Содержание

Предисловие 24

Глава 1. Введение в компиляцию 29
1.1 Компиляторы 29
1.1.1 Упражнения к разделу 1.1 32
1.2 Структура компилятора 32
1.2.1 Лексический анализ 33
1.2.2 Синтаксический анализ 35
1.2.3 Семантический анализ 37
1.2.4 Генерация промежуточного кода 38
1.2.5 Оптимизация кода 39
1.2.6 Генерация кода 39
1.2.7 Управление таблицей символов 40
1.2.8 Объединение фаз в проходы 41
1.2.9 Инструментарий для создания компиляторов 41
1.3 Эволюция языков программирования 42
1.3.1 Переход к языкам высокого уровня 42
1.3.2 Влияние на компиляторы 44
1.3.3 Упражнения к разделу 1.3 45
1.4 “Компилятороведение” 45
1.4.1 Моделирование при проектировании и реализации компилятора 45
1.4.2 Изучение оптимизации кода 46
1.5 Применения технологий компиляторов 48
1.5.1 Реализация высокоуровневых языков программирования 48
1.5.2 Оптимизация для архитектуры компьютера 50
1.5.3 Разработка новых архитектур компьютеров 52
1.5.4 Трансляции программ 54
1.5.5 Инструментарий для повышения производительности программного обеспечения 55

1.6 Азы языков программирования 57
1.6.1 Понятия статического и динамического 58
1.6.2 Среды и состояния 58
1.6.3 Статическая область видимости и блочная структура 61
1.6.4 Явное управление доступом 65
1.6.5 Динамическая область видимости 65
1.6.6 Механизмы передачи параметров 68
1.6.7 Псевдонимы 70
1.6.8 Упражнения к разделу 1.6 70
1.7 Резюме к главе 1 72
1.8 Список литературы к главе 1 73

Глава 2. Простой синтаксически управляемый транслятор 75
2.1 Введение 76
2.2 Определение синтаксиса 78
2.2.1 Определения грамматик 79
2.2.2 Выведение 81
2.2.3 Деревья разбора 82
2.2.4 Неоднозначности 84
2.2.5 Ассоциативность операторов 85
2.2.6 Приоритет операторов 86
2.2.7 Упражнения к разделу 2.2 89
2.3 Синтаксически управляемая трансляция 90
2.3.1 Постфиксная запись 91
2.3.2 Синтезированные атрибуты 92
2.3.3 Простые синтаксически управляемые определения 94
2.3.4 Обходы дерева 95
2.3.5 Схемы трансляции 97
2.3.6 Упражнения к разделу 2.3 99
2.4 Разбор 100
2.4.1 Нисходящий анализ 101
2.4.2 Предиктивный анализ 104
2.4.3 Использование пустых продукций 106
2.4.4 Разработка предиктивного анализатора 106
2.4.5 Левая рекурсия 107
2.4.6 Упражнения к разделу 2.4 109
2.5 Транслятор простых выражений 109
2.5.1 Абстрактный и конкретный синтаксис 110
2.5.2 Адаптация схемы трансляции 111
2.5.3 Процедуры для нетерминалов 113
2.5.4 Упрощение транслятора 114

2.5.5 Завершенная программа 115
2.6 Лексический анализ 118
2.6.1 Удаление пробельных символов и комментариев 119
2.6.2 Опережающее чтение 120
2.6.3 Константы 121
2.6.4 Распознавание ключевых слов и идентификаторов 121
2.6.5 Лексический анализатор 123
2.6.6 Упражнения к разделу 2.6 128
2.7 Таблицы символов 128
2.7.1 Таблица символов для области видимости 129
2.7.2 Использование таблиц символов 133
2.8 Генерация промежуточного кода 136
2.8.1 Два вида промежуточных представлений 136
2.8.2 Построение синтаксических деревьев 137
2.8.3 Статические проверки 142
2.8.4 Трехадресный код 144
2.8.5 Упражнения к разделу 2.8 151
2.9 Резюме к главе 2 152

Глава 3. Лексический анализ 155
3.1 Роль лексического анализатора 155
3.1.1 Лексический и синтаксический анализ 157
3.1.2 Токены, шаблоны и лексемы 157
3.1.3 Атрибуты токенов 159
3.1.4 Лексические ошибки 160
3.1.5 Упражнения к разделу 3.1 161
3.2 Буферизация ввода 162
3.2.1 Пары буферов 162
3.2.2 Ограничители 163
3.3 Спецификация токенов 165
3.3.1 Строки и языки 165
3.3.2 Операции над языками 166
3.3.3 Регулярные выражения 168
3.3.4 Регулярные определения 170
3.3.5 Расширения регулярных выражений 172
3.3.6 Упражнения к разделу 3.3 173
3.4 Распознавание токенов 177
3.4.1 Диаграммы переходов 179
3.4.2 Распознавание зарезервированных слов и идентификаторов 181
3.4.3 Завершение примера 183

3.4.4 Архитектура лексического анализатора на основе диаграммы переходов 184
3.4.5 Упражнения к разделу 3.4 187
3.5 Генератор лексических анализаторов Lex 191
3.5.1 Использование Lex 191
3.5.2 Структура программ Lex 192
3.5.3 Разрешение конфликтов в Lex 196
3.5.4 Прогностический оператор 197
3.5.5 Упражнения к разделу 3.5 198
3.6 Конечные автоматы 199
3.6.1 Недетерминированные конечные автоматы 200
3.6.2 Таблицы переходов 201
3.6.3 Принятие входной строки автоматом 201
3.6.4 Детерминированный конечный автомат 203
3.6.5 Упражнения к разделу 3.6 204
3.7 От регулярных выражений к автоматам 205
3.7.1 Преобразование НКА в ДКА 206
3.7.2 Моделирование НКА 210
3.7.3 Эффективность моделирования НКА 210
3.7.4 Построение НКА из регулярного выражения 213
3.7.5 Эффективность алгоритма обработки строк 218
3.7.6 Упражнения к разделу 3.7 221
3.8 Разработка генератора лексических анализаторов 221
3.8.1 Структура генерируемого анализатора 221
3.8.2 Распознавание шаблонов на основе НКА 223
3.8.3 ДКА для лексических анализаторов 225
3.8.4 Реализация прогностического оператора 226
3.8.5 Упражнения к разделу 3.8 228
3.9 Оптимизация распознавателей на основе ДКА 229
3.9.1 Важные состояния НКА 229
3.9.2 Функции, вычисляемые на синтаксическом дереве 231
3.9.3 Вычисление nullable, firstpos и lastpos 232
3.9.4 Вычисление followpos 234
3.9.5 Преобразование регулярного выражения непосредственно в ДКА 235
3.9.6 Минимизация количества состояний ДКА 237
3.9.7 Минимизация состояний в лексических анализаторах 242
3.9.8 Компромисс между скоростью и используемой памятью при моделировании ДКА 242
3.9.9 Упражнения к разделу 3.9 244

3.10 Резюме к главе 3 245
3.11 Список литературы к главе 3 247

Глава 4. Синтаксический анализ 251
4.1 Введение 252
4.1.1 Роль синтаксического анализатора 252
4.1.2 Образцы грамматик 253
4.1.3 Обработка синтаксических ошибок 254
4.1.4 Стратегии восстановления после ошибок 256
4.2 Контекстно-свободные грамматики 258
4.2.1 Формальное определение контекстно-свободной грамматики 258
4.2.2 Соглашения об обозначениях 260
4.2.3 Порождения 261
4.2.4 Деревья разбора и порождения 263
4.2.5 Неоднозначность 265
4.2.6 Проверка языка, сгенерированного грамматикой 266
4.2.7 Контекстно-свободные грамматики и регулярные выражения 267
4.2.8 Упражнения к разделу 4.2 269
4.3 Разработка грамматики 272
4.3.1 Лексический и синтаксический анализ 273
4.3.2 Устранение неоднозначности 273
4.3.3 Устранение левой рекурсии 275
4.3.4 Левая факторизация 278
4.3.5 Не контекстно-свободные языковые конструкции 279
4.3.6 Упражнения к разделу 4.3 280
4.4 Нисходящий синтаксический анализ 281
4.4.1 Синтаксический анализ методом рекурсивного спуска 283
4.4.2 FIRST и FOLLOW 285
4.4.3 LL(1)-грамматики 288
4.4.4 Нерекурсивный предиктивный синтаксический анализ 292
4.4.5 Восстановление после ошибок в предиктивном синтаксическом анализе 295
4.4.6 Упражнения к разделу 4.4 298
4.5 Восходящий синтаксический анализ 301
4.5.1 Свертки 302
4.5.2 Обрезка основ 302
4.5.3 Синтаксический анализ “перенос/свертка” 304
4.5.4 Конфликты в процессе ПС-анализа 306
4.5.5 Упражнения к разделу 4.5 309

4.6 Введение в LR-анализ: простой LR 309
4.6.1 Обоснование использования LR-анализаторов 310
4.6.2 Пункты и LR(0)-автомат 311
4.6.3 Алгоритм LR-анализа 317
4.6.4 Построение таблиц SLR-анализа 322
4.6.5 Активные префиксы 326
4.6.6 Упражнения к разделу 4.6 328
4.7 Более мощные LR-анализаторы 330
4.7.1 Канонические LR(1)-пункты 331
4.7.2 Построение множеств LR(1)-пунктов 332
4.7.3 Канонические таблицы LR(1)-анализа 336
4.7.4 Построение LALR-таблиц синтаксического анализа 338
4.7.5 Эффективное построение таблиц LALR-анализа 344
4.7.6 Уплотнение таблиц LR-анализа 349
4.7.7 Упражнения к разделу 4.7 352
4.8 Использование неоднозначных грамматик 353
4.8.1 Использование приоритетов и ассоциативности для разрешения конфликтов 353
4.8.2 Неоднозначность “висящего else” 357
4.8.3 Восстановление после ошибок в LR-анализе 358
4.8.4 Упражнения к разделу 4.8 361
4.9 Генераторы синтаксических анализаторов 363
4.9.1 Генератор синтаксических анализаторов `Yacc` 364
4.9.2 Использование `Yacc` с неоднозначной грамматикой 368
4.9.3 Создание лексического анализатора в `Yacc` с помощью `Lex` 371
4.9.4 Восстановление после ошибок в `Yacc` 372
4.9.5 Упражнения к разделу 4.9 375
4.10 Резюме к главе 4 375
4.11 Список литературы к главе 4 378

Глава 5. Синтаксически управляемая трансляция 383
5.1 Синтаксически управляемые определения 384
5.1.1 Наследуемые и синтезируемые атрибуты 384
5.1.2 Вычисление СУО в узлах дерева разбора 387
5.1.3 Упражнения к разделу 5.1 390
5.2 Порядок вычисления в СУО 391
5.2.1 Графы зависимостей 391
5.2.2 Упорядочение вычисления атрибутов 393
5.2.3 S-атрибутные определения 394
5.2.4 L-атрибутные определения 394

5.2.5 Семантические правила с контролируемыми побочными действиями 396
5.2.6 Упражнения к разделу 5.2 398
5.3 Применения синтаксически управляемой трансляции 399
5.3.1 Построение синтаксических деревьев 400
5.3.2 Структура типа 404
5.3.3 Упражнения к разделу 5.3 406
5.4 Синтаксически управляемые схемы трансляции 406
5.4.1 Постфиксные схемы трансляции 407
5.4.2 Реализация постфиксной СУТ с использованием стека синтаксического анализатора 407
5.4.3 СУТ с действиями внутри продукций 410
5.4.4 Устранение левой рекурсии из СУТ 411
5.4.5 СУТ для L-атрибутных определений 414
5.4.6 Упражнения к разделу 5.4 421
5.5 Реализация L-атрибутных СУО 422
5.5.1 Трансляция в процессе синтаксического анализа методом рекурсивного спуска 423
5.5.2 Генерация кода “на лету” 425
5.5.3 L-атрибутные СУО и LL-синтаксический анализ 428
5.5.4 Восходящий синтаксический анализ L-атрибутных СУО 434
5.5.5 Упражнения к разделу 5.5 439
5.6 Резюме к главе 5 439
5.7 Список литературы к главе 5 441

Глава 6. Генерация промежуточного кода 443
6.1 Варианты синтаксических деревьев 445
6.1.1 Ориентированные ациклические графы для выражений 445
6.1.2 Метод номера значения для построения ориентированных ациклических графов 447
6.1.3 Упражнения к разделу 6.1 450
6.2 Трехадресный код 450
6.2.1 Адреса и команды 450
6.2.2 Четверки 454
6.2.3 Тройки 455
6.2.4 Представление в виде статических единственных присваиваний 457
6.2.5 Упражнения к разделу 6.2 458
6.3 Типы и объявления 459
6.3.1 Выражения типов 459
6.3.2 Эквивалентность типов 461

6.3.3 Объявления 462
6.3.4 Размещение локальных имен в памяти 462
6.3.5 Последовательности объявлений 464
6.3.6 Поля в записях и классах 466
6.3.7 Упражнения к разделу 6.3 467
6.4 Трансляция выражений 468
6.4.1 Операции в выражениях 468
6.4.2 Инкрементная трансляция 470
6.4.3 Адресация элементов массива 471
6.4.4 Трансляция обращений к массиву 473
6.4.5 Упражнения к разделу 6.4 475
6.5 Проверка типов 477
6.5.1 Правила проверки типов 477
6.5.2 Преобразования типов 478
6.5.3 Перегрузка функций и операторов 481
6.5.4 Выведение типа и полиморфные функции 482
6.5.5 Алгоритм унификации 487
6.5.6 Упражнения к разделу 6.5 490
6.6 Поток управления 491
6.6.1 Булевы выражения 492
6.6.2 Код сокращенного вычисления 492
6.6.3 Инструкции потока управления 493
6.6.4 Трансляция логических выражений с помощью потока управления 496
6.6.5 Устранение излишних команд перехода 499
6.6.6 Булевы значения и код с переходами 501
6.6.7 Упражнения к разделу 6.6 502
6.7 Обратные поправки 504
6.7.1 Однопроходная генерация кода с использованием обратных поправок 504
6.7.2 Обратные поправки для булевых выражений 505
6.7.3 Инструкции потока управления 508
6.7.4 Инструкции break, continue и goto 511
6.7.5 Упражнения к разделу 6.7 512
6.8 Инструкции выбора 514
6.8.1 Трансляция инструкций выбора 514
6.8.2 Синтаксически управляемая трансляция инструкций выбора 515
6.8.3 Упражнения к разделу 6.8 517
6.9 Промежуточный код процедур 517

6.10 Резюме к главе 6 520
6.11 Список литературы к главе 6 521

Глава 7. Среды времени выполнения 525
7.1 Организация памяти 525
7.1.1 Статическое и динамическое распределение памяти 527
7.2 Выделение памяти в стеке 528
7.2.1 Деревья активации 529
7.2.2 Записи активации 532
7.2.3 Последовательности вызовов 535
7.2.4 Данные переменной длины в стеке 539
7.2.5 Упражнения к разделу 7.2 540
7.3 Доступ к нелокальным данным в стеке 542
7.3.1 Доступ к данным при отсутствии вложенных процедур 542
7.3.2 Вложенные процедуры 543
7.3.3 Язык с вложенными объявлениями процедур 544
7.3.4 Глубина вложенности 545
7.3.5 Связи доступа 547
7.3.6 Работа со связями доступа 548
7.3.7 Связи доступа для процедур, являющихся параметрами 550
7.3.8 Дисплеи 551
7.3.9 Упражнения к разделу 7.3 554
7.4 Управление кучей 555
7.4.1 Диспетчер памяти 555
7.4.2 Иерархия памяти компьютера 557
7.4.3 Локальность в программах 559
7.4.4 Снижение фрагментации 561
7.4.5 Освобождение памяти вручную 565
7.4.6 Упражнения к разделу 7.4 568
7.5 Введение в сборку мусора 568
7.5.1 Цели проектирования сборщиков мусора 569
7.5.2 Достижимость 572
7.5.3 Сборщики мусора с подсчетом ссылок 574
7.5.4 Упражнения к разделу 7.5 576
7.6 Введение в сборку на основе отслеживания 576
7.6.1 Базовый сборщик мусора 577
7.6.2 Базовая абстракция 580
7.6.3 Оптимизация алгоритма “пометить и подмести” 582
7.6.4 Сборщики мусора “пометить и сжать” 583
7.6.5 Копирующие сборщики 587
7.6.6 Сравнение стоимости 589

7.6.7 Упражнения к разделу 7.6 590
7.7 Распределенная сборка мусора 591
7.7.1 Инкрементная сборка мусора 591
7.7.2 Инкрементный анализ достижимости 593
7.7.3 Основы частичной сборки 596
7.7.4 Сборка мусора по поколениям 597
7.7.5 Алгоритм поезда 599
7.7.6 Упражнения к разделу 7.7 603
7.8 Дополнительные вопросы сборки мусора 604
7.8.1 Параллельная сборка мусора 605
7.8.2 Частичное перемещение объектов 608
7.8.3 Консервативная сборка мусора для небезопасных языков программирования 608
7.8.4 Слабые ссылки 609
7.8.5 Упражнения к разделу 7.8 610
7.9 Резюме к главе 7 611
7.10 Список литературы к главе 7 614

