную компоненты с помощью механизма абстрактных методов (наличие обязательной реализации которых компилятор может проверить в подклассе) или перехвата методов, которые перекрываются в случае необходимости.

Обращаясь к этим классам извне, вы всегда работаете с рукописным конкретным классом. Сгенерированный класс фактически игнорируется остальным кодом.

Распространенной вариацией этого метода является добавление третьего рукописного класса, который представляет собой суперкласс для генерируемого класса. Это делается для переноса из сгенерированного класса всей логики, которая не зависит от генерации кода. Размещение неизмененного кода в суперклассе вместо его генерации упрощает его отслеживание, в частности с помощью интегрированных сред разработки. В целом мое предложение по генерации кода можно сформулировать просто — генерировать как можно меньшее количество кода. Это связано с тем, что редактировать любой сгенерированный код сложнее, чем написанный вручную. При внесении изменений в сгенерированный код необходимо перезапустить систему генерации кода. Возможности рефакторинга современных интегрированных сред разработки со сгенерированным кодом работать должным образом не будут.

Таким образом, потенциально в конечном итоге вы получите три класса со следующей структурой наследования.

- **Рукописный базовый класс** содержит логику, которая не зависит от параметров генерации кода.
- **Сгенерированный класс** содержит логику, которая может быть создана автоматически на основании параметров генерации.
- **Рукописный конкретный класс** содержит логику, которая не может быть сгенерирована и опирается на сгенерированные возможности. Этот класс является единственным классом, к которому может обращаться другой код.

Все три класса нужны не всегда. Если у вас нет неизменной логики, вам не нужен рукописный базовый класс. Аналогично, если вы никогда не переопределяете сгенерированный код, вы можете обойтись без рукописного конкретного класса. Таким образом, еще одной вариацией рассматриваемого шаблона является рукописный суперкласс и сгенерированный подкласс.

Часто встречаются и более сложные структуры сгенерированных и рукописных классов, связанных как наследованием, так и использованием друг друга путем вызовов. Взаимодействие генерации кода и написания вручную ведет к более сложной структуре классов; это цена, которую вы платите за удобство генерации кода.

Одной из сложностей применения данного шаблона является вопрос о том, что делать, если конкретные рукописные классы есть, но не всегда. В этом случае необходимо решить, что же делать с теми классами, для которых нет конкретных рукописных классов. Вы можете сделать сгенерированный класс именованным классом, используемым вызывающим кодом, но это приведет к путанице в именовании и применении. Поэтому лично я предпочитаю всегда создавать конкретный класс, пусть даже пустой, если ничего переопределять не требуется.

Остается еще вопрос: должен ли программист сам создавать эти пустые классы вручную или их должна создавать система генерации кода? Если их немного, а меняются они редко, то лучше оставить их создание программисту. Однако если таких классов много и они часто меняются, то лучше так настроить систему генерации кода, чтобы она проверяла наличие конкретного класса и, если такого нет, генерировала соответствующий пустой класс.

57.2. Когда это использовать

Описанный метод очень эффективен и позволяет создавать логически один класс, разделенный на отдельные файлы, чтобы хранить сгенерированный код отдельно. Чтобы им воспользоваться, нужен язык, поддерживающий наследование. Использование наследования означает, что любые члены, которые могут быть перекрыты, должны иметь соответствующие права доступа и быть видимыми для подклассов, т.е. не быть закрытыми членами Java или C#.

Если ваш язык позволяет разместить код одного класса в нескольких файлах, как частичные классы C# или открытые классы Ruby, то это может быть вполне работоспособной и разумной альтернативой отделению генерируемого кода с помощью наследования. Преимущество файлов частичных классов в том, что можно разделить генерируемый и рукописный коды без наследования — все будет находиться в одном классе. Недостатком частичных классов C# является то, что хотя этот механизм пригоден для добавления функциональных возможностей в генерируемые классы, он не позволяет переопределять функциональные возможности. Открытые классы Ruby могут справиться с этим, выполняя рукописный код после сгенерированного, что позволяет заменить сгенерированный метод написанным вручную.

Распространенной ранней альтернативой данного метода была генерация кода в выделенную область файла между комментариями специального вида наподобие `code gen start` и `code gen end`. Здесь проблема в том, что такое выделение части файла запутывает программиста, приводит к частой модификации сгенерированного кода и мешает системе управления версиями. Хранение сгенерированного кода в отдельных файлах, если вы сможете этого добиться, почти всегда — наилучшая идея.