Глава 8. Генерация кода 617
8.1 Вопросы проектирования генератора кода 619
8.1.1 Вход генератора кода 619
8.1.2 Целевая программа 620
8.1.3 Выбор команд 621
8.1.4 Распределение регистров 623
8.1.5 Порядок вычислений 625
8.2 Целевой язык 625
8.2.1 Простая модель целевой машины 625
8.2.2 Стоимость программ и команд 629
8.2.3 Упражнения к разделу 8.2 630
8.3 Адреса в целевом коде 632
8.3.1 Статическое выделение памяти 632
8.3.2 Выделение памяти в стеке 635
8.3.3 Адреса имен времени выполнения 638
8.3.4 Упражнения к разделу 8.3 639
8.4 Базовые блоки и графы потоков 640
8.4.1 Базовые блоки 641
8.4.2 Информация о дальнейшем использовании 643
8.4.3 Графы потоков 644
8.4.4 Представление графов потоков 646
8.4.5 Циклы 646
8.4.6 Упражнения к разделу 8.4 647

8.5 Оптимизация базовых блоков 648
8.5.1 Представление базовых блоков с использованием ориентированных ациклических графов 648
8.5.2 Поиск локальных общих подвыражений 649
8.5.3 Устранение неиспользуемого кода 651
8.5.4 Применение алгебраических тождеств 652
8.5.5 Представление обращений к массивам 653
8.5.6 Присваивание указателей и вызовы процедур 655
8.5.7 Сборка базового блока из ориентированного ациклического графа 656
8.5.8 Упражнения к разделу 8.5 658
8.6 Простой генератор кода 660
8.6.1 Дескрипторы регистров и адресов 661
8.6.2 Алгоритм генерации кода 661
8.6.3 Разработка функции *getReg* 665
8.6.4 Упражнения к разделу 8.6 667
8.7 Локальная оптимизация 668
8.7.1 Устранение излишних загрузок и сохранений 668
8.7.2 Устранение недостижимого кода 669
8.7.3 Оптимизация потока управления 670
8.7.4 Алгебраические упрощения и снижение стоимости 671
8.7.5 Использование машинных идиом 671
8.7.6 Упражнения к разделу 8.7 671
8.8 Распределение и назначение регистров 672
8.8.1 Глобальное распределение регистров 672
8.8.2 Счетчики использований 673
8.8.3 Назначение регистров для внешних циклов 676
8.8.4 Распределение регистров путем раскраски графа 676
8.8.5 Упражнения к разделу 8.8 677
8.9 Выбор команд путем переписывания дерева 677
8.9.1 Схемы трансляции деревьев 678
8.9.2 Генерация кода путем замощения входного дерева 681
8.9.3 Поиск соответствий с использованием синтаксического анализа 684
8.9.4 Программы семантической проверки 685
8.9.5 Обобщенный поиск соответствий 686
8.9.6 Упражнения к разделу 8.9 688
8.10 Генерация оптимального кода для выражений 688
8.10.1 Числа Ершова 689
8.10.2 Генерация кода на основе помеченных деревьев выражений 690

8.10.3 Вычисление выражений при недостаточном количестве регистров 692
8.10.4 Упражнения к разделу 8.10 694
8.11 Генерация кода с использованием динамического программирования 695
8.11.1 Последовательные вычисления 696
8.11.2 Алгоритм динамического программирования 697
8.11.3 Упражнения к разделу 8.11 700
8.12 Резюме к главе 8 700
8.13 Список литературы к главе 8 702

Глава 9. Машинно-независимые оптимизации 705
9.1 Основные источники оптимизации 706
9.1.1 Причины избыточности 706
9.1.2 Конкретный пример: быстрая сортировка 707
9.1.3 Трансформации, сохраняющие семантику 710
9.1.4 Глобальные общие подвыражения 710
9.1.5 Распространение копий 712
9.1.6 Удаление бесполезного кода 713
9.1.7 Перемещение кода 714
9.1.8 Переменные индукции и снижение стоимости 715
9.1.9 Упражнения к разделу 9.1 718
9.2 Введение в анализ потоков данных 719
9.2.1 Абстракция потока данных 720
9.2.2 Схема анализа потока данных 722
9.2.3 Схемы потоков данных в базовых блоках 723
9.2.4 Достигающие определения 725
9.2.5 Анализ активных переменных 733
9.2.6 Доступные выражения 735
9.2.7 Резюме 740
9.2.8 Упражнения к разделу 9.2 740
9.3 Основы анализа потока данных 743
9.3.1 Полурешетки 744
9.3.2 Передаточные функции 750
9.3.3 Итеративный алгоритм в обобщенной структуре 753
9.3.4 Смысл решения потока данных 755
9.3.5 Упражнения к разделу 9.3 759
9.4 Распространение констант 760
9.4.1 Значения потока данных для структуры распространения констант 761
9.4.2 Сбор в структуре распространения констант 762

9.4.3 Передаточные функции для структуры распространения констант 762
9.4.4 Монотонность структуры распространения констант 763
9.4.5 Недистрибутивность структуры распространения констант 764
9.4.6 Интерпретация результатов 765
9.4.7 Упражнения к разделу 9.4 767
9.5 Устранение частичной избыточности 768
9.5.1 Источники избыточности 769
9.5.2 Все ли избыточные вычисления могут быть устранены? 771
9.5.3 Отложенное перемещение кода 773
9.5.4 Ожидаемость выражений 774
9.5.5 Алгоритм отложенного перемещения кода 775
9.5.6 Упражнения к разделу 9.5 785
9.6 Циклы в графах потоков 787
9.6.1 Доминаторы 787
9.6.2 Упорядочение в глубину 790
9.6.3 Ребра в глубинном остовном дереве 792
9.6.4 Обратные ребра и приводимость 794
9.6.5 Глубина графа потока 795
9.6.6 Естественные циклы 796
9.6.7 Скорость сходимости итеративных алгоритмов потоков данных 798
9.6.8 Упражнения к разделу 9.6 801
9.7 Анализ на основе областей 803
9.7.1 Области 804
9.7.2 Иерархии областей для приводимых графов потоков 805
9.7.3 Обзор анализа на основании областей 808
9.7.4 Необходимые предположения о передаточных функциях 810
9.7.5 Алгоритм анализа на основе областей 811
9.7.6 Обработка неприводимых графов потоков 816
9.7.7 Упражнения к разделу 9.7 818
9.8 Символический анализ 819
9.8.1 Аффинные выражения ссылочных переменных 819
9.8.2 Формулировка задачи потока данных 822
9.8.3 Символический анализ на основе областей 827
9.8.4 Упражнения к разделу 9.8 832
9.9 Резюме к главе 9 833
9.10 Список литературы к главе 9 838

Глава 10. Параллелизм на уровне команд 841
10.1 Архитектуры процессоров 842
10.1.1 Конвейерная обработка команд и задержки ветвления 842
10.1.2 Конвейерное выполнение 843
10.1.3 Многоадресные команды 844
10.2 Ограничения планирования кода 845
10.2.1 Зависимость через данные 846
10.2.2 Поиск зависимостей среди обращений к памяти 846
10.2.3 Компромиссы между использованием регистров и параллелизмом 848
10.2.4 Упорядочение фаз распределения регистров и планирования кода 851
10.2.5 Зависимость от управления 852
10.2.6 Поддержка опережающего выполнения 853
10.2.7 Базовая модель машины 855
10.2.8 Упражнения к разделу 10.2 856
10.3 Планирование базовых блоков 857
10.3.1 Графы зависимости данных 858
10.3.2 Планирование списков базовых блоков 859
10.3.3 Приоритетные топологические порядки 861
10.3.4 Упражнения к разделу 10.3 863
10.4 Глобальное планирование кода 864
10.4.1 Примитивное перемещение кода 865
10.4.2 Восходящее перемещение кода 867
10.4.3 Нисходящее перемещение кода 868
10.4.4 Обновление зависимостей данных 870
10.4.5 Алгоритм глобального планирования 870
10.4.6 Усовершенствованные методы перемещения кода 874
10.4.7 Взаимодействие с динамическими планировщиками 875
10.4.8 Упражнения к разделу 10.4 876
10.5 Программная конвейеризация 876
10.5.1 Введение 877
10.5.2 Программная конвейеризация циклов 879
10.5.3 Распределение регистров и генерация кода 882
10.5.4 Циклы с зависимыми итерациями 883
10.5.5 Цели и ограничения программной конвейеризации 884
10.5.6 Алгоритм программной конвейеризации 888
10.5.7 Планирование ациклических графов зависимости данных 889
10.5.8 Планирование графов с циклическими зависимостями 891
10.5.9 Усовершенствования алгоритма конвейеризации 899

10.5.10 Модульное расширение переменных 900
10.5.11 Условные инструкции 903
10.5.12 Аппаратная поддержка программной конвейеризации 904
10.5.13 Упражнения к разделу 10.5 904
10.6 Резюме к главе 10 907
10.7 Список литературы к главе 10 909

Глава 11. Оптимизация параллелизма и локальности 911
11.1 Фундаментальные концепции 914
11.1.1 Многопроцессорность 914
11.1.2 Параллелизм в приложениях 917
11.1.3 Параллелизм на уровне циклов 918
11.1.4 Локальность данных 920
11.1.5 Введение в теорию аффинных преобразований 922
11.2 Пример посерьезнее: умножение матриц 927
11.2.1 Алгоритм умножения матриц 927
11.2.2 Оптимизации 930
11.2.3 Интерференция кэша 933
11.2.4 Упражнения к разделу 11.2 933
11.3 Пространства итераций 934
11.3.1 Построение пространств итераций вложений циклов 934
11.3.2 Порядок выполнения вложенности циклов 936
11.3.3 Матричная запись неравенств 938
11.3.4 Добавление символьных констант 938
11.3.5 Управление порядком выполнения 939
11.3.6 Изменение осей 944
11.3.7 Упражнения к разделу 11.3 945
11.4 Аффинные индексы массивов 948
11.4.1 Аффинные обращения к данным 948
11.4.2 Аффинное и неаффинное обращения на практике 949
11.4.3 Упражнения к разделу 11.4 950
11.5 Повторное использование данных 951
11.5.1 Типы повторных использований 952
11.5.2 Собственные повторные использования 953
11.5.3 Собственное пространственное повторное использование 958
11.5.4 Групповое повторное использование 959
11.5.5 Упражнения к разделу 11.5 962
11.6 Анализ зависимости данных в массивах 964
11.6.1 Определение зависимостей данных доступов к массивам 965
11.6.2 Целочисленное линейное программирование 966

11.6.3 НОД 967
11.6.4 Эвристики для решения задачи целочисленного линейного программирования 969
11.6.5 Решение обобщенной задачи целочисленного линейного программирования 973
11.6.6 Резюме 975
11.6.7 Упражнения к разделу 11.6 976
11.7 Поиск параллельности, не требующей синхронизации 978
11.7.1 Вводный пример 978
11.7.2 Разбиения аффинного пространства 981
11.7.3 Ограничения разбиений пространства 982
11.7.4 Решение ограничений разбиений пространств 986
11.7.5 Простой алгоритм генерации кода 990
11.7.6 Устранение пустых итераций 993
11.7.7 Устранение проверок из внутреннего цикла 996
11.7.8 Преобразования исходного кода 998
11.7.9 Упражнения к разделу 11.7 1003
11.8 Синхронизация между параллельными циклами 1005
11.8.1 Постоянное количество синхронизаций 1006
11.8.2 Графы зависимостей программ 1007
11.8.3 Иерархическое время 1009
11.8.4 Алгоритм распараллеливания 1012
11.8.5 Упражнения к разделу 11.8 1013
11.9 Конвейеризация 1013
11.9.1 Что такое конвейеризация 1014
11.9.2 Последовательная сверхрелаксация: пример 1016
11.9.3 Полностью переставляемые циклы 1017
11.9.4 Конвейеризация полностью переставляемых циклов 1018
11.9.5 Общая теория 1021
11.9.6 Ограничения временного разбиения 1022
11.9.7 Решение временны́х ограничений с использованием леммы Фаркаша 1026
11.9.8 Преобразования кода 1029
11.9.9 Параллелизм с минимальной синхронизацией 1035
11.9.10 Упражнения к разделу 11.9 1037
11.10 Оптимизации локальности 1039
11.10.1 Временна́я локальность вычисляемых данных 1040
11.10.2 Сжатие массива 1041
11.10.3 Чередование частей 1044
11.10.4 Алгоритмы оптимизации локальности 1047
11.10.5 Упражнения к разделу 11.10 1050

11.11 Прочие применения аффинных преобразований 1050
11.11.1 Машины с распределенной памятью 1050
11.11.2 Процессоры с одновременным выполнением нескольких команд 1051
11.11.3 Векторные и SIMD-команды 1052
11.11.4 Предвыборка 1053
11.12 Резюме к главе 11 1054
11.13 Список литературы к главе 11 1057

Глава 12. Межпроцедурный анализ 1061
12.1 Базовые концепции 1062
12.1.1 Графы вызовов 1062
12.1.2 Чувствительность к контексту 1064
12.1.3 Строки вызовов 1067
12.1.4 Контекстно-чувствительный анализ на основе клонирования 1069
12.1.5 Контекстно-чувствительный анализ на основе резюме 1070
12.1.6 Упражнения к разделу 12.1 1073
12.2 Необходимость межпроцедурного анализа 1075
12.2.1 Вызовы виртуальных методов 1076
12.2.2 Анализ псевдонимов указателей 1076
12.2.3 Параллельность 1077
12.2.4 Поиск программных ошибок и уязвимых мест 1077
12.2.5 SQL-ввод 1078
12.2.6 Переполнение буфера 1080
12.3 Логическое представление потока данных 1081
12.3.1 Введение в Datalog 1082
12.3.2 Правила Datalog 1083
12.3.3 Интенсиональные и экстенсиональные предикаты 1085
12.3.4 Выполнение программы Datalog 1088
12.3.5 Инкрементное вычисление программ Datalog 1090
12.3.6 Проблематичные правила Datalog 1091
12.3.7 Упражнения к разделу 12.3 1093
12.4 Простой алгоритм анализа указателей 1095
12.4.1 Сложность анализа указателей 1096
12.4.2 Модель указателей и ссылок 1097
12.4.3 Нечувствительность к потоку 1098
12.4.4 Формулировка с применением Datalog 1099
12.4.5 Использование информации о типе 1101
12.4.6 Упражнения к разделу 12.4 1102
12.5 Контекстно-нечувствительный межпроцедурный анализ 1105

12.5.1 Влияние вызовов методов 1105
12.5.2 Построение графа вызовов в Datalog 1107
12.5.3 Динамическая загрузка и отражение 1108
12.5.4 Упражнения к разделу 12.5 1109
12.6 Контекстно-чувствительный анализ указателей 1109
12.6.1 Контексты и строки вызовов 1110
12.6.2 Добавление контекста в правила Datalog 1113
12.6.3 Дополнительные наблюдения о чувствительности 1114
12.6.4 Упражнения к разделу 12.6 1115
12.7 Реализация Datalog с применением BDD 1115
12.7.1 Диаграммы бинарного выбора 1116
12.7.2 Преобразования диаграмм бинарного выбора 1118
12.7.3 Представление отношений при помощи BDD 1119
12.7.4 Операции отношений и BDD-операции 1120
12.7.5 Использование диаграмм бинарного выбора для анализа целей указателей 1123
12.7.6 Упражнения к разделу 12.7 1124
12.8 Резюме к главе 12 1124
12.9 Список литературы к главе 12 1128

Приложение А. Завершенный пример начальной стадии компилятора 1133
A.1 Исходный язык 1133
A.2 Main 1135
A.3 Лексический анализатор 1135
A.4 Таблицы символов и типы 1139
A.5 Промежуточный код выражений 1140
A.6 Переходы для булевых выражений 1144
A.7 Промежуточный код для инструкций 1149
A.8 Синтаксический анализатор 1153
A.9 Построение начальной стадии 1160

Приложение Б. Поиск линейно независимых решений 1163

Предметный указатель 1167

Глава 5

Синтаксически управляемая трансляция

В данной главе развивается затронутая в разделе 2.3 тема — трансляция языков, управляемая контекстно-свободной грамматикой. Методы трансляции из этой главы будут использоваться в главе 6 для проверки типов и генерации промежуточного кода. Эти методы применимы также при реализации небольших языков для специализированных задач; в данной главе приведен пример из полиграфии.

Мы связываем информацию с конструкциями языка, назначая *атрибуты* символу (или символам) грамматики, представляющему конструкцию (этот вопрос уже рассматривался в разделе 2.3.2). Синтаксически управляемое определение указывает значения атрибутов при помощи связывания с грамматическими продукциями семантических правил. Например, транслятор инфиксных выражений в постфиксные может иметь следующие продукцию и правило:

$$\begin{array}{ll} \text{Продукция} & \text{Семантическое правило} \\ E \to E_1 + T & E.code = E_1.code \| T.code \| ' + ' \end{array} \tag{5.1}$$

Эта продукция содержит два нетерминала, E и T; индекс у E_1 предназначен для того, чтобы отличать E в теле продукции от E в заголовке. И E, и T имеют атрибут *code*, представляющий собой строку. Семантическое правило указывает, что строка $E.code$ образуется путем конкатенации строк $E_1.code$, $T.code$ и символа '+'. Правило явно указывает, что трансляция E образуется из трансляций E, T и '+', но реализация в ходе непосредственной работы со строками может оказаться неэффективной.

В разделе 2.3.5 схема синтаксически управляемой трансляции вставляет в тела продукций программные фрагменты, называющиеся семантическими действиями, как в следующем случае:

$$E \to E_1 + T \left\{ print\ ' + ' \right\} \tag{5.2}$$

По соглашению семантические действия заключаются в фигурные скобки. (Если фигурная скобка встречается в качестве грамматического символа, она заключается в одинарные кавычки — '{' и '}'.) Положение семантического действия

в теле продукции определяет порядок, в котором выполняются действия. В продукции (5.2) действие выполняется в конце, после всех грамматических символов; в общем случае семантические действия могут располагаться в любом месте тела продукции.

Сравнивая эти два варианта записи, можно заметить, что синтаксически управляемые определения более удобочитаемы, а следовательно, более подходят в качестве спецификаций. Однако схемы трансляций могут оказаться более эффективны и потому в большей степени пригодны для реализаций.

Наиболее общий подход к синтаксически управляемой трансляции состоит в построении дерева разбора или синтаксического дерева с последующим вычислением значений атрибутов в узлах путем обхода узлов этого дерева. Во многих случаях трансляция может выполняться в процессе синтаксического анализа, без явного построения дерева разбора. Мы познакомимся с классом синтаксически управляемых трансляций, которые называются "L-атрибутными трансляциями" (здесь L означает "слева направо") и включают почти все трансляции, которые могут выполняться в процессе синтаксического анализа. Мы также изучим меньший класс — "S-атрибутные трансляции" (здесь S означает "синтезируемые"), которые легко выполняются при восходящем синтаксическом анализе.

5.1 Синтаксически управляемые определения

Синтаксически управляемое определение (СУО) является контекстно-свободной грамматикой с атрибутами и правилами. Атрибуты связаны с грамматическими символами, а правила — с продукциями. Если X представляет собой символ, а a — один из его атрибутов, то значение a в некотором узле дерева разбора, помеченном X, записывается как $X.a$. Если узлы дерева разбора реализованы в виде записей или объектов, то атрибуты X могут быть реализованы как поля данных в записях, представляющих узлы X. Атрибуты могут быть любого вида: числами, типами, таблицами ссылок или строками. Строки могут представлять собой, в частности, длинные последовательности кода, например кода на промежуточном языке, используемом компилятором.

5.1.1 Наследуемые и синтезируемые атрибуты

Мы будем работать с двумя типами атрибутов для нетерминалов.

1. *Синтезируемый атрибут* для нетерминала A в узле N дерева разбора определяется семантическим правилом, связанным с продукцией в N. Заметим, что в качестве заголовка этой продукции должен выступать нетерминал A. Синтезируемый атрибут в узле N определяется только с использованием значений атрибутов в дочерних по отношению к N узлах и в самом узле N.

Альтернативное определение наследуемых атрибутов

Если позволить наследуемому атрибуту $B.c$ в узле N быть определенным с использованием значений атрибутов в дочерних узлах N, а также в самом N, его родителе и братьях, то не нужны никакие дополнительные трансляции. Такие правила могут быть "имитированы" путем создания дополнительных атрибутов B, скажем, $B.c_1$, $B.c_2$, ... Это синтезируемые атрибуты, которые копируют необходимые атрибуты узлов, дочерних по отношению к узлу B. Затем $B.c$ вычисляется как наследуемый атрибут с использованием атрибутов $B.c_1$, $B.c_2$, ... вместо атрибутов дочерних узлов. На практике такие атрибуты требуются редко.

2. *Наследуемый атрибут* для нетерминала B в узле N дерева разбора определяется семантическим правилом, связанным с продукцией в N. Заметим, что нетерминал B должен участвовать в продукции в качестве символа в ее теле. Наследуемый атрибут в узле N определяется с использованием значений атрибутов в родительском по отношению к N узле, в самом узле N и дочерних узлах его родительского узла ("братьях" N).

Наследуемые атрибуты в узле N не могут определяться через атрибуты дочерних по отношению к N узлов; синтезируемые же атрибуты в узле N могут определяться через значения наследуемых атрибутов в самом узле N.

Терминалы могут иметь синтезируемые, но не наследуемые атрибуты. Атрибуты терминалов имеют лексические значения, придаваемые им лексическим анализатором. В СУО не существует семантических правил для вычисления значений атрибутов терминалов.

Пример 5.1. СУО на рис. 5.1 основано на знакомой нам грамматике для арифметических выражений с операторами + и *. Оно вычисляет выражения, завершающиеся ограничивающим маркером **n**. Каждый нетерминал в СУО имеет единственный синтезируемый атрибут *val*. Мы также считаем, что терминал **digit** имеет синтезируемый атрибут *lexval*, представляющий собой целое значение, возвращаемое лексическим анализатором.

Правило для продукции 1, $L \to E$ **n**, устанавливает $L.val$ равным $E.val$, которое, как мы увидим, равно числовому значению всего выражения.

Продукция 2, $E \to E_1 + T$, также имеет одно правило, которое вычисляет атрибут *val* заголовка E как сумму значений E_1 и T. В любом помеченном E узле N дерева разбора значение *val* для E представляет собой сумму значений *val* дочерних по отношению к N узлов с метками E и T.

	Продукция	Семантическое правило
1)	$L \to E$ **n**	$L.val = E.val$
2)	$E \to E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \to T$	$E.val = T.val$
4)	$T \to T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \to F$	$T.val = F.val$
6)	$F \to (E)$	$F.val = E.val$
7)	$F \to$ **digit**	$F.val =$ **digit**$.lexval$

Рис. 5.1. Синтаксически управляемое определение простого настольного калькулятора

Продукция 3, $E \to T$, имеет единственное правило, которое определяет значение *val* для E как значение того же атрибута у дочернего узла T. Продукция 4 подобна продукции 2: ее правило просто умножает значения атрибутов в дочерних узлах вместо их сложения. Правила продукций 5 и 6 копируют значения атрибутов дочерних узлов, так же, как и в продукции 3. Продукция 7 присваивает атрибуту $F.val$ числовое значение токена **digit**, возвращаемое лексическим анализатором.□

СУО, которые включают только синтезируемые атрибуты, называются *S-атрибутными* определениями; СУО на рис. 5.1 обладает данным свойством. В S-атрибутном СУО каждое правило вычисляет атрибут нетерминала в заголовке продукции, используя атрибуты, полученные из тела продукции.

Для простоты в примерах этого раздела используются семантические правила без побочных действий. На практике бывает удобно допускать в СУО небольшие ограниченные побочные действия, такие как печать результатов вычислений настольного калькулятора или работа с таблицей символов. При рассмотрении порядка вычислений атрибутов в разделе 5.2 мы позволим семантическим правилам вычислять произвольные функции, возможно, с побочными действиями.

S-атрибутное СУО может быть естественным образом реализовано вместе с LR-синтаксическим анализатором. Фактически СУО на рис. 5.1 отображает `Yacc`-программу, представленную на рис. 4.58, которая иллюстрирует трансляцию во время синтаксического анализа. Отличие заключается в том, что в правиле для продукции 1 `Yacc`-программа в качестве побочного действия выводит $E.val$ вместо определения значения атрибута $L.val$.

СУО без побочных эффектов иногда называют *атрибутной грамматикой*. Правила атрибутной грамматики определяют значения атрибутов через значения других атрибутов и констант и не выполняют никаких иных действий.

5.1.2 Вычисление СУО в узлах дерева разбора

Визуализировать трансляцию, определяемую СУО, может помочь работа с деревом разбора, хотя в действительности транслятор может его и не строить. Представим, что правила СУО применяются путем построения дерева разбора с последующим использованием этих правил для вычисления атрибутов в каждом узле дерева. Дерево разбора с указанием значений его атрибутов называется *аннотированным деревом разбора*.

Каким образом строится аннотированное дерево разбора? В каком порядке вычисляются его атрибуты? Перед тем как вычислять атрибут в узле дерева разбора, следует вычислить все атрибуты, от которых может зависеть вычисляемое значение. Например, если все атрибуты являются синтезируемыми, как в примере 5.1, то, прежде чем приступить к вычислению значения атрибута *val* в узле, требуется вычислить атрибуты *val* во всех его дочерних узлах.

Синтезируемые атрибуты можно вычислять в произвольном восходящем порядке, таком как обратный порядок обхода дерева разбора; вычисление S-атрибутных определений рассматривается в разделе 5.2.3.

Для СУО с атрибутами обоих типов — наследуемыми и синтезируемыми — не существует гарантии наличия даже одного порядка обхода для вычисления всех атрибутов в узлах дерева разбора. Рассмотрим, например, нетерминалы A и B с синтезируемым и наследуемым атрибутами $A.s$ и $B.i$ соответственно, а также следующую продукцию и правила:

Продукция	Семантические правила
$A \to B$	$A.s = B.i$
	$B.i = A.s + 1$

Данные правила циклические; невозможно вычислить ни $A.s$ в узле N, ни $B.i$ в дочернем по отношению к N узле, не зная значение другого атрибута. Циклическая зависимость $A.s$ и $B.i$ в некоторой паре узлов дерева разбора проиллюстрирована на рис. 5.2.

Задача поиска циклов в дереве разбора для данного СУО вычислительно достаточно сложна[1]. К счастью, существуют подклассы СУО, для которых можно гарантировать существование порядка вычисления (этот вопрос рассматривается в разделе 5.2).

Пример 5.2. На рис. 5.3 показано аннотированное дерево разбора для входной строки $3 * 5 + 4$ **n**, построенное с применением грамматики и правил, представленных на рис. 5.1. Предполагается, что значения *lexval* предоставляются лексическим анализатором. Каждый из узлов для нетерминалов имеет атрибут *val*,

[1]Не вдаваясь в детали, отметим, что хотя данная задача и разрешима, она не может быть решена алгоритмом с полиномиальным временем работы (даже если $\mathcal{P} = \mathcal{NP}$) в силу экспоненциальной сложности.

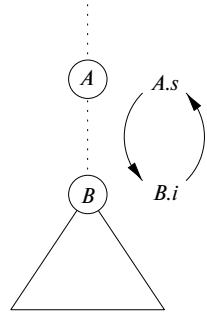

Рис. 5.2. Циклическая зависимость атрибутов $A.s$ и $B.i$ друг от друга

вычисляемый в восходящем порядке; на рисунке показаны значения атрибутов в каждом узле дерева разбора. Например, к узлу, среди дочерних узлов которого имеется узел, помеченный *, после вычисления значений атрибутов $T.val = 3$ и $F.val = 5$ в его первом и третьем дочернем узлах применяется правило, гласящее, что $T.val$ является произведением указанных значений, или 15. □

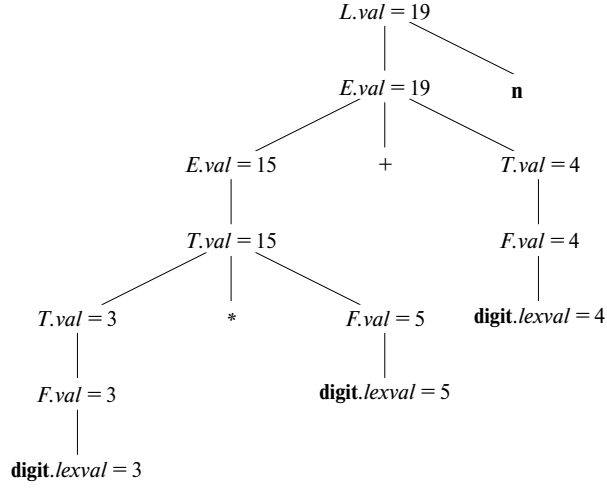

Рис. 5.3. Аннотированное дерево разбора для $3 * 5 + 4$ **n**

Наследуемые атрибуты полезны в случае, когда структура дерева разбора "не соответствует" абстрактному синтаксису исходного кода. Приведенный далее пример показывает, как наследуемые атрибуты могут использоваться для преодоления такого несоответствия, возникшего вследствие того, что грамматика разрабатывалась для синтаксического анализа, а не для трансляции.

Пример 5.3. СУО на рис. 5.4 вычисляет выражения наподобие $3 * 5$ и $3 * 5 * 7$. Нисходящий синтаксический анализ входной строки $3 * 5$ начинается с продукции $T \rightarrow F\ T'$. Здесь F генерирует цифру 3, но оператор * генерируется нетерминалом T'. Таким образом, левый операнд 3 находится в дереве разбора в другом поддереве, не в том, в котором находится оператор *. Следовательно, для передачи операнда оператору используются наследуемые атрибуты.

	Продукция	Семантические правила
1)	$T \rightarrow F\ T'$	$T'.inh = F.val$
		$T.val = T'.syn$
2)	$T' \rightarrow *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$
		$T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Рис. 5.4. СУО на основе грамматики, пригодной для нисходящего синтаксического анализа

Грамматика в этом примере представляет собой фрагмент нелеворекурсивной версии знакомой нам грамматики выражений; мы использовали такую грамматику в примере, иллюстрирующем нисходящий синтаксический анализ (см. раздел 4.4).

Каждый из нетерминалов T и F имеет синтезируемый атрибут *val*; терминал **digit** имеет синтезируемый атрибут *lexval*. Нетерминал T' имеет два атрибута: наследуемый атрибут *inh* и синтезируемый атрибут *syn*.

Семантические правила основаны на той идее, что левый операнд оператора * является наследуемым. Точнее, заголовок T' продукции $T' \rightarrow *\ F\ T'_1$ наследует левый операнд оператора * в теле продукции. У выражения $x * y * z$ корень поддерева для $*y * z$ наследует x. Далее, корень поддерева $*z$ наследует значение $x * y$, и так далее при наличии большего количества сомножителей в выражении. Когда все сомножители накоплены, результат передается назад по дереву с использованием синтезируемых атрибутов.

Чтобы увидеть, как используются семантические правила, рассмотрим аннотированное дерево разбора для выражения $3 * 5$ на рис. 5.5. Крайний слева лист дерева, помеченный **digit**, имеет значение атрибута $lexval = 3$, где 3 передается лексическим анализатором. Родительским узлом для него является узел продукции $F \rightarrow$ **digit**. Единственное семантическое правило, связанное с этой продукцией, определяет, что $F.val =$ **digit**$.lexval$, равному 3.

Во втором дочернем узле корня наследуемый атрибут $T'.inh$ определяется семантическим правилом $T'.inh = F.val$, связанным с продукцией 1. Таким образом,

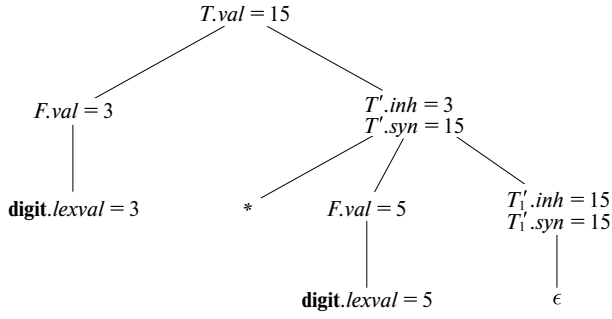

Рис. 5.5. Аннотированное дерево разбора для выражения $3 * 5$

левый операнд оператора * — 3 — передается слева направо между дочерними узлами корня.

Продукция в узле для $T' - T' \rightarrow * \; F \; T'_1$ (индекс 1 оставлен в аннотированном дереве разбора для того, чтобы различать два узла T'). Наследуемый атрибут $T'_1.inh$ определяется семантическим правилом $T'_1.inh = T'.inh \times F.val$, связанным с продукцией 2.

При $T'.inh = 3$ и $F.val = 5$ мы получаем $T'_1.inh = 15$. Узлу для T'_1 соответствует продукция $T' \rightarrow \epsilon$. Семантическое правило $T'.syn = T'.inh$ определяет значение $T'_1.syn = 15$. Атрибут *syn* в узлах для нетерминала T' передает значение 15 вверх по дереву до узла для нетерминала T, так что $T.val = 15$. □

5.1.3 Упражнения к разделу 5.1

Упражнение 5.1.1. Для СУО на рис. 5.1 постройте аннотированные деревья разбора для следующих выражений:

а) $(3 + 4) * (5 + 6)$ **n**;

б) $1 * 2 * 3 * (4 + 5)$ **n**;

в) $(9 + 8 * (7 + 6) + 5) * 4$ **n**.

Упражнение 5.1.2. Расширьте СУО на рис. 5.4 так, чтобы оно могло обрабатывать те же выражения, что и СУО на рис. 5.1.

Упражнение 5.1.3. Повторите упражнение 5.1.1 с использованием СУО, построенного вами при решении упражнения 5.1.2.

5.2 Порядок вычисления в СУО

Полезным инструментом для установления порядка вычисления атрибутов в конкретном дереве разбора является граф зависимостей. В то время как аннотированное дерево разбора показывает значения атрибутов, граф зависимостей помогает определить, каким образом эти значения могут быть вычислены.

В этом разделе в дополнение к графам зависимостей мы определим два важных класса СУО: S-атрибутные и более общие L-атрибутные СУО. Трансляции, определяемые этими двумя классами, хорошо согласуются с изучавшимися нами методами синтаксического анализа, и большинство трансляторов, встречающихся на практике, могут быть написаны так, чтобы соответствовать требованиям как минимум одного из упомянутых классов.

5.2.1 Графы зависимостей

Граф зависимостей (dependency graph) изображает поток информации между экземплярами атрибутов в определенном дереве разбора; ребро от одного экземпляра атрибута к другому означает, что значение первого атрибута необходимо для вычисления второго. Ребра выражают ограничения, следующие из семантических правил. Вот более детальное описание графа зависимостей.

- Для каждого узла дерева разбора, скажем, узла, помеченного грамматическим символом X, в графе зависимостей имеются узлы для каждого из атрибутов, связанных с X.