Хотя отделение генерируемого кода с помощью наследования является неплохим подходом, это не единственный способ хранить сгенерированный код отдельно от рукописного. Часто достаточно разместить эти коды в разных классах с вызовами между ними. Сотрудничающие классы — механизм, простой для использования и понимания, поэтому в целом я предпочитаю именно его. Я прибегаю к отделению генерируемого кода с помощью наследования только тогда, когда такое взаимодействие становится более сложным, например когда в сгенерированном классе есть поведение по умолчанию, которое я хочу переопределить для частных случаев.

57.3. Генерация классов из схемы данных (Java и немного Ruby)

Распространенной ситуацией является генерация определения данных для классов, основанных на некоторой схеме данных. Если вы пишете *Row Data Gateway* [10] для доступа к базе данных, можете сгенерировать большую часть этого класса из самой схемы базы данных.

Мне лень возиться с SQL или XML, поэтому я выберу что-нибудь попроще. Предположим, что я читаю простые CSV-файлы (comma separated values — значения, разделен-

ные запятыми), причем настолько простые, что в них нет ни кавычек, ни комментариев, ни служебных последовательностей символов. Для каждого файла у меня есть простой файл схемы для определения имен файлов и типов данных для каждого поля. Ниже приведена схема для людей.

```
firstName : text
lastName : text
empID : int
```

А вот — данные о них.

```
martin, fowler, 222
neal, ford, 718
rebecca, parsons, 20
```

На основании приведенных данных я хочу генерировать объект *DTO* на языке Java [10] с правильным типом для каждого поля схемы, методы доступа для каждого поля, а также возможность выполнения некоторых проверок.

При генерации кода на компилируемом языке типа Java процесс сборки может оказаться достаточно длинным. Если я пишу сам генератор кода на Java, я должен скомпилировать генератор кода отдельно от остального кода. Это “загрязняет” грязный процесс сборки, особенно при работе с интегрированной средой разработки. Альтернативный подход состоит в использовании для генерации кода языка сценариев. Тогда для генерации кода требуется только выполнить этот сценарий. Это упрощает процесс сборки за счет введения другого языка. На мой взгляд, всегда нужно иметь под рукой какой-нибудь язык сценариев, так как всегда найдется задача, которую стоит автоматизировать с помощью сценариев. В данном случае я использую Ruby, так как это давно выбранный мною язык сценариев. Я буду применять шаблонную генерацию (*Templated Generation* (529)) со стандартной библиотекой ERB, которая представляет собой встроенную систему обработки шаблонов Ruby.

Семантическая модель (*Semantic Model* (171)) схемы очень проста. Схема представляет собой набор полей с именем и типом данных для каждого поля.

```
class Schema...
  attr_reader :name
  def initialize name
    @name = name
    @fields = []
  end

  class Field...
    attr_accessor :name, :type
    def initialize name, type
      @name = name
      @type = type
    end
  end
end
```

Синтаксический анализ файла схемы очень прост — я просточитываю каждую строку, разбиваю ее на токены с двух сторон от двоеточия и создаю объекты полей. Так как логика синтаксического анализа очень проста, я не отделяю код синтаксического анализа от объектов семантической модели.

```
class Schema...
  def load input
    input.readlines.each {|line| load_line line }
  end
```

```

def load_line line
  return if blank?(line)
  tokens = line.split ':'
  tokens.map! { |t| t.strip }
  @fields << Field.new(tokens[0], tokens[1])
end

def blank? line
  return line =~ /^ \s* $/
end

```

Как только я наполняю семантическую модель, я могу использовать ее для создания классов данных. Я начну с определения полей и методов для доступа к ним. В результате я хочу генерировать код наподобие следующего.

```

public class PersonDataGen extends AbstractData {

  private String firstName;
  public String getFirstName () {
    return firstName ;
  }
  public void setFirstName (String arg ) {
    firstName = arg;
  }
  protected void checkFirstName(Notification note) {};

  private String lastName;
  public String getLastname () {
    return lastName ;
  }
  public void setLastName (String arg ) {
    lastName = arg;
  }
  protected void checkLastName(Notification note) {};

  private int empID;
  public int getEmpID () {
    return empID ;
  }
  public void setEmpID (int arg ) {
    empID = arg;
  }
  protected void checkEmpID(Notification note) {};
}

```

Я создаю генерируемый класс как подкласс неизмененного написанного вручную класса. Я не использую его для базового определения полей и вскоре продемонстрирую вам его применение.

Для этого я создаю шаблон.

```

public class <%=name%>DataGen extends AbstractData {
  <% @fields.each do |f| %>
  private <%= f.java_type %> <%= f.name %>;
  public <%=f.java_type%> <%=f.getter_name%> () {
    return <%=f.name%> ;
  }
  public void <%= f.setter_name %> (<%= f.java_type %> arg){
    <%= f.name %> = arg;
  }
  protected void <%= f.checker_name %>(Notification note){};
<% end %>

```

Шаблон обращается к некоторым методам семантической модели за помощью в генерации кода.

```
class Field...
  def java_type
    case @type
      when "text" : "String"
      when "int" : "int"
      else raise "Неизвестный тип поля"
    end
  end

  def method_suffix
    @name[0..0].capitalize + @name[1..-1]
  end

  def getter_name
    "get#{method_suffix}"
  end

  def setter_name
    "set#{method_suffix}"
  end

  def checker_name
    "check#{method_suffix}"
  end
```

Генерация таких полей позволяет переопределять методы получения и установки или добавлять в класс новые методы. В этом случае я могу, например, возвращать имена с большой буквы и добавить возможность формировать полное имя.

```
public class PersonData extends PersonDataGen {
  public String getLastName() {
    return capitalize(super.getLastName());
  }
  public String getFirstName() {
    return capitalize(super.getFirstName());
  }
  private String capitalize(String s) {
    StringBuilder result = new StringBuilder(s);
    result.replace(0,1, result.substring(0,1).toUpperCase());
    return result.toString();
  }
  public String getFullName() {
    return getFirstName() + " " + getLastName();
  }
```

Помимо доступа к данным, я хочу также иметь возможность проверок. В настоящий момент я делаю это путем добавления кода к создаваемому вручную подтипу. Однако я хочу, чтобы все методы проверок могли легко запускаться вместе. Этого можно добиться путем добавления кода в базовый класс, написанный вручную.

```
class AbstractData...
  public Notification validate() {
    Notification note = new Notification();
    checkAllFields(note);
    checkClass(note);
    return note;
  }
```

```
protected abstract void checkAllFields(Notification note);  
protected void checkClass(Notification note) {}
```

Здесь метод проверки вызывает абстрактный метод для проверки всех полей по отдельности и пустой метод для проверок, которые включают несколько полей. Идея в том, что я могу переопределить этот метод в своем конкретном рукописном классе. Генерируемый класс реализует абстрактный метод с помощью той же информации, которая использовалась для генерации полей.

```
class PersonDataGen...  
protected void checkAllFields(Notification note) {  
    checkFirstName (note);  
    checkLastName (note);  
    checkEmpID (note);  
}
```

Как вы могли заметить, в предыдущем примере кода эти методы проверок были просто пустыми. Я могу их перекрыть, добавив в них некоторое проверяющее поведение.

```
class PersonData...  
protected void checkEmpID(Notification note) {  
    if (getEmpID() < 1)  
        note.error("Идентификатор должен быть положительным");  
}
```