- Предположим, что семантическое правило, связанное с продукцией p, определяет значение синтезируемого атрибута $A.b$ через значение $X.c$ (правило может использовать для определения $A.b$ и другие атрибуты, помимо $X.c$). В таком случае в графе зависимостей имеется ребро от $X.c$ к $A.b$. Точнее, в каждом узле N, помеченном A, в котором применяется продукция p, создается ребро к атрибуту b в N от атрибута c в дочернем по отношению к N узле, соответствующем экземпляру символа X в теле продукции[2].

- Предположим, что семантическое правило, связанное с продукцией p, определяет значение наследуемого атрибута $B.c$ с использованием значения $X.a$. Тогда граф зависимостей содержит ребро от $X.a$ к $B.c$. Для каждого узла N, помеченного B, который соответствует появлению этого B в теле продукции p, создается ребро к атрибуту c в N от атрибута a в узле M, который соответствует данному вхождению X. Заметим, что M может быть родителем либо братом N.

[2]Поскольку узел N может иметь несколько дочерних узлов, помеченных X, считаем, что различать один и тот же символ в разных местах продукции позволяют их индексы.

Пример 5.4. Рассмотрим следующую продукцию и правило:

ПРОДУКЦИЯ	СЕМАНТИЧЕСКОЕ ПРАВИЛО
$E \to E_1 + T$	$E.val = E_1.val + T.val$

В каждом узле N, помеченном E, с дочерними узлами, соответствующими телу этой продукции, синтезируемый атрибут *val* в N вычисляется с использованием значений *val* в двух дочерних узлах, помеченных E и T. Таким образом, часть графа зависимостей для каждого дерева разбора, в котором используется эта продукция, выглядит так, как показано на рис. 5.6. По соглашению ребра дерева разбора будут изображаться пунктирной, а ребра графа зависимостей — сплошной линией. □

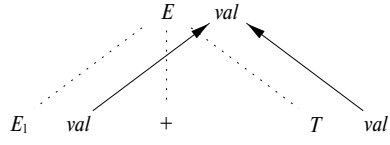

Рис. 5.6. $E.val$ синтезируется из $E_1.val$ и $T.val$

Пример 5.5. Пример полного графа зависимостей приведен на рис. 5.7. Узлы графа зависимостей, представленные числами от 1 до 9, соответствуют атрибутам в аннотированном дереве разбора на рис. 5.5.

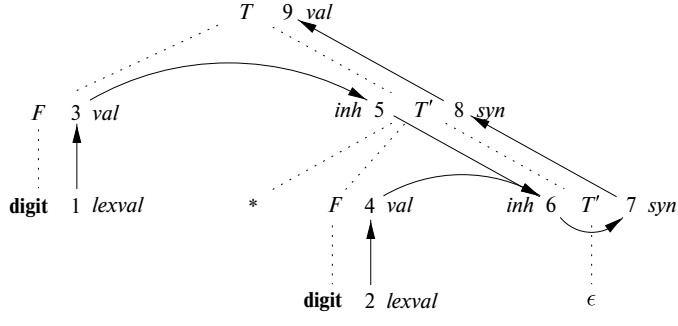

Рис. 5.7. Граф зависимостей для аннотированного дерева разбора, представленного на рис. 5.5

Узлы 1 и 2 представляют атрибут *lexval*, связанный с двумя листьями с метками **digit**. Узлы 3 и 4 представляют атрибут *val*, связанный с двумя узлами с метками F. Ребра в узел 3 из 1 и в узел 4 из 2 получаются из семантического правила, определяющего $F.val$ через **digit**.*lexval*. В действительности $F.val$ равно **digit**.*lexval*, но ребро представляет зависимость, а не равенство.

Узлы 5 и 6 представляют наследуемый атрибут $T'.inh$, связанный с каждым появлением нетерминала T'. Причиной наличия ребра из 3 в 5 служит правило $T'.inh = F.val$, определяющее $T'.inh$ в правом дочернем узле корня с использованием значения $F.val$ в левом дочернем узле. Имеются также ребра в узел 6 из узлов 5 (атрибут $T'.inh$) и 4 (атрибут $F.val$), поскольку указанные значения перемножаются для получения атрибута *inh* в узле 6.

Узлы 7 и 8 представляют синтезируемый атрибут *syn*, связанный с вхождениями T'. Ребро из узла 6 в узел 7 связано с семантическим правилом $T'.syn = T'.inh$, назначенным продукции 3 на рис. 5.4. Ребро из узла 7 в узел 8 связано с семантическим правилом, назначенным продукции 2.

Наконец, узел 9 представляет атрибут $T.val$. Ребро из узла 8 в узел 9 вызвано семантическим правилом $T.val = T'.syn$, связанным с продукцией 1. □

5.2.2 Упорядочение вычисления атрибутов

Граф зависимостей определяет возможные порядки вычисления атрибутов в различных узлах дерева разбора. Если граф зависимостей имеет ребро из узла M в узел N, то атрибут, соответствующий M, должен быть вычислен до атрибута N. Таким образом, допустимыми порядками вычисления атрибутов являются последовательности узлов $N_1, N_2, \ldots, N_k$, такие, что если в графе зависимостей имеется ребро от N_i к N_j, то $i < j$. Такое упорядочение графа зависимостей выстраивает его узлы в линейном порядке и называется *топологической сортировкой* графа.

Если в графе имеется цикл, топологическая сортировка такого графа невозможна, а значит, невозможно и вычисление СУО для данного дерева разбора. Если циклов нет, то существует как минимум одна топологическая сортировка. Чтобы понять, почему это так, убедимся, что в графе без циклов всегда имеется узел, в который не входит ни одно ребро. Если бы это было не так, то, переходя по входящим ребрам от предшественника к его предшественнику, мы бы в конечном счете попали в узел, в котором уже бывали, т.е. обнаружили бы цикл. Сделаем узел без входящих в него ребер первым в топологическом порядке, удалим его из графа зависимостей и повторим тот же процесс с оставшимися узлами.

Пример 5.6. Граф зависимостей на рис. 5.7 не имеет циклов. Одной из топологических сортировок является порядок, в котором пронумерованы узлы: $1, 2, \ldots, 9$. Заметим, что все ребра графа идут из узла с меньшим номером в узел с бóльшим номером, так что данный порядок определенно является топологической сортировкой. Имеются и другие топологические сортировки, например $1, 3, 5, 2, 4, 6, 7, 8, 9$. □

5.2.3 S-атрибутные определения

Как упоминалось ранее, для заданного СУО очень трудно сказать, существует ли дерево разбора, граф зависимостей которого содержит циклы. На практике трансляция может быть реализована с использованием классов СУО, для которых гарантированно существует порядок вычислений атрибутов, поскольку они не допускают наличия графов зависимостей с циклами. Два класса, рассматриваемых в этом разделе, могут быть эффективно реализованы в связи с нисходящим и восходящим синтаксическим анализом.

Первый из этих классов определяется следующим образом.

- СУО является *S-атрибутным*, если все его атрибуты синтезируемые.

Пример 5.7. СУО на рис. 5.1 является примером S-атрибутного определения. Каждый из атрибутов $L.val$, $E.val$, $T.val$ и $F.val$ является синтезируемым. □

Когда СУО является S-атрибутным, его атрибуты могут вычисляться в любом восходящем порядке узлов в дереве разбора. Зачастую особенно просто вычислить атрибуты путем обхода дерева разбора в обратном порядке и вычисления атрибутов в узле N, когда обход покидает узел последний раз, т.е. если применить определенную ниже функцию *postorder* к корню дерева разбора (см. также врезку "Обходы в прямом и обратном порядке" из раздела 2.3.4):

```
postorder (N) {
    for ( каждый дочерний узел C узла N, начиная слева) postorder (C);
    Вычислить атрибуты, связанные с узлом N;
}
```

S-атрибутные определения могут быть реализованы во время нисходящего синтаксического анализа, поскольку нисходящий синтаксический анализ соответствует обходу в обратном порядке. В частности, обратный порядок обхода в точности соответствует порядку, в котором LR-синтаксический анализатор сворачивает тела продукций в их заголовки. Этот факт будет использован в разделе 5.4.2 для вычисления синтезируемых атрибутов и сохранения их в стеке в процессе LR-синтаксического анализа без явного создания узлов дерева разбора.

5.2.4 L-атрибутные определения

Второй класс СУО называется *L-атрибутными определениями*. Идея, лежащая в основе этого класса, заключается в том, что ребра графа зависимостей между атрибутами, связанными с телом продукции, идут только слева направо, но не справа налево (отсюда и название — "L-атрибутный"). Точнее, каждый атрибут должен быть либо

1. синтезируемым, либо
2. наследуемым, но при выполнении определенных ограничивающих правил. Предположим, что имеется продукция $A \to X_1 X_2 \ldots X_n$ и что существует наследуемый атрибут $X_i.a$, вычисляемый при помощи правила, связанного с данной продукцией. Тогда это правило может использовать только
   а) наследуемые атрибуты, связанные с заголовком A;
   б) наследуемые либо синтезируемые атрибуты, связанные с вхождениями символов $X_1, X_2, \ldots X_{i-1}$, расположенных слева от X_i;
   в) наследуемые либо синтезируемые атрибуты, связанные с вхождениями самого X_i, но только таким образом, что в графе зависимостей, образованном атрибутами этого X_i, не имеется циклов.

Пример 5.8. СУО на рис. 5.4 является L-атрибутным. Чтобы увидеть, почему это так, рассмотрим семантические правила для наследуемых атрибутов, которые для удобства повторим еще раз:

Продукция	Семантическое правило
$T \to F\ T'$	$T'.inh = F.val$
$T' \to *F\ T'_1$	$T'_1.inh = T'.inh \times F.val$

Первое из этих правил определяет наследуемый атрибут $T'.inh$, использующий только $F.val$, причем, как и требуется, F находится в теле продукции слева от T'. Второе правило определяет $T'_1.inh$ с использованием наследуемого атрибута $T'.inh$, связанного с заголовком, и $F.val$, где F располагается слева от T'_1 в теле продукции.

В каждом из этих случаев правила используют информацию “сверху или слева”, как и требуется определением класса. Остальные атрибуты синтезируемые. Следовательно, рассмотренное СУО является L-атрибутным. □

Пример 5.9. Любое СУО, содержащее следующие продукции и правила, не может быть L-атрибутным:

Продукция	Семантические правила
$A \to B\ C$	$A.s = B.b$
	$B.i = f\,(C.c, A.s)$

Первое правило, $A.s = B.b$, является корректным как для S-атрибутного, так и для L-атрибутного СУО. Оно определяет синтезируемый атрибут $A.s$ с использованием атрибута в дочернем узле (т.е. символа в теле продукции).

Второе правило определяет наследуемый атрибут $B.i$, так что СУО, в целом, не может быть S-атрибутным. Далее, хотя правило и корректно, СУО не может

быть L-атрибутным, поскольку в определении $B.i$ участвует $C.c$, а C находится справа от B в теле продукции. Чтобы в L-атрибутном СУО могли использоваться атрибуты из братских узлов, они должны располагаться слева от символа, для которого определяется рассматриваемый атрибут. □

5.2.5 Семантические правила с контролируемыми побочными действиями

На практике трансляция включает побочные действия: настольный калькулятор может выводить результат вычисления, генератор кода может вносить тип идентификатора в таблицу символов. В случае СУО нас интересует баланс между атрибутными грамматиками и схемами трансляции. Атрибутные грамматики не имеют побочных действий и допускают любой порядок вычислений, согласующийся с графом зависимостей. Схемы трансляции требуют вычислений слева направо и допускают семантические действия, содержащие произвольные программные фрагменты; схемы трансляции рассматриваются в разделе 5.4.

Мы будем контролировать побочные действия в СУО одним из следующих способов.

- Позволяя второстепенные побочные действия, не препятствующие вычислению атрибутов. Другими словами, побочные действия разрешаются, если вычисление атрибутов на основе любой топологической сортировки графа зависимостей дает "корректную" трансляцию, где "корректность" зависит от конкретного приложения.

- Ограничивая допустимые порядки вычисления так, что при любом из разрешенных порядков вычисления получается одна и та же трансляция. Ограничения могут рассматриваться как неявные ребра, добавленные к графу зависимостей.

В качестве примера второстепенного побочного действия изменим калькулятор из примера 5.1 так, чтобы он выводил полученный результат. Вместо правила $L.val = E.val$, которое сохраняет результат вычислений в синтезируемом атрибуте $L.val$, рассмотрим следующее правило:

	Продукция	Семантическое правило
1)	$L \rightarrow E$ **n**	$print\,(E.val)$

Семантические правила, выполнение которых сводится только к побочным действиям, будут рассматриваться как определения фиктивных синтезируемых атрибутов, связанных с заголовком продукции. Данное модифицированное СУО дает ту же трансляцию при любой топологической сортировке, поскольку инструкция вывода выполняется в самом конце, после того как результат вычислений сохранен в $E.val$.

Пример 5.10. СУО на рис. 5.8 получает простое объявление D, состоящее из базового типа T, за которым следует список идентификаторов L. T может быть **int** или **float**. Для каждого идентификатора в списке тип вносится в запись таблицы символов для данного идентификатора. Мы считаем, что внесение типа в запись таблицы символов для данного идентификатора не влияет на записи таблицы символов для прочих идентификаторов. Таким образом, записи могут обновляться в произвольном порядке. Это СУО не проверяет, не объявлен ли идентификатор более одного раза; для этого в СУО можно внести необходимые изменения.

	Продукция	Семантические правила
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to$ **int**	$T.type =$ integer
3)	$T \to$ **float**	$T.type =$ float
4)	$L \to L_1$, **id**	$L_1.inh = L.inh$
		addType (**id**.*entry*, $L.inh$)
5)	$L \to$ **id**	*addType* (**id**.*entry*, $L.inh$)

Рис. 5.8. Синтаксически управляемое определение для простых объявлений типов

Нетерминал D представляет объявление, которое в соответствии с продукцией 1 состоит из типа T, за которым следует список идентификаторов L. T имеет один атрибут, $T.type$, который является типом объявления D. Нетерминал L также имеет один атрибут, который назван *inh* для того, чтобы подчеркнуть, что этот атрибут — наследуемый. Назначение $L.inh$ — передача объявленного типа вниз по списку идентификаторов, чтобы он мог быть добавлен в соответствующие записи таблицы символов.

Каждая из продукций 2 и 3 вычисляет синтезируемый атрибут $T.type$, присваивая ему корректное значение — integer или float. Этот тип передается атрибуту $L.inh$ в правиле для продукции 1. Продукция 4 передает $L.inh$ вниз по дереву разбора, т.е. значение $L_1.inh$ вычисляется в узле дерева разбора путем копирования значения $L.inh$ из родительского по отношению к данному узлу; родительский узел соответствует заголовку продукции.

Продукции 4 и 5 имеют также правило, в соответствии с которым вызывается функция *addType* с двумя аргументами:

1. **id**.*entry*, лексическим значением, которое указывает на объект таблицы символов;

2. $L.inh$, типом, назначенным каждому идентификатору из списка.

Мы предполагаем, что функция *addType* корректно устанавливает тип $L.inh$ в качестве типа представленного идентификатора.

Граф зависимостей для входной строки **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$ показан на рис. 5.9. Числа от 1 до 10 представляют узлы графа зависимостей. Узлы 1, 2 и 3 представляют атрибут *entry*, связанный с каждым из листьев с метками **id**. Узлы 6, 8 и 10 являются фиктивными атрибутами, представляющими применение функции *addType* к типу и одному из упомянутых значений *entry*.

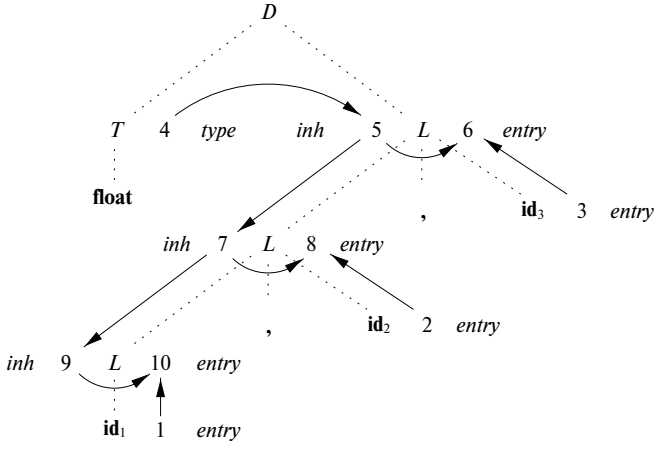

Рис. 5.9. Граф зависимостей для объявления **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Узел 4 представляет атрибут $T.type$, и здесь действительно начинается вычисление атрибута. Этот тип затем передается узлам 5, 7 и 9, представляющим атрибут $L.inh$, связанный с каждым появлением нетерминала L. □

5.2.6 Упражнения к разделу 5.2

Упражнение 5.2.1. Приведите все возможные топологические сортировки для графа зависимостей на рис. 5.7.

Упражнение 5.2.2. Изобразите аннотированные деревья разбора для СУО на рис. 5.8 для следующих выражений:

а) `int a, b, c;`

б) `float w, x, y, z.`

Упражнение 5.2.3. Предположим, что у нас есть продукция $A \rightarrow B\ C\ D$. Каждый из нетерминалов A, B, C и D имеет два атрибута: синтезируемый атрибут s и наследуемый атрибут i. Для каждого из перечисленных ниже наборов правил

укажите, согласуются ли эти правила с S- и L-атрибутным определениями и существует ли вообще какой-либо порядок вычисления атрибутов в соответствии с указанными правилами.