Список литературы

1. Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition. Addison-Wesley. 2006. ISBN 0321486811 (Альфред В. Ахо, Моника С. Лам, Рави Сети, Джейфри Д. Ульман. *Компиляторы: принципы, технологии и инструментарий*, 2 издание. ИД "Вильямс", 2008.)
2. Anderson, Chris. *Essential Windows Presentation Foundation*. Addison- Wesley. ISBN 0321374479.
3. Beck, Kent. *Implementation Patterns*. Addison-Wesley. ISBN 0321413091 (Кент Бек. *Шаблоны реализации корпоративных приложений*. ИД "Вильямс", 2008.)
4. Beck, Kent. *Test-Driven Development*. Addison-Wesley. ISBN 0321146530.
5. Beck, Kent. *Smalltalk Best Practice Patterns*. Addison-Wesley. ISBN 013476904X.
6. Cross, Bradford. *The Compositional DSL vs. Computational DSL Smack Down*. <http://measuringmeasures.blogspot.com/2009/02/compositional-dsl-vs-computational-dsl.html>.
7. Evans, Eric. *Domain-Driven Design*. Addison-Wesley. ISBN 0321125215 (Эрик Эванс. *Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем*. ИД "Вильямс", 2010.)
8. <http://martinfowler.com/bliki/ComposedRegex.html>.
9. <http://martinfowler.com/articles/evodb.html>.
10. Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 0321127420 (Мартин Фаулер. *Шаблоны корпоративных приложений*. ИД "Вильямс", 2009.)
11. Fowler, Martin. *Analysis Patterns*. Addison-Wesley. ISBN 0201895420.
12. Fowler, Martin. *Refactoring*. Addison-Wesley. ISBN 0201485672.
13. Freeman, Steve and Nat Pryce. "Evolving an Embedded Domain-Specific Language in Java." In: *Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications..* www.jmock.org/oopsla2006.pdf.
14. <http://martinfowler.com/articles/rake.html>.
15. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley. ISBN 0201633612.
16. www.paulgraham.com/onlisp.html.
17. Herrington, Jack. *Code Generation in Action*. Manning. ISBN 1930110979.
18. Kabanov, Jevgeni, Michael Hunger, and Rein Raudjdrv. *On Designing Safe and Embedded DSLs with Java 5*.
19. Hohpe, Gregor and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley. ISBN 0321200683.
20. Meszaros, Gerard. *xUnit Test Patterns*. Addison-Wesley. ISBN 0131495054.
21. Meyer, Bertrand. *Object-Oriented Software Construction*. Addison-Wesley. ISBN 0136291554.
22. Parr, Terence. *The Definitive Antlr Reference*. Pragmatic Bookshelf. 2007. ISBN 0978739256.
23. Parr, Terence. *Language Implementation Patterns*. Pragmatic Bookshelf. 2009. ISBN 193435645X.
24. <http://tools.ietf.org/html/rfc3501>.
25. <http://tools.ietf.org/html/rfc5322>.
26. www.adaptiveobjectmodel.com/WICSA3/ArchitectureOfAOMsWICSA3.htm.
27. www.voelter.de/data/pub/ProgramGeneration.pdf.

Предметный указатель

A

ANTLR, 108; 278
API командных запросов, 41

D

DSL, 17; 49
внешний, 39; 50; 105
внутренний, 39; 50
встроенный, 39
дизайн, 62
жизненный цикл, 60
миграция, 82
обучение, 119
определение, 49
тестирование, 72

F

Fitness, 166

G

Graphviz, 159

P

Parsing Expression Grammar, 113
PEG, 113

Y

Yacc, 115

A

Абстрактное синтаксическое дерево, 235
Адаптивная модель, 42
Альтернативная токенизация, 325
Аннотация, 100; 439
Анонимная функция, 397
Атрибут, 100

B

Видимость
глобальная, 358
Визуализация, 48
Вложенная функция, 361
Вложенное операторное выражение, 333
Внешний код, 315
Восходящий синтаксический анализ, 114
Встраиваемое вложение, 97
Встроенная интерпретация, 109; 311
Встроенная трансляция, 305
Встроенный помощник, 538
Вычисление экземпляра, 394

Г

Генератор синтаксических анализаторов, 107; 277
Генерация кода, 43; 135; 545
игнорирующая модель, 557
осведомленная о модели, 545
разделение, 561
с помощью преобразователя, 523
удобочитаемость, 140
шаблонная, 529
Глобальная видимость, 358
Грамматика, 68; 106; 230
альтернатива, 238
контекстно-зависимая, 113
контекстно-свободная, 112; 240
модульная, 345
разбирающая выражения, 240
регулярная, 112
синтаксического анализа выражений, 113
Терминал, 238

Д

Двойная проверка, 80
Декларативное программирование, 37; 43
Дерево
разбора, 67
синтаксическое, 450
Динамический отклик, 102; 423

3

Замыкание, 96; 397
 вложенное, 403
Захват переменной, 198

И

Иерархия Хомского, 112
Иллюстративное программирование, 47; 151
Интерпретация, 43
 встроенная, 311
Интерфейс свободный, 41

К

Класс таблицы символов, 461
Коллизия имен, 363
Комментарий, 232
Компиляция, 43
Конечный автомат, 30; 132; 517
Контекстно-зависимая грамматика, 113
Контекстно-свободная грамматика, 112

Л

Левая рекурсия, 115; 242; 257
Левая факторизация, 115
Лексический анализ, 110
Лексический анализатор, 231; 247
Литерал, 473
Лямбда, 397

М

Макрос, 72; 195; 203
 захват переменной, 198
 многократное вычисление, 197
 синтаксический, 199
Метамодель, 147
Метаморфозы типа, 377; 473
Миграция инкрементная, 83
Миграция на основе моделей, 83
Многократное вычисление, 197
Модель
 адаптивная, 42; 129
 вычислительная альтернативная, 127
 декларативная, 37
 императивная, 37; 127
 предметной области, 172
 семантическая, 41

Модульная грамматика, 345

Н

Нисходящий синтаксический анализ, 114

О

Обработка ошибок, 80
Обратное умозаключение, 505
Оператор
 до, 240
 клини, 239
Опережающая ссылка, 179
Опубликованный интерфейс, 82
Отложенное вычисление, 97; 200
Отображение, 94

П

Переменная контекста, 71; 90; 187
Перенос области видимости, 387
Последовательность функций, 357
Постройтель выражений, 349
Постройтель конструкции, 191
Правая рекурсия, 242
Предметно-ориентированный язык,
 см. DSL
Предметный язык, 53
Пробельный символ, 232
Проверка типов, 103
Проекционное редактирование, 149
Прямое умозаключение, 505

Р

Разделение запросов и команд, 87
Расширение литералов, 101
Регулярная грамматика, 112
Регулярные выражения, 52
 составные, 220
Рекурсивный спуск, 107; 253

С

Сбрасывающее событие, 31
Свободный интерфейс, 41; 85; 349
Семантика, 41
Семантическая модель, 42; 171
 интерфейс наполнения, 173
 операционный интерфейс, 173
Семантический предикат, 279

Сеть зависимостей, 132; 168; 495

Символ, 95

Синтаксис, 41; 69

Синтаксически управляемая трансляция,
229

Синтаксический анализ, 105

LL и LR, 114

восходящий, 114

нисходящий, 114

Синтаксический анализатор, 67; 233

генератор, 107

рекурсивного спуска, 107; 253

Синтаксический пробел, 110

Синтаксическое дерево, 67; 98; 235; 290; 450

абстрактное, 290

Система правил вывода, 130; 503

Список литералов, 415

Статическая типизация, 183

Статические данные, 358

Стековая машина, 231; 328

Т

Таблица принятия решений, 130; 485

Таблица символов, 70; 113; 177

класс, 461

Тестирование, 72

Токенизатор, 231

Трансляция

встроенная, 108; 305

синтаксически управляемая, 229

управляемая разделителями, 106; 213

У

Уведомление, 205

Управляемая разделителями трансляция, 213

Ф

Форма Бэкуса–Наура, 237

Функция анонимная, 397

III

Шлифовка текста, 469

Шум, 18; 98; 101; 117

Э

Электронные таблицы, 47; 152

Я

Язык

Algol, 237

Ant, 122; 168

Applescript, 62

C, 196; 546

C#, 18; 94; 399

C++, 196; 199

COBOL, 19

Common Lisp, 201

CSS, 162

DOT, 159

Graphviz, 48

HQL, 163

Java, 18; 94

JSON, 118

Linq, 99; 451

Lisp, 50; 92; 99; 195; 199; 416

Make, 122; 167

PIC, 165

PL/1, 232

Python, 344

Ruby, 18; 94; 198; 388; 400; 445

Smalltalk, 98; 474

XAML, 164

XML, 117

YAML, 118; 344

определения схемы, 147

синтаксис, 41

шаблонов, 530

Язык программирования, 49

Языковая какофония, 58

Языковые инструментальные средства, 46;

50; 143

CASE-инструменты, 154

Intentional Workbench, 153

MetaEdit, 153

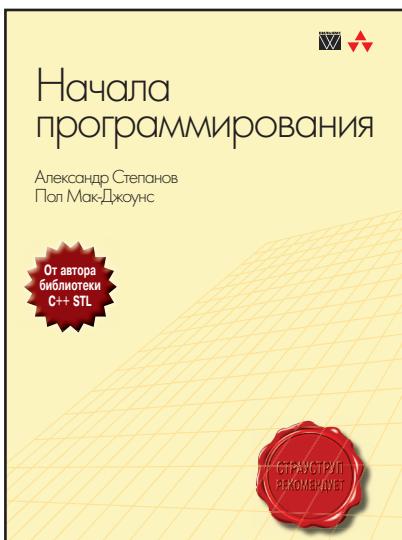
Meta-Programming System, 154

SQL Server Modeling, 154

Xtext, 154

НАЧАЛА ПРОГРАММИРОВАНИЯ

**Александр Степанов,
Пол Мак-Джоунс**



www.williamspublishing.com

В настоящей книге применяется дедуктивный подход к программированию, основанный на объединении программ с абстрактными математическими теориями, которые обеспечивают их работу. Представлены вместе описания этих теорий, алгоритмы, записанные с точки зрения этих теорий, а также теоремы и леммы, описывающие их свойства. Реализация алгоритмов на реальном языке программирования является центральной темой книги. Эта книга предназначена для тех, кто стремится глубже понять суть программирования, будь то профессиональные программисты или ученые и инженеры, для которых программирование составляет важную часть их профессиональной деятельности.