а) $A.s = B.i + C.s$

б) $A.s = B.i + C.s$ и $D.i = A.i + B.s$

в) $A.s = B.s + D.s$

! г) $A.s = D.i$, $B.i = A.s + C.s$, $C.i = B.s$ и $D.i = B.i + C.i$

! **Упражнение 5.2.4.** Приведенная грамматика генерирует бинарные числа с "десятичной" точкой:

$$\begin{array}{lcl} S & \rightarrow & L \,.\, L \mid L \\ L & \rightarrow & L\, B \mid B \\ B & \rightarrow & 0 \mid 1 \end{array}$$

Разработайте L-атрибутное СУО для вычисления $S.val$, десятичного значения входной строки. Например, трансляция строки `101.101` должна давать десятичное число 5.625. *Указание*: воспользуйтесь наследуемым атрибутом $L.inh$, который говорит, с какой стороны от десятичной точки располагается бит.

!! **Упражнение 5.2.5.** Разработайте S-атрибутное СУО для грамматики и трансляции, описанных в упражнении 5.2.4.

!! **Упражнение 5.2.6.** Реализуйте алгоритм 3.23, который преобразует регулярные выражения в недетерминированные конечные автоматы, как L-атрибутное СУО для грамматики, к которой применим нисходящий синтаксический анализ. Считаем, что токен **char** представляет произвольный символ, а **char**.*lexval* — символ, который он представляет. Вы можете также считать, что существует функция *new* (), которая возвращает новое состояние, т.е. состояние, которое ранее этой функцией не возвращалось. Используйте любые подходящие обозначения для указания переходов НКА.

5.3 Применения синтаксически управляемой трансляции

Синтаксически управляемые методы из данной главы будут применены в главе 6 к проверке типов и генерации промежуточного кода. Здесь же мы рассмотрим избранные примеры, иллюстрирующие некоторые представительные СУО.

Основное применение в этом разделе заключается в построении синтаксических деревьев. Поскольку некоторые компиляторы в качестве промежуточного представления используют синтаксические деревья, распространенным видом

СУО является преобразование входной строки в дерево. Для завершения трансляции в промежуточный код компилятор затем может обойти построенное синтаксическое дерево, используя другой набор правил. (В главе 6 рассматривается подход к построению промежуточного кода, при котором СУО применяется без явного построения дерева.)

Мы рассмотрим два СУО для построения синтаксических деревьев для выражений. Первое S-атрибутное определение может использоваться в процессе восходящего синтаксического анализа. Второе L-атрибутное определение подходит для использования во время нисходящего синтаксического анализа.

Заключительный пример в этой главе представляет собой L-атрибутное определение, работающее с базовыми типами и массивами.

5.3.1 Построение синтаксических деревьев

Как говорилось в разделе 2.8.2, каждый узел синтаксического дерева представляет конструкцию; его дочерние узлы представляют значащие компоненты конструкции. Синтаксическое дерево, представляющее выражение $E_1 + E_2$, имеет метку + и два дочерних узла, представляющих подвыражения E_1 и E_2.

Мы реализуем узлы синтаксического дерева при помощи объектов с соответствующим количеством полей. Каждый объект будет иметь поле *op*, являющееся меткой узла. У объектов будут следующие дополнительные поля.

- Если узел является листом, дополнительное поле хранит лексическое значение листа. Конструктор $Leaf(op, val)$ создает объект листа. В качестве альтернативы, если узлы рассматриваются как записи, *Leaf* возвращает указатель на новую запись для листа.

- Если узел — внутренний, то в нем имеется столько дополнительных полей, сколько у него дочерних узлов в синтаксическом дереве. Конструктор *Node* получает два или больше аргументов: $Node(op, c_1, c_2, \ldots, c_k)$ создает объект с первым полем *op* и k дополнительными полями для k дочерних узлов $c_1, \ldots, c_k$.

Пример 5.11. S-атрибутное определение на рис. 5.10 строит синтаксическое дерево для простой грамматики выражений, включающей только два бинарных оператора: + и −. Как обычно, эти операторы имеют один и тот же приоритет и совместно левоассоциативны. Все нетерминалы имеют один синтезируемый атрибут *node*, который представляет узел синтаксического дерева.

Каждый раз при использовании первой продукции, $E \to E_1 + T$, ее правило создает узел с '+' в качестве *op* и двумя дочерними узлами $E_1.node$ и $T.node$ для подвыражений. Вторая продукция имеет похожее правило.

В случае продукции 3, $E \to T$, узел не создается, поскольку $E.node$ имеет то же значение, что и $T.node$. Аналогично не создается узел и при использовании

	Продукция	Семантическое правило
1)	$E \to E_1 + T$	$E.node =$ **new** *Node* $('+', E_1.node, T.node)$
2)	$E \to E_1 - T$	$E.node =$ **new** *Node* $('-', E_1.node, T.node)$
3)	$E \to T$	$E.node = T.node$
4)	$T \to (E)$	$T.node = E.node$
5)	$T \to$ **id**	$T.node =$ **new** *Leaf* $($**id**, **id**.*entry*$)$
6)	$T \to$ **num**	$T.node =$ **new** *Leaf* $($**num**, **num**.*val*$)$

Рис. 5.10. Построение синтаксических деревьев для простых выражений

продукции 4, $T \to (E)$. Значение $T.node$ то же, что и значение $E.node$, поскольку скобки используются для группирования; они влияют на структуру дерева разбора и синтаксического дерева, но после выполнения своих функций их присутствие в синтаксическом дереве не является необходимым.

Последние две T-продукции имеют справа один терминал. Мы используем конструктор *Leaf* для создания соответствующего узла, который становится значением $T.node$.

На рис. 5.11 показано построение синтаксического дерева для выражения $a - 4 + c$. Узлы синтаксического дерева показаны как записи с первым полем *op*. Здесь ребра синтаксического дерева показаны сплошными линиями, а лежащее в основе дерево разбора, построение которого в явном виде не требуется, — линиями, состоящими из точек. Третий вид линий — пунктирные — представляет значения $E.node$ и $T.node$; каждая линия указывает на соответствующий узел синтаксического дерева.

Внизу находятся листья для a, 4 и c, построенные с применением конструктора *Leaf*. Считаем, что лексическое значение **id**.*entry* указывает на запись в таблице символов, а лексическое значение **num**.*val* представляет собой числовую константу. Эти листья, или указатели на них, становятся значениями $T.node$ трех узлов дерева разбора, помеченных T, согласно правилам 5 и 6. Заметим, что в соответствии с правилом 3 указатель на лист a является также значением $E.node$ для крайнего слева E в дереве разбора.

Правило 2 заставляет создать узел с *op*, равным знаку −, и указателями на два первых листа. Затем правило 1 создает корневой узел синтаксического дерева, объединяя узел для − с третьим листом.

Если правила вычисляются в процессе обратного обхода дерева разбора или при свертках в процессе восходящего синтаксического анализа, то последовательность шагов, показанная на рис. 5.12, заканчивается указателем p_5, указывающим на корень построенного синтаксического дерева. □

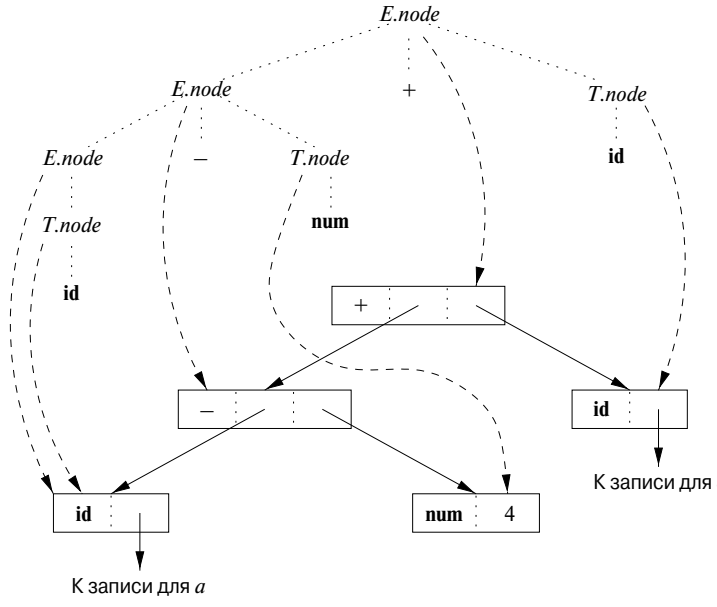

Рис. 5.11. Синтаксическое дерево для $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}a)$;
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4)$;
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2)$;
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry}\text{-}c)$;
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4)$;

Рис. 5.12. Шаги построения синтаксического дерева для $a - 4 + c$

В случае грамматики для нисходящего синтаксического анализа строятся те же синтаксические деревья с использованием той же последовательности шагов, несмотря на то что структура деревьев разбора существенно отличается от структуры синтаксических деревьев.

Пример 5.12. L-атрибутное определение на рис. 5.13 выполняет ту же трансляцию, что и S-атрибутное определение на рис. 5.10. Атрибуты для грамматических символов E, T, **id** и **num** рассмотрены в примере 5.11.

Правила для построения синтаксических деревьев в этом примере подобны правилам для настольного калькулятора из примера 5.3. В примере с калькулятором член $x * y$ вычислялся путем передачи x как наследуемого атрибута, поскольку x и $*y$ находятся в разных частях дерева разбора. Здесь идея состоит в том, чтобы построить синтаксическое дерево для $x + y$, передавая x в качестве насле-

	Продукция	Семантические правила
1)	$E \to T\ E'$	$E.node = E'.syn$
		$E'.inh = T.node$
2)	$E' \to +\ T\ E'_1$	$E'_1.inh =$ **new** $Node\,('+', E'.inh, T.node)$
		$E'.syn = E'_1.syn$
3)	$E' \to -\ T\ E'_1$	$E'_1.inh =$ **new** $Node\,('-', E'.inh, T.node)$
		$E'.syn = E'_1.syn$
4)	$E' \to \epsilon$	$E'.syn = E'.inh$
5)	$T \to (E)$	$T.node = E.node$
6)	$T \to$ **id**	$T.node =$ **new** $Leaf\,($**id**$,$ **id**$.entry)$
7)	$T \to$ **num**	$T.node =$ **new** $Leaf\,($**num**$,$ **num**$.val)$

Рис. 5.13. Построение синтаксических деревьев в процессе нисходящего синтаксического анализа

дуемого атрибута, поскольку x и $+y$ находятся в разных поддеревьях. Нетерминал E' является копией нетерминала T' из примера 5.3. Сравните граф зависимостей для $a - 4 + c$ на рис. 5.14 с графом зависимостей для $3 * 5$ на рис. 5.7.

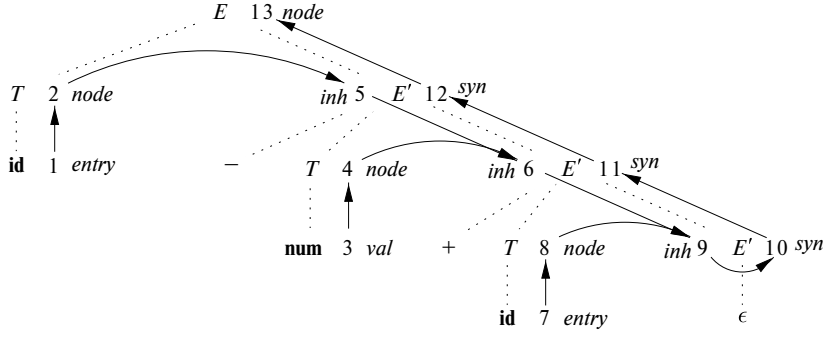

Рис. 5.14. Граф зависимостей для выражения $a - 4 + c$ при использовании СУО на рис. 5.13

Нетерминал E' имеет наследуемый атрибут *inh* и синтезируемый атрибут *syn*. Атрибут $E'.inh$ представляет неполное синтаксическое дерево, построенное к этому моменту. А именно, он представляет корень дерева для префикса входной строки, находящегося слева от поддерева для E'. В узле 5 графа зависимостей на рис. 5.14 $E'.inh$ обозначает корень неполного синтаксического дерева для идентификатора a, т.е. попросту лист для a. В узле 6 $E'.inh$ обозначает корень неполного

синтаксического дерева для входной строки $a - 4$. В узле 9 $E'.inh$ обозначает корень синтаксического дерева для $a - 4 + c$.

Поскольку на этом входная строка заканчивается, в узле 9 $E'.inh$ указывает на корень всего синтаксического дерева. Атрибут *syn* передает это значение обратно вверх по дереву разбора, пока оно не становится значением $E.node$. Точнее, значение атрибута в узле 10 определяется правилом $E'.syn = E'.inh$, связанным с продукцией $E' \to \epsilon$. Значение атрибута в узле 11 определяется правилом $E'.syn = E'_1.syn$, связанным с продукцией 2 на рис. 5.13. Аналогичные правила определяют значения атрибутов в узлах 12 и 13. □

5.3.2 Структура типа

Наследуемые атрибуты полезны, когда структура дерева разбора отличается от абстрактного синтаксиса входной строки; тогда атрибуты могут использоваться для передачи информации из одной части дерева разбора в другую. Приведенный далее пример показывает, что несоответствие структуры может являться следствием дизайна языка, но не ограничений, накладываемых методом синтаксического анализа.

Пример 5.13. В C тип **int** [2] [3] можно прочесть как "массив из двух массивов по 3 целых числа". Соответствующее выражение типа $array\,(2, array\,(3, integer))$ представлено деревом на рис. 5.15. Оператор *array* принимает два параметра, число и тип. Если типы представлены деревьями, то этот оператор возвращает дерево с меткой *array* с двумя дочерними узлами — для числа и для типа.

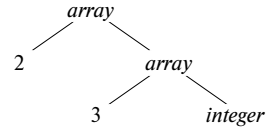

Рис. 5.15. Выражение типа для **int** [2] [3]

При использовании СУО, представленного на рис. 5.16, нетерминал T генерирует либо фундаментальный тип, либо тип массива. Нетерминал B генерирует один из базовых типов: **int** или **float**. T генерирует фундаментальный тип, если T порождает $B\ C$, а C порождает ϵ. В противном случае C генерирует компоненты массива, состоящие из последовательности целых чисел, каждое из которых заключено в квадратные скобки.

Нетерминалы B и T имеют синтезируемый атрибут t, представляющий тип. Нетерминал C имеет два атрибута: наследуемый атрибут b и синтезируемый атрибут t. Наследуемые атрибуты b передают фундаментальный тип вниз по дереву, а синтезируемые атрибуты t накапливают результат.

Продукция	Семантические правила
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t =$ *integer*
$B \rightarrow$ **float**	$B.t =$ *float*
$C \rightarrow [$**num**$]\ C_1$	$C.t =$ *array* $(\mathbf{num}.val, C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Рис. 5.16. T генерирует либо фундаментальный тип, либо тип массива

Аннотированное дерево разбора для входной строки **int** $[2]\,[3]$ показано на рис. 5.17. Соответствующее выражение типа на рис. 5.15 построено путем передачи типа *integer* от B вниз по цепочке C посредством наследуемых атрибутов b. Тип массива синтезируется при продвижении вверх по цепочке C посредством атрибутов t.

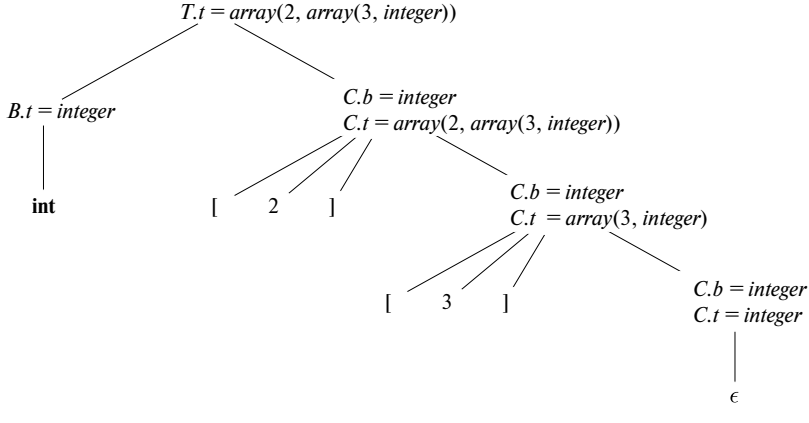

Рис. 5.17. Синтаксически управляемая трансляция типов массивов

Говоря более подробно, в корне для $T \rightarrow B\ C$ нетерминал C наследует тип от B при помощи наследуемого атрибута $C.b$. В крайнем справа узле для C используется продукция $C \rightarrow \epsilon$, так что $C.t$ равно $C.b$. Семантические правила для продукции $C \rightarrow [$**num**$]\ C_1$ образуют $C.t$ путем применения оператора *array* к операндам **num**.*val* и $C_1.t$. □

5.3.3 Упражнения к разделу 5.3

Упражнение 5.3.1. Ниже приведена грамматика для выражений, включающих оператор + и операнды, представляющие собой целые числа либо числа с плавающей точкой (числа с плавающей точкой распознаются по наличию десятичной точки):

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow \textbf{num . num} \mid \textbf{num} \end{aligned}$$

а) Разработайте СУО для определения типа каждого члена T и выражения E.

б) Распространите свое СУО из п. *а* на выражения в постфиксной записи. Воспользуйтесь унарным оператором **intToFloat** для преобразования целого числа в эквивалентное число с плавающей точкой.