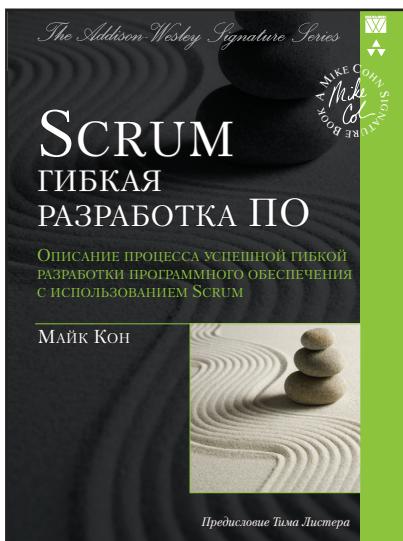
ISBN 978-5-8459-1708-9 в продаже

SCRUM

ГИБКАЯ РАЗРАБОТКА ПО

**ОПИСАНИЕ ПРОЦЕССА УСПЕШНОЙ ГИБКОЙ РАЗРАБОТКИ ПРОГРАММНОГО
ОБЕСПЕЧЕНИЯ С ИСПОЛЬЗОВАНИЕМ SCRUM**

Майк Кон



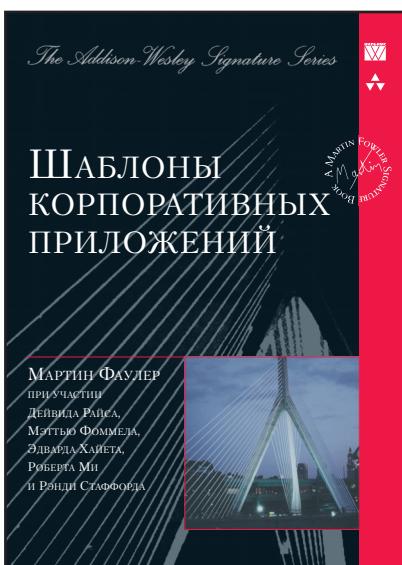
www.williamspublishing.com

Данная книга предназначена для pragматичных специалистов в области разработки программного обеспечения, которые хотят получить надежные, заслуживающие доверия ответы на большинство трудных вопросов, с которыми им приходится сталкиваться в процессе внедрения Scrum. В своей книге автор описывает все аспекты процесса внедрения: запуск процесса, оказание людям помощи в освоении новых ролей, структуризация коллективов, увеличение охвата, работа с рассредоточенным коллективом и, наконец, внедрение эффективных показателей и непрерывное совершенствование. В книге встречаются врезки под заголовком "Попробуйте прямо сейчас", включающие наиболее эффективные советы автора. Во врезках под заголовком "Возражения" автор воспроизводит типичные дискуссии с теми, кто сопротивляется переменам.

ISBN 978-5-8459-1731-7 **в продаже**

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

ISBN 978-5-8459-1611-2

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

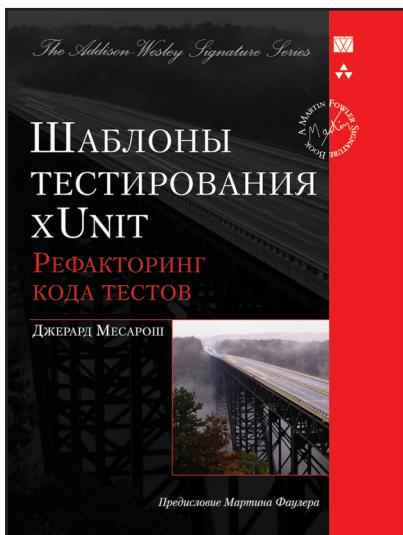
в продаже

ШАБЛОНЫ ТЕСТИРОВАНИЯ

XUNIT

РЕФАКТОРИНГ КОДА ТЕСТОВ

Джерард Месарош



www.williamspublishing.com

ISBN 978-5-8459-1448-4

В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. Книга состоит из трех частей.

В первой части приводятся теоретические основы методов разработки тестов, описываются концепции шаблонов и "запахов" тестов (признаков существующей проблемы). Во второй и третьей частях книги приводится каталог шаблонов проектирования тестов, "запахов" и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в третьей части книги сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения. Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки.

в продаже