! **Упражнение 5.3.2.** Разработайте СУО для трансляции инфиксных выражений с + и * в эквивалентные выражения без излишних скобок. Например, поскольку оба оператора левоассоциативны, а приоритет * выше приоритета +, $((a*(b+c))*(d))$ транслируется в $a*(b+c)*d$.

! **Упражнение 5.3.3.** Разработайте СУО для дифференцирования выражений наподобие $x*(3*x+x*x)$, которые включают операторы + и *, переменную x и константы. Будем считать, что никакие упрощения не выполняются, т.е. $3*x$, например, транслируется в $3*1+0*x$.

5.4 Синтаксически управляемые схемы трансляции

Синтаксически управляемые схемы трансляции представляют собой запись, дополняющую синтаксически управляемые определения. Все применения синтаксически управляемых определений в разделе 5.3 могут быть реализованы с использованием синтаксически управляемых схем трансляции.

Из раздела 2.3.5 мы знаем, что *синтаксически управляемая схема трансляции* (СУТ) представляет собой контекстно-свободную грамматику с программными фрагментами, внедренными в тела продукций. Эти фрагменты называются *семантическими действиями* и могут находиться в любой позиции в теле продукции. По соглашению действия располагаются внутри фигурных скобок; фигурные скобки, являющиеся грамматическими символами, заключаются в кавычки.

Любая СУТ может быть реализована путем построения дерева разбора с последующим выполнением действий в порядке в глубину слева направо, т.е. в порядке прямого обхода дерева. Соответствующий пример приведен в разделе 5.4.3.

Обычно СУТ реализуется в процессе синтаксического анализа, без построения дерева разбора. В данном разделе мы остановимся на использовании СУТ для реализации двух важных классов СУО.

1. Грамматика поддается LR-синтаксическому анализу, а СУО — S-атрибутное.

2. Грамматика поддается LL-синтаксическому анализу, а СУО — L-атрибутное.

Мы увидим, каким образом в обоих случаях семантические правила СУО могут быть преобразованы в СУТ с действиями, выполняемыми в нужный момент. В процессе синтаксического анализа действие в теле продукции выполняется, как только все грамматические символы слева от него сопоставлены входной строке.

СУТ, которые могут быть реализованы в процессе синтаксического анализа, можно охарактеризовать путем вставки вместо действий отличающихся друг от друга нетерминалов-маркеров; каждый маркер M имеет единственную продукцию $M \to \epsilon$. Если синтаксический анализ грамматики с такими нетерминалами-маркерами может быть выполнен некоторым методом, то СУТ может быть реализована в процессе данного синтаксического анализа.

5.4.1 Постфиксные схемы трансляции

Простейшая реализация СУТ имеет место в случае восходящего синтаксического анализа и S-атрибутного СУО. В этом случае можно построить СУТ, в которой каждое действие размещается в конце продукции и выполняется вместе со сверткой тела продукции в заголовок. СУТ со всеми действиями, расположенными на правом конце тел продукций, называются *постфиксными СУТ*.

Пример 5.14. Постфиксная СУТ на рис. 5.18 реализует СУО настольного калькулятора, представленного на рис. 5.1, с единственным изменением: действие первой продукции выводит вычисленное значение. Поскольку лежащая в основе СУТ грамматика является LR-грамматикой, а СУО — S-атрибутное, эти действия могут корректно выполняться синтаксическим анализатором при свертках. □

5.4.2 Реализация постфиксной СУТ с использованием стека синтаксического анализатора

Постфиксная СУТ может быть реализована в процессе LR-синтаксического анализа путем выполнения действий при свертках. Атрибут (или атрибуты) каждого грамматического символа может помещаться в стек в том месте, где он может быть обнаружен в процессе свертки. Лучше всего разместить атрибуты вместе с грамматическими символами (или LR-состояниями, представляющими эти символы) в записях стека.

$$
\begin{array}{ll}
L \to E\ \mathbf{n} & \{print\,(E.val)\,;\} \\
E \to E_1 + T & \{E.val = E_1.val + T.val;\} \\
E \to T & \{E.val = T.val;\} \\
T \to T_1 * F & \{T.val = T_1.val \times F.val;\} \\
T \to F & \{T.val = F.val;\} \\
F \to (E) & \{F.val = E.val;\} \\
F \to \mathbf{digit} & \{F.val = \mathbf{digit}.lexval;\}
\end{array}
$$

Рис. 5.18. Постфиксная СУТ, реализующая настольный калькулятор

На рис. 5.19 стек синтаксического анализатора содержит записи с полем для грамматических символов (или состояний синтаксического анализатора), а под ним — поле для атрибутов. На вершине стека находятся три грамматических символа — $X\ Y\ Z$; возможно, они будут свернуты в соответствии с продукцией наподобие $A \to X\ Y\ Z$. На рисунке показан атрибут $X.x$ грамматического символа X и т.д. В общем случае могут иметься несколько атрибутов; в этом случае следует либо увеличить размер записи в стеке так, чтобы она могла содержать все атрибуты, либо поместить в стек указатели на соответствующие записи. В случае атрибутов небольшого размера и в небольшом количестве может оказаться проще увеличить размер записи в стеке, даже если некоторые поля будут время от времени пустовать. Но если один или несколько атрибутов имеют неограниченный размер — например, представляют собой строки, — то лучше помещать в стек указатель на значение атрибута, который хранится в некотором другом месте памяти, не являющейся частью стека.

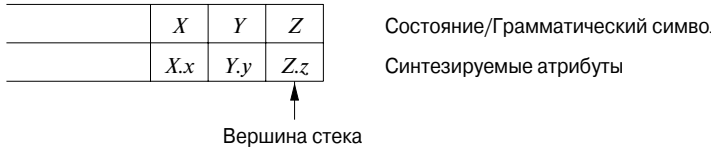

Рис. 5.19. Стек синтаксического анализатора с полем для синтезируемых атрибутов

Если все атрибуты синтезируемые, а действия располагаются в конце продукций, то можно вычислять атрибуты заголовков при выполнении свертки тела продукции к заголовку. Если выполняется свертка в соответствии с продукцией, такой как $A \to X\ Y\ Z$, то все атрибуты X, Y и Z оказываются доступны и находятся в известных позициях в стеке, как показано на рис. 5.19. После выполнения действия A и его атрибуты находятся на вершине стека, в позиции, которую занимала запись для X.

Пример 5.15. Перепишем действия из СУТ настольного калькулятора из примера 5.14 так, чтобы они явно работали со стеком. Такая работа со стеком обычно выполняется синтаксическим анализатором автоматически.

Предположим, что стек поддерживается в виде массива записей с именем *stack*, с курсором *top*, указывающим на вершину стека. Таким образом, $stack\,[top]$ представляет собой запись на вершине стека, $stack\,[top-1]$ — запись под ней и т.д. Предположим также, что каждая запись имеет поле с именем *val*, в котором хранится атрибут грамматического символа, представленного данной записью. Таким образом, обратиться к атрибуту $E.val$, находящемуся в третьей позиции стека, можно как к $stack\,[top-2]\,.val$. Полностью СУТ показана на рис. 5.20.

Продукция	Действия
$L \to E$ **n**	{ print($stack\,[top-1].val$); $top = top - 1$; }
$E \to E_1 + T$	{ $stack\,[top-2].val = stack\,[top-2].val + stack\,[top].val$; $top = top - 2$; }
$E \to T$	
$T \to T_1 * F$	{ $stack\,[top-2].val = stack\,[top-2].val \times stack\,[top].val$; $top = top - 2$; }
$T \to F$	
$F \to (\ E\ )$	{ $stack\,[top-2].val = stack\,[top-1].val$; $top = top - 2$; }
$F \to$ **digit**	

Рис. 5.20. Реализация настольного калькулятора в стеке восходящего синтаксического анализа

Например, во второй продукции, $E \to E_1 + T$, значение E_1 расположено на две позиции ниже вершины стека, в то время как значение T находится на вершине стека. Вычисленная сумма помещается в то место, в котором после свертки находится заголовок E, т.е. двумя позициями ниже текущей вершины стека. Причина этого в том, что после свертки три грамматических символа на вершине стека будут заменены одним. После вычисления $E.val$ со стека снимаются два символа, так что запись, в которую было помещено значение $E.val$, после этого окажется на вершине стека.

В третьей продукции $E \to T$ действие не является необходимым, поскольку размер стека не изменяется и значение атрибута $T.val$ на вершине стека просто станет значением $E.val$. Те же самые рассуждения применимы и к продукциям

$T \to F$ и $F \to$ **digit**. Продукция $F \to (E)$ несколько отличается тем, что, хотя значение атрибута и не изменяется, при свертке со стека снимаются два элемента, так что значение атрибута надо перенести в позицию, которая окажется на вершине стека после свертки.

Заметим, что здесь опущены шаги, управляющие первым полем записей в стеке — полем, которое содержит LR-состояние или представляет грамматический символ. При выполнении LR-синтаксического анализа таблица анализа говорит нам о том, в какое новое состояние будет выполнена свертка (см. алгоритм 4.44). Таким образом, можно просто поместить состояние в запись на новой вершине стека. □

5.4.3 СУТ с действиями внутри продукций

Действия могут находиться в любой позиции в теле продукции. Действие выполняется сразу же после того, как обработаны все символы слева от него. Таким образом, если у нас есть продукция $B \to X\ \{a\}\ Y$, то действие a выполняется после того, как распознан X (если X — терминал) или все терминалы, порожденные из X (если X — нетерминал). Говоря более точно,

- при восходящем синтаксическом анализе действие a выполняется, как только данный грамматический символ X оказывается на вершине стека;
- при нисходящем синтаксическом анализе действие a выполняется непосредственно перед тем, как выполняется раскрытие данного Y (если Y — нетерминал) или проверяется его наличие во входной строке (если Y — терминал).

СУТ, которые могут быть реализованы в процессе синтаксического анализа, включают постфиксные СУТ и класс СУТ, рассматриваемый в разделе 5.5, который реализует L-атрибутные определения. Как вы узнаете из приведенного ниже примера, не все СУТ могут быть реализованы в процессе синтаксического анализа.

Пример 5.16. В качестве крайнего примера проблемной СУТ предположим, что мы преобразовали наш калькулятор в СУТ, которая печатает выражения в префиксном виде, а не вычисляет их значения. Продукции и действия такой СУТ показаны на рис. 5.21.

К сожалению, эту СУТ невозможно реализовать в процессе нисходящего или восходящего синтаксического анализа, поскольку синтаксический анализатор должен выполнять критические действия, такие как вывод экземпляра * или +, задолго до того, как станет известно о его наличии во входном потоке.

Использование нетерминалов-маркеров M_2 и M_4 для действий в продукциях соответственно 2 и 4 приводит к тому, что ПС-синтаксический анализатор (см.

1) $L \rightarrow E$ **n**
2) $K \rightarrow \{print('+');\}\ E_1 + T$
3) $E \rightarrow T$
4) $T \rightarrow \{print('*');\}\ T_1 * F$
5) $T \rightarrow F$
6) $F \rightarrow (E)$
7) $F \rightarrow$ **digit** $\{print($**digit**$.lexval);\}$

Рис. 5.21. Проблемная СУТ для трансляции инфиксных выражений в префиксные в процессе синтаксического анализа

раздел 4.5.3) при входном символе, например, 3 обнаруживает конфликт между сверткой согласно $M_2 \rightarrow \epsilon$, сверткой согласно $M_4 \rightarrow \epsilon$ и переносом обнаруженной во входном потоке цифры. □

Любая СУТ может быть реализована следующим образом.

1. Игнорируя действия, выполняется синтаксический анализ входного потока и строится дерево разбора.

2. Проверяется каждый внутренний узел N, скажем, для продукции $A \rightarrow \alpha$. Добавляем к N дополнительные дочерние узлы для действий из α таким образом, что дочерние узлы N слева направо составляют символы и действия α.

3. Выполняется обход дерева в прямом порядке (см. раздел 2.3.4) и при посещении узла, помеченного действием, выполняется это действие.

Например, на рис. 5.22 показано дерево разбора со вставленными действиями для выражения $3*5+4$. При посещении узлов дерева в прямом порядке получается префиксная запись выражения: $+ * 3\ 5\ 4$.

5.4.4 Устранение левой рекурсии из СУТ

Поскольку ни одна грамматика с левой рекурсией не может быть детерминированно проанализирована нисходящим синтаксическим анализатором, нам надо устранять из грамматики левую рекурсию (с этим действием мы уже встречались в разделе 4.3.3). Когда грамматика является частью СУТ, при устранении левой рекурсии необходимо побеспокоиться также и о действиях.

Начнем с рассмотрения простого случая, в котором нас интересует только порядок выполнения действий в СУТ. Например, если каждое действие состоит в печати строки, то нас интересует только порядок, в котором будут напечатаны эти строки. В этом случае можно руководствоваться следующим принципом.

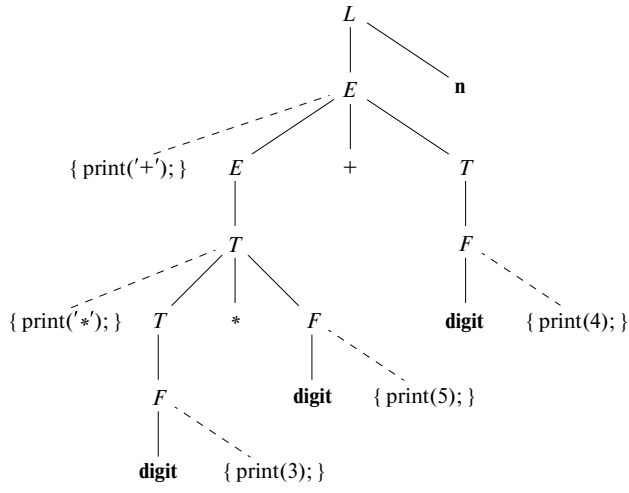

Рис. 5.22. Дерево разбора с добавленными действиями

- При внесении изменений в грамматику следует рассматривать действия так, как если бы они были терминальными символами.

Этот принцип основан на той идее, что преобразования грамматики сохраняют порядок терминалов в генерируемой строке. Действия, таким образом, выполняются в одном и том же порядке при любом синтаксическом анализе слева направо, как нисходящем, так и восходящем.

"Трюк" устранения левой рекурсии состоит в том, что мы берем продукции

$$A \quad \rightarrow \quad A\alpha \mid \beta$$

Они генерируют строки, состоящие из β и произвольного количества α, и заменяем их продукциями, которые генерируют те же строки с использованием нового нетерминала R:

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

Если β не начинается с A, то A больше не имеет леворекурсивной продукции. В терминах регулярных определений в обоих множествах продукций A определяется как $\beta\,(\alpha)^*$. Как справиться с ситуациями, когда A имеет больше рекурсивных или нерекурсивных продукций, описано в разделе 4.3.3.

Пример 5.17. Рассмотрим следующие E-продукции из СУТ для преобразования инфиксных выражений в постфиксную запись:

$$\begin{aligned} E &\rightarrow E_1 + T \quad \{\mathit{print}\,('+')\,;\} \\ E &\rightarrow T \end{aligned}$$

Если мы применим стандартное преобразование к E, то остатком леворекурсивной продукции будет

$$\alpha = +T \quad \{print\,('+')\,;\}$$

А β, тело другой продукции, — T. Если мы введем R как остаток E, то получим следующее множество продукций:

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & +T\ \{print\,('+')\,;\} \\ R & \rightarrow & \epsilon \end{array}$$

□

Если же действия СУО не просто выводят строки, но и вычисляют атрибуты, то следует быть более осторожным при устранении левой рекурсии из грамматики. Однако, если СУО является S-атрибутным, то мы всегда можем построить СУТ путем размещения действий по вычислению атрибутов в соответствующих позициях в новых продукциях.

Приведем общую схему для случая одной рекурсивной продукции, одной нерекурсивной продукции и одного атрибута леворекурсивного нетерминала; обобщение на случай многих продукций каждого типа не сложное, но весьма громоздкое. Предположим, что эти две продукции —

$$\begin{array}{lcl} A & \rightarrow & A_1\ Y \quad \{A.a = g\,(A_1.a, Y.y)\} \\ A & \rightarrow & X \quad \{A.a = f\,(X.x)\} \end{array}$$

Здесь $A.a$ — синтезируемый атрибут леворекурсивного нетерминала A, а X и Y — отдельные грамматические символы с синтезируемыми атрибутами $X.x$ и $Y.y$ соответственно. Они могут представлять строки из нескольких грамматических символов, каждый со своими атрибутами, поскольку схема содержит произвольную функцию g, вычисляющую $A.a$ в рекурсивной продукции, и произвольную функцию f, вычисляющую $A.a$ в другой продукции. В любом случае f и g получают в качестве аргументов атрибуты, доступ к которым разрешен в случае S-атрибутного СУО.

Мы хотим преобразовать лежащую в основе грамматику в

$$\begin{array}{lcl} A & \rightarrow & X\ R \\ R & \rightarrow & Y\ R \mid \epsilon \end{array}$$

На рис. 5.23 показано, что должна делать СУТ на основе новой грамматики. На рис. 5.23, *а* мы видим результат работы постфиксной СУТ на основе исходной грамматики. Здесь один раз применяется функция f, соответствующая использованию продукции $A \rightarrow X$, а затем функция g применяется столько раз, сколько раз используется продукция $A \rightarrow A\ Y$. Поскольку R генерирует "остаток" строки из Y, его трансляция зависит от строки слева от этого грамматического символа, т.е.

от строки вида $XYY\cdots Y$. Каждое использование продукции $R \to Y\ R$ приводит к применению функции g. В случае R мы используем наследуемый атрибут $R.i$ для накопления результата последовательного применения функций g, начиная со значения атрибута $A.a$.

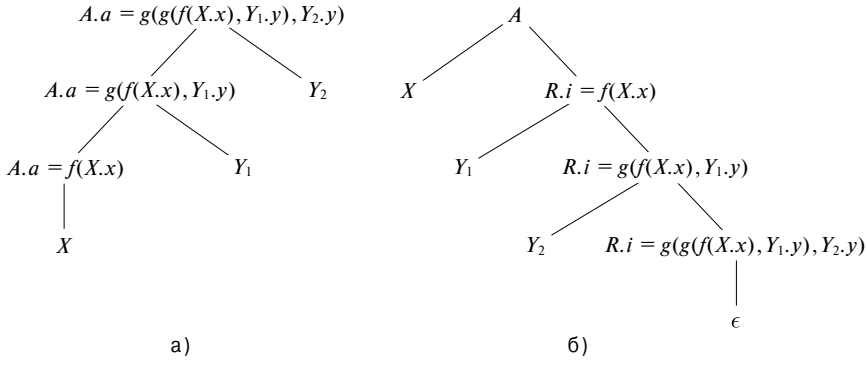

Рис. 5.23. Устранение левой рекурсии из постфиксной СУТ

Кроме того, R имеет синтезируемый атрибут $R.s$, не показанный на рис. 5.23. Этот атрибут в первый раз вычисляется, когда R завершает генерацию символов Y, о чем свидетельствует использование продукции $R \to \epsilon$. Затем $R.s$ копируется вверх по дереву, так что он становится значением $A.a$ для всего выражения $XYY\cdots Y$. На рис. 5.23 показана ситуация, когда A генерирует строку XYY, и видно, что значение $A.a$ в корне дерева на рис. 5.23, *а* включает два использования функции g. То же самое можно сказать и о $R.i$ в нижней части дерева на рис. 5.23, *б*; это значение уже как значение $R.s$ затем копируется вверх по дереву.

Итак, для завершения трансляции используется следующая СУТ:

$$
\begin{array}{lcl}
A & \to & X\ \ \{R.i = f\,(X.x)\}\ \ R\ \ \{A.a = R.s\} \\
R & \to & Y\ \ \{R_1.i = g\,(R.i, Y.y)\}\ \ R_1\ \ \{R.s = R_1.s\} \\
R & \to & \epsilon\ \ \{R.s = R.i\}
\end{array}
$$

Обратите внимание, что наследуемый атрибут $R.i$ вычисляется непосредственно перед использованием R в теле продукции, в то время как синтезируемые атрибуты $A.a$ и $R.s$ вычисляются в конце продукций. Таким образом, все значения, необходимые для вычисления этих атрибутов, будут доступны из вычислений, проведенных в телах продукций слева.

5.4.5 СУТ для L-атрибутных определений

В разделе 5.4.1 мы преобразовали S-атрибутное СУО в постфиксную СУТ с действиями на правом конце продукций. Поскольку лежащая в основе СУТ

грамматика принадлежит классу LR, постфиксная СУТ может анализироваться и транслироваться снизу вверх.

Рассмотрим теперь более общий случай L-атрибутного СУО. Будем считать, что грамматика поддается нисходящему синтаксическому анализу, поскольку, если это не так, зачастую трансляцию невозможно выполнить ни LL-, ни LR-синтаксическим анализатором. Для произвольной грамматики применим метод, состоящий в присоединении действий к дереву разбора и выполнении их в процессе обхода дерева в прямом порядке.

Вот правила, используемые при превращении L-атрибутного СУО в СУТ.

1. Вставить действие, которое вычисляет наследуемые атрибуты нетерминала A, непосредственно перед вхождением A в тело продукции. Если несколько наследуемых атрибутов A ациклически зависят друг от друга, следует упорядочить вычисление атрибутов с тем, чтобы сначала вычислялись те атрибуты, которые требуются первыми.

2. Поместить действия, вычисляющие синтезируемые атрибуты заголовка продукции, в конце тела соответствующей продукции.

Проиллюстрируем эти принципы на двух расширенных примерах. Первый относится к области полиграфии и иллюстрирует, как методы компиляции могут использоваться в приложениях, далеких от того, что обычно представляется при словосочетании “язык программирования”. Второй пример связан с генерацией промежуточного кода для типичной конструкции языка программирования — цикла while.

Пример 5.18. Этот пример вдохновлен языками для набора математических формул. Одним из первых таких языков был язык `Eqn`; ряд идей из `Eqn` можно обнаружить и в системе TEX, которая использовалась при подготовке данной книги.

Мы сосредоточимся только на возможности определения нижних индексов, индексов у индексов и т.д., игнорируя верхние индексы, встроенные дроби и прочие особенности математических формул. В `Eqn` строка `a sub i sub j` указывала на выражение a_{i_j}. Простейшей грамматикой для *боксов* (элементов текста, ограниченных прямоугольником) является

$$B \quad \rightarrow \quad B_1\ B_2 \mid B_1\ \textbf{sub}\ B_2 \mid (B_1) \mid \textbf{text}$$

В соответствии с приведенными четырьмя продукциями бокс может представлять собой одно из нижеперечисленного.

1. Два смежных бокса, первый из которых, B_1, находится слева от второго, B_2.

2. Бокс и бокс нижнего индекса. Второй бокс имеет меньший размер и находится ниже и правее первого бокса.

3. Бокс в скобках для группировки боксов и индексов. `Eqn` и TEX для группировки используют фигурные скобки, но, чтобы избежать путаницы с фигурными скобками, указывающими действия в СУТ, здесь для группировки будут использоваться круглые скобки.

4. Текстовая строка, т.е. произвольная строка символов.

Эта грамматика неоднозначна, но ее можно использовать для восходящего синтаксического анализа при условии правоассоциативности отношения смежности и индексирования, причем приоритет оператора **sub** выше приоритета смежности.

Выражение выводится путем построения бóльших боксов, окружающих меньшие. На рис. 5.24 смежные боксы для E_1 и *.height* образуют бокс для E_1*.height*. Левый бокс — для E_1 — состоит из боксов для E и для индекса 1. Индекс 1 обрабатывается путем уменьшения его размера примерно на 30%, опускания его вниз и размещения после бокса для E. Хотя *.height* рассматривается нами как текстовая строка, прямоугольники в ее боксе показывают, каким образом он может быть построен из боксов для отдельных букв.

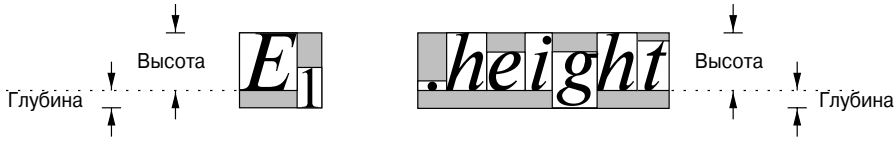

Рис. 5.24. Построение больших боксов из меньших

В этом примере мы остановимся только на вертикальной геометрии боксов. Горизонтальная геометрия — ширина боксов — также представляет интерес, в особенности в связи с тем, что различные символы имеют разную ширину. Возможно, это не сразу бросается в глаза, но каждый из различных символов на рис. 5.24 имеет свою ширину, отличную от ширины других символов.

С вертикальной геометрией боксов связаны следующие значения.

а) *Кегль* (point size) используется для размещения текста в боксе. Будем считать, что символы, не являющиеся символами индексов, имеют кегль 10. Далее, считаем, что если бокс имеет кегль p, то кегль его индекса — $0.7p$. Наследуемый атрибут $B.ps$ представляет кегль блока B. Этот атрибут должен быть наследуемым, поскольку то, насколько данный бокс должен быть меньше основного, определяется его уровнем индексирования.

б) Каждый бокс имеет *базовую линию*, или базис (baseline), представляющий собой вертикальное положение, соответствующее низу текста (без учета букв наподобие “g”, части которых опускаются ниже базовой линии). На рис. 5.24 базовая линия для E, *.height* и всего выражения в целом показана

точками. Базовая линия индекса (на рисунке — базис бокса, содержащего 1) находится ниже основной базовой линии.

в) Бокс имеет *высоту* (height), представляющую собой расстояние от базовой линии до вершины бокса. Высота бокса B определяется его синтезируемым атрибутом $B.ht$.

г) Бокс имеет *глубину* (depth), представляющую собой расстояние от базовой линии до дна бокса. Глубина бокса B определяется его синтезируемым атрибутом $B.dp$.

СУО на рис. 5.25 содержит правила для вычисления кегля, высоты и глубины боксов. Продукция 1 используется для назначения атрибуту $B.ps$ начального значения кегля 10.

Продукция	Семантические правила
1) $S \to B$	$B.ps = 10$
2) $B \to B_1\ B_2$	$B_1.ps = B.ps$
	$B_2.ps = B.ps$
	$B.ht = \max(B_1.ht, B_2.ht)$
	$B.dp = \max(B_1.dp, B_2.dp)$
3) $B \to B_1$ **sub** B_2	$B_1.ps = B.ps$
	$B_2.ps = 0.7 \times B.ps$
	$B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps)$
	$B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps)$
4) $B \to (B_1)$	$B_1.ps = B.ps$
	$B.ht = B_1.ht$
	$B.dp = B_1.dp$
5) $B \to$ **text**	$B.ht = \mathit{getHt}(B.ps, \textbf{text}.\mathit{lexval})$
	$B.dp = \mathit{getDp}(B.ps, \textbf{text}.\mathit{lexval})$

Рис. 5.25. СУО для текстовых боксов

Продукция 2 обрабатывает смежные боксы. Кегль копируется вниз по дереву разбора, т.е. два подбокса основного бокса наследуют тот же кегль, что и у большего бокса. Высота и глубина вычисляются вверх по дереву при помощи функции поиска максимального значения, т.е. высота большего бокса представляет собой максимальную из высот своих двух компонентов; то же касается и глубины бокса.

Продукция 3 обрабатывает индексы, что является немного более сложной задачей, чем предыдущая. В этом очень упрощенном примере считается, что кегль ин-

декса составляет 70% от родительского. Реальность существенно более сложная, поскольку индексы не могут уменьшаться до бесконечности; на практике после нескольких уровней размеры индексов перестают уменьшаться. Далее, в примере полагается, что базовая линия бокса индекса опущена на 25% от размера родителя; ситуация в реальности существенно сложнее, чем эта упрощенная модель.

Продукция 4 копирует атрибуты при использовании скобок. Наконец, продукция 5 обрабатывает листья, представляющие боксы текста. Здесь опять же реальная ситуация весьма сложна, так что в СУО показаны две не определенные функции *getHt* и *getDp*, которые работают с таблицами, указывающими для каждого шрифта максимальную высоту и максимальную глубину любого символа текстовой строки. Сама по себе строка передается как атрибут *lexval* терминала **text**.

Наша последняя задача состоит в преобразовании СУО в СУТ, следуя правилам для L-атрибутных СУО, к каковым относится и СУО на рис. 5.25. Соответствующая СУТ показана на рис. 5.26. Для большей удобочитаемости ставшие весьма длинными тела продукций разбиты на несколько строк. Тело каждой продукции состоит из всех строк, расположенных выше заголовка следующей продукции. □

	Продукция	Действия
1)	$S \rightarrow B$	$\{\ B.ps = 10;\ \}$
2)	$B \rightarrow$	$\{\ B_1.ps = B.ps;\ \}$
	B_1	$\{\ B_2.ps = B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht);$ $B.dp = \max(B_1.dp, B_2.dp);\ \}$
3)	$B \rightarrow$	$\{\ B_1.ps = B.ps;\ \}$
	B_1 **sub**	$\{\ B_2.ps = 0.7 \times B.ps;\ \}$
	B_2	$\{\ B.ht = \max(B_1.ht, B_2.ht - 0.25 \times B.ps);$ $B.dp = \max(B_1.dp, B_2.dp + 0.25 \times B.ps);\ \}$
4)	$B \rightarrow$ (	$\{\ B_1.ps = B.ps;\ \}$
	B_1)	$\{\ B.ht = B_1.ht\,;$ $B.dp = B_1.dp\,;\ \}$
5)	$B \rightarrow$ **text**	$\{\ B.ht = \mathit{getHt}\,(B.ps, \textbf{text}.\mathit{lexval});$ $B.dp = \mathit{getDp}\,(B.ps, \textbf{text}.\mathit{lexval});\ \}$

Рис. 5.26. СУТ для текстовых боксов

Следующий пример связан с простой инструкцией while и генерацией промежуточного кода для данного типа инструкций. Промежуточный код рассматривается как атрибут со строковым значением. Позже мы познакомимся с методами, которые включают запись частей строковых атрибутов в процессе синтаксического анализа, что позволяет избегать копирования длинных строк для построения еще более длинных строк. Такого рода метод использовался в примере 5.17, при генерации постфиксной записи инфиксного выражения "на лету" вместо вычисления атрибута. Однако пока что будем создавать строковые атрибуты путем конкатенации.

Пример 5.19. В этом примере нам достаточно одной продукции:

$$S \quad \rightarrow \quad \textbf{while}\ (C)\ S_1$$

Здесь S — нетерминал, который генерирует все виды инструкций, вероятно, включая условные конструкции, присваивания и др. В нашем примере C представляет собой условное выражение — булево выражение, которое после вычисления принимает значение true или false.

В этом примере единственный объект, который мы будем генерировать, — это метки. Все прочие команды промежуточного кода считаются генерируемыми не показанными частями СУТ. В частности, мы генерируем явные команды вида **label** L, где L — идентификатор, указывающие, что L — метка следующей за ней команды. Предполагается, что промежуточный код подобен рассматривавшемуся в разделе 2.8.4.

Конструкция while работает следующим образом. Сначала вычисляется условное выражение C. Если оно истинно, управление передается в начало кода для S_1; если оно ложно, управление передается коду, следующему за конструкцией while. Код S_1 должен по завершении выполнения передать управление в начало конструкции while, к коду, который вычисляет значение C и который не показан на рис. 5.27.

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$

$L1 = \mathit{new}();$
$L2 = \mathit{new}();$
$S_1.\mathit{next} = L1;$
$C.\mathit{false} = S.\mathit{next};$
$C.\mathit{true} = L2;$
$S.\mathit{code} = \textbf{label} \parallel L1 \parallel C.\mathit{code} \parallel \textbf{label} \parallel L2 \parallel S_1.\mathit{code}$

Рис. 5.27. СУО для конструкции while

Для генерации корректного промежуточного кода используются следующие атрибуты.

1. Наследуемый атрибут $S.next$ помечает начало кода, который должен быть выполнен после завершения S.

2. Синтезируемый атрибут $S.code$ представляет собой последовательность шагов промежуточного кода, которая реализует инструкцию S и завершается переходом к метке $S.next$.

3. Наследуемый атрибут $C.true$ помечает начало кода, который должен быть выполнен, если значение C истинно.

4. Наследуемый атрибут $C.false$ помечает начало кода, который должен быть выполнен, если значение C ложно.

5. Синтезируемый атрибут $C.code$ представляет собой последовательность шагов промежуточного кода, которая реализует условие C и переходит в зависимости от вычисленного значения C либо к метке $C.true$, либо к метке $C.false$.

СУО, вычисляющее эти атрибуты конструкции while, приведено на рис. 5.27. Некоторые моменты этого СУО требуют пояснений.

- Функция *new* генерирует новые метки.

- Переменные $L1$ и $L2$ хранят метки, которые потребуются в генерируемом коде. $L1$ представляет начало кода конструкции while, и код S_1 по окончании работы должен выполнять переход к ней. Именно поэтому выполняется присваивание $L1$ атрибуту $S_1.next$. $L2$ представляет начало кода S_1 и становится значением атрибута $C.true$, поскольку именно сюда выполняется переход, если вычисленное условие истинно.

- Обратите внимание, что $C.false$ устанавливается равным $S.next$, поскольку, когда условие ложно, выполняется код, следующий за кодом S.

- Символ $||$ используется для обозначения конкатенации фрагментов промежуточного кода. Таким образом, значение $S.code$ начинается с метки $L1$, за которой следуют код условия C, другая метка $L2$ и код S_1.

Данное СУО является L-атрибутным. При его преобразовании в СУТ остается единственный вопрос — как обрабатывать метки $L1$ и $L2$, представляющие собой переменные, а не атрибуты? Если рассматривать действия как фиктивные нетерминалы, то такие переменные могут трактоваться как синтезируемые атрибуты фиктивных нетерминалов. Поскольку $L1$ и $L2$ не зависят от каких-либо других атрибутов, они могут быть назначены первому действию продукции. В результате получается СУТ со встроенными действиями, реализующая рассмотренное L-атрибутное определение и показанная на рис. 5.28. □

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& \{\ L1 = \mathit{new}();\ L2 = \mathit{new}();\ C.\mathit{false} = S.\mathit{next};\ C.\mathit{true} = L2;\ \} \\
C\ ) & \{\ S_1.\mathit{next} = L1;\ \} \\
S_1 & \{\ S.\mathit{code} = \textbf{label} \parallel L1 \parallel C.\mathit{code} \parallel \textbf{label} \parallel L2 \parallel S_1.\mathit{code};\ \}
\end{array}
$$

Рис. 5.28. СУТ для конструкции while

5.4.6 Упражнения к разделу 5.4

Упражнение 5.4.1. В разделе 5.4.2 упоминалось, что из LR-состояния в стеке синтаксического анализатора можно вывести, какой именно грамматический символ представлен этим состоянием. Каким образом можно это сделать?

Упражнение 5.4.2. Перепишите СУТ

$$
\begin{array}{lcl}
A & \rightarrow & A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0 \\
B & \rightarrow & B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1
\end{array}
$$

так, чтобы лежащая в ее основе грамматика перестала быть леворекурсивной. Здесь a, b, c и d — действия, а 0 и 1 — терминалы.

! **Упражнение 5.4.3.** Приведенная ниже СУТ вычисляет значение строки из 0 и 1, рассматривая ее как положительное бинарное число.

$$
\begin{array}{lcl}
B & \rightarrow & B_1\ 0\ \{B.\mathit{val} = 2 \times B_1.\mathit{val}\} \\
 & | & B_1\ 1\ \{B.\mathit{val} = 2 \times B_1.\mathit{val} + 1\} \\
 & | & 1\ \{B.\mathit{val} = 1\}
\end{array}
$$

Перепишите эту СУТ так, чтобы лежащая в ее основе грамматика перестала быть леворекурсивной, но вычисленное значение $B.\mathit{val}$ для входной строки оставалось тем же.

! **Упражнение 5.4.4.** Разработайте L-атрибутное СУО, аналогичное СУО из примера 5.10, для приведенных ниже продукций, каждая из которых представляет собой знакомую конструкцию управления потоком, имеющуюся в языке программирования C. Вам может потребоваться сгенерировать трехадресную команду для перехода к конкретной метке L — в этом случае генерируйте команду **goto** L.

а) $S \rightarrow \textbf{if}\ (C)\ S_1\ \textbf{else}\ S_2$

б) $S \rightarrow \textbf{do}\ S_1\ \textbf{while}\ (C)$

в) $S \rightarrow '\{'\ L\ '\}';\ L \rightarrow L\ S \mid \epsilon$

Обратите внимание на то, что любая инструкция может содержать переход из средины к следующей инструкции, так что просто по порядку сгенерировать код для каждой инструкции недостаточно.

Упражнение 5.4.5. Преобразуйте каждое из ваших СУО из упражнения 5.4.4 в СУТ, как это было сделано в примере 5.19.

Упражнение 5.4.6. Модифицируйте СУО на рис. 5.25 так, чтобы оно включало синтезируемый атрибут $B.le$, длину бокса. Длина двух смежных боксов равна сумме их длин. Затем добавьте новые правила в соответствующие места СУТ на рис. 5.26.

Упражнение 5.4.7. Модифицируйте СУО на рис. 5.25 так, чтобы оно включало надстрочные индексы, с использованием оператора **sup** между боксами. Если бокс B_2 является надстрочным индексом бокса B_1, то позиция базовой линии B_2 находится на 0.6 кегля бокса B_1 выше базовой линии бокса B_1. Добавьте новую продукцию и правила в СУТ на рис. 5.26.

5.5 Реализация L-атрибутных СУО

Поскольку многие приложения трансляций используют L-атрибутные определения, в этом разделе мы рассмотрим их реализацию более подробно. Трансляция путем обхода дерева разбора осуществляется следующими методами.

1. *Построение дерева разбора и его аннотирование*. Этот метод работает для любого нециклического СУО. С аннотированными деревьями разбора мы уже знакомились в разделе 5.1.2.

2. *Построение дерева разбора, добавление действий и выполнение действий в прямом порядке обхода*. Этот подход работает для любого L-атрибутного определения. Преобразование L-атрибутного СУО в СУТ рассматривалось в разделе 5.4.5; в частности, в нем рассматривались вопросы вставки действий в продукции на основе семантических правил такого СУО.

В этом разделе мы рассмотрим другие методы трансляции в процессе синтаксического анализа.

3. *Использование синтаксического анализатора, работающего методом рекурсивного спуска*, с одной функцией для каждого нетерминала. Функция для нетерминала A получает наследуемые атрибуты A в качестве аргументов и возвращает синтезируемые атрибуты A.

4. *Генерация кода "на лету"* с применением синтаксического анализатора, работающего методом рекурсивного спуска.

5. *Реализация СУТ вместе с LL-синтаксическим анализатором*. Атрибуты хранятся в стеке синтаксического анализа, а правила выбирают требующиеся им атрибуты из известных позиций в стеке.

6. *Реализация СУТ вместе с LR-синтаксическим анализатором.* Этот метод может показаться неожиданным, поскольку СУТ для L-атрибутного СУО обычно содержит действия в средине продукций, и в процессе LR-синтаксического анализа мы не можем знать точно, какая именно продукция используется, пока не построим все ее тело. Однако, как мы увидим, если лежащая в основе грамматика принадлежит к классу LL, то всегда можно провести как синтаксический анализ, так и трансляцию в восходящем направлении.

5.5.1 Трансляция в процессе синтаксического анализа методом рекурсивного спуска

Синтаксический анализатор, работающий методом рекурсивного спуска, для каждого нетерминала A имеет отдельную функцию A (об этом говорилось в разделе 4.4.1). Можно расширить синтаксический анализатор и превратить его в транслятор, если следовать следующим правилам.

а) Аргументами функции A являются наследуемые атрибуты нетерминала A.

б) Возвращаемое функцией A значение представляет собой набор синтезируемых атрибутов нетерминала A.

В теле функции A требуется как выполнить синтаксический анализ, так и обработать атрибуты.

1. Принимается решение о том, какая продукция используется для развертывания A.

2. Когда это необходимо, проверяется каждый терминал входного потока. Будем считать, что возврат не требуется, но перейти к синтаксическому анализу методом рекурсивного спуска с возвратом можно путем восстановления позиции во входном потоке при ошибке, как описано в разделе 4.4.1.

3. В локальных переменных сохраняются значения всех атрибутов, необходимые для вычисления наследуемых атрибутов нетерминалов в теле продукции или синтезируемых атрибутов нетерминала заголовка.

4. Вызываются функции, соответствующие нетерминалам в теле выбранной продукции, с передачей им корректных аргументов. Поскольку лежащее в основе СУО является L-атрибутным, все необходимые для передачи атрибуты к этому моменту уже вычислены и сохранены в локальных переменных.

```
string S(label next) {
    string Scode, Ccode; /* Локальные переменные с фрагментами кода */
    label L1, L2; /* Локальные метки */
    if ( Текущий входной символ == токен while ) {
        Перемещение по входному потоку;
        Проверить наличие '(' во входной строке и перейти к новой
            позиции;
        L1 = new();
        L2 = new();
        Ccode = C(next, L2);
        Проверить наличие ')' во входной строке и перейти к новой
            позиции;
        Scode = S(L1);
        return("label"||L1||Ccode||"label"||L2||Scode);
    }
    else /* Инструкции других видов */
}
```

Рис. 5.29. Реализация конструкции while при помощи синтаксического анализатора, работающего методом рекурсивного спуска

Пример 5.20. Рассмотрим СУО и СУТ для конструкции while из примера 5.19. Набросок псевдокода значимой части функции S показан на рис. 5.29.

Здесь функция S показана как хранящая и возвращающая длинные строки. На практике было бы более эффективно, если бы функции наподобие S и C возвращали указатели на записи, представляющие эти строки. Тогда инструкция **return** в функции S не выполняла бы показанную конкатенацию строк, а вместо этого строила бы запись или, возможно, дерево записей, выражающее конкатенацию строк, представленных *Scode* и *Ccode*, метками $L1$ и $L2$, а также двух строк `"label"`. □

Пример 5.21. Теперь вернемся к СУТ на рис. 5.26 для текстовых боксов. Сначала мы обратимся к синтаксическому анализу, поскольку грамматика, лежащая в основе СУТ на рис. 5.26, неоднозначна. Приведенная далее преобразованная грамматика делает отношение смежности и индексирование правоассоциативными, при этом оператор **sub** имеет более высокий приоритет, чем смежность:

$$
\begin{aligned}
S &\rightarrow B \\
B &\rightarrow T\ B_1 \mid T \\
T &\rightarrow F\ \textbf{sub}\ T_1 \mid F \\
F &\rightarrow (B) \mid \textbf{text}
\end{aligned}
$$

Два новых нетерминала, T и F, выполняют те же функции, что и слагаемые и сомножители в выражениях. Здесь "сомножитель", генерируемый F, представляет собой либо бокс в скобках, либо строку. "Слагаемое", генерируемое T, представляет собой "множитель" с последовательностью индексов, а бокс, генерируемый B, — последовательность смежных "слагаемых".

Атрибуты B переходят к T и F, поскольку эти новые нетерминалы также обозначают боксы; они введены исключительно во вспомогательных целях. Таким образом, и T, и F имеют наследуемый атрибут *ps* и синтезируемые атрибуты *ht* и *dp*, с семантическими действиями, которые могут быть получены из СУТ на рис. 5.26.

Грамматика пока что не готова для нисходящего синтаксического анализа, поскольку продукции для B и T имеют общие префиксы. Рассмотрим, например, T. Нисходящий синтаксический анализатор не может выбрать одну из двух продукций для T, просматривая только один символ из входного потока. К счастью, можно воспользоваться разновидностью левой факторизации, рассматривавшейся в разделе 4.3.4. В случае СУТ понятие общего префикса применимо также и к действиям. Обе продукции для T начинаются с нетерминала F, наследующего атрибут *ps* от T.

Псевдокод на рис. 5.30 для $T\,(ps)$ включает код $F\,(ps)$. После применения левой факторизации к $T \rightarrow F$ **sub** $T_1 \mid F$ остается только один вызов F; в псевдокоде показан результат подстановки кода F вместо этого вызова.

Функция T будет вызвана функцией B как $T\,(10.0)$ (этот вызов здесь не показан). Функция возвращает пару, состоящую из значений высоты и глубины бокса, генерируемого нетерминалом T; на практике она возвращает запись, содержащую высоту и глубину.

Функция T начинается с проверки наличия левой скобки — в этом случае используется продукция $F \rightarrow (B)$. Функция сохраняет значения, которые возвращает функция B, но если после B не следует правая скобка, то это означает, что мы столкнулись с синтаксической ошибкой, обработка которой здесь не показана.

В противном случае, если текущий входной символ — **text**, функция T использует функции *getHt* и *getDp* для определения высоты и глубины этого текста.

Затем T выясняет, является ли следующий бокс индексом, и если является, то соответствующим образом изменяет кегль. Для вычислений используются действия, связанные с продукцией $B \rightarrow B$ **sub** B на рис. 5.26 и вычисляющие высоту и глубину большого бокса. В противном случае функция просто возвращает значения, вычисленные функцией F: $(h1, d1)$. □

5.5.2 Генерация кода "на лету"

Построение длинных строк кода, являющихся значениями атрибутов, как это было сделано в примере 5.20, нежелательно по ряду причин, включая время,

```
(float, float) T(float ps) {
    float h1, h2, d1, d2; /* Локальные переменные для высоты и глубины */
    /* Начало кода F(ps) */
    if ( Текущий символ == '(' ) {
        Перемещение по входному потоку;
        (h1, d1) = B(ps);
        if (Текущий символ != ')' ) Синтаксическая ошибка: требуется ')';
        Перемещение по входному потоку;
    }
    else if ( Текущий символ == text ) {
        Обозначим значение text.lexval как t;
        Перемещение по входному потоку;
        h1 = getHt(ps, t);
        d1 = getDp(ps, t);
    }
    else Синтаксическая ошибка: требуется text или ')';
    /* Конец кода F(ps) */
    if ( Текущий символ == sub ) {
        Перемещение по входному потоку;
        (h2, d2) = T(0.7*ps);
        return (max(h1, h2 − 0.25 ∗ ps), max(d1, d2 + 0.25 ∗ ps));
    }
    return (h1, d1);
}
```

Рис. 5.30. Рекурсивный спуск для текстовых боксов

необходимое для копирования и перемещения длинных строк. В распространенных случаях, таких как наш пример с генерацией кода, вместо этого можно инкрементно генерировать части кода с записью в массив или выходной файл при помощи действий из СУТ. Для этого требуется, чтобы выполнялось следующее.

1. *Главный* (main) атрибут для одного или нескольких нетерминалов. Для удобства будем считать, что все главные атрибуты представляют собой строковые значения. В примере 5.20 таковыми были атрибуты $S.code$ и $C.code$; прочие атрибуты не были главными.

2. Главные атрибуты являются синтезируемыми.

3. Правила для вычисления главных атрибутов гарантируют следующее.

   а) Главный атрибут представляет собой конкатенацию главных атрибутов нетерминалов, имеющихся в теле соответствующей продукции,

Типы главных атрибутов

Наше упрощающее предположение о том, что главные атрибуты представляют собой строки, слишком ограничительное. На самом деле типы главных атрибутов должны иметь значения, которые могут быть построены путем конкатенации элементов. Например, список объектов любого типа вполне годится в качестве главного атрибута, поскольку список может быть представлен таким образом, что к его концу можно эффективно добавлять новые элементы. Таким образом, если назначение главного атрибута состоит в представлении последовательности команд промежуточного кода, то этот код можно получить путем записи инструкций в конец массива объектов. Конечно, требования, перечисленные в разделе 5.5.2, применимы и к спискам; например, главные атрибуты должны собираться из других главных атрибутов путем конкатенации в соответствующем порядке.

возможно, с другими элементами, не являющимися главными атрибутами, такими как строки **label** или значения меток $L1$ и $L2$.

б) Главные атрибуты нетерминалов находятся в правиле в том же порядке, что и сами нетерминалы в теле продукции.

Как следствие приведенных условий главный атрибут может быть построен путем добавления в конкатенацию элементов, не являющихся главными атрибутами. Для инкрементной генерации значений главных атрибутов нетерминалов в теле продукции можно основываться на рекурсивных вызовах их функций.

Пример 5.22. Можно модифицировать функцию, приведенную на рис. 5.29, таким образом, чтобы она выводила элементы трансляции $S.code$ вместо возврата соответствующего значения. Такая измененная функция S показана на рис. 5.31.

На рис. 5.31 S и C не имеют возвращаемых значений, поскольку их синтезируемые атрибуты генерируются путем печати. Обратите внимание на важность положений инструкций *print*: сначала выводится `label` $L1$, затем — код для C (который представляет собой то же, что и значение *Ccode* на рис. 5.29), затем — `label` $L2$ и наконец — код из рекурсивного вызова для S (который представляет собой то же, что и значение *Scode* на рис. 5.29). Таким образом, код, выводимый этим вызовом S, в точности такой же, как и значение, возвращаемое на рис. 5.29.□

Кстати, можно внести те же изменения и в соответствующую СУТ — превратить построение главного атрибута в действия по выводу элементов этого атрибута. На рис. 5.32 показана СУТ из рис. 5.28, переделанная для генерации кода “на лету”.

```
void S(label next) {
    label L1, L2; /* Локальные метки */
    if ( Текущий символ == токен while ) {
        Перемещение по входному потоку;
        Проверить наличие '(' во входной строке и перейти к новой
            позиции;
        L1 = new();
        L2 = new();
        print("label", L1);
        C(next, L2);
        Проверить наличие ')' во входной строке и перейти к новой
            позиции;
        print("label", L2);
        S(L1);
    }
    else /* Инструкции других видов */
}
```

Рис. 5.31. Генерация кода для конструкции while синтаксическим анализатором методом рекурсивного спуска "на лету"

$$
\begin{array}{lll}
S \rightarrow \textbf{while}\ (& \{\ L1 = new();\ L2 = new();\ C.false = S.next; & \\
 & \quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} & \\
C\ ) & \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} & \\
S_1 & &
\end{array}
$$

Рис. 5.32. СУТ для генерации "на лету" кода конструкции while

5.5.3 L-атрибутные СУО и LL-синтаксический анализ

Предположим, что L-атрибутное СУО основано на LL-грамматике и что мы конвертировали его в СУТ с действиями, вставленными в продукции, как описано в разделе 5.4.5. После этого можно выполнить трансляцию в процессе LL-анализа путем расширения стека синтаксического анализатора таким образом, чтобы он хранил действия и некоторые данные, необходимые для вычисления атрибутов. Обычно данные представляют собой копии атрибутов.

В дополнение к записям, представляющим терминалы и нетерминалы, стек синтаксического анализатора хранит *записи действий* (action-records), представляющие выполняемые действия, и *записи синтеза* (synthesize-records) для хранения синтезируемых атрибутов нетерминалов. Управление атрибутами в стеке осуществляется в соответствии со следующими принципами.

- Наследуемые атрибуты нетерминала A помещаются в записи стека, представляющей данный нетерминал. Код для вычисления атрибутов обычно представляется записью действий непосредственно над записью стека для A; в действительности преобразование L-атрибутного СУО в СУТ гарантирует, что запись действия будет находиться непосредственно над A.

- Синтезируемые атрибуты нетерминала A размещаются в отдельной записи синтеза, которая находится в стеке непосредственно под записью для A.

Данная стратегия размещает записи различных типов в стеке синтаксического анализа, полагая, что такие вариантные типы записей корректно обрабатываются как подклассы класса "запись стека". На практике можно скомбинировать несколько записей в одну, но, пожалуй, пояснить идеи будет проще, если данные для различных целей будут храниться в различных записях.

Записи действий содержат указатели на выполняемый код. Действия могут также появляться и в записях синтеза; эти действия обычно размещают копии синтезируемых атрибутов в других записях ниже по стеку, где значения этих атрибутов потребуются после того, как запись синтеза и ее атрибуты будут сняты со стека.

Бегло взглянем на LL-синтаксический анализ, чтобы понять необходимость создания временных копий атрибутов. Из раздела 4.4.4 известно, что управляемый таблицей синтаксического анализа LL-анализатор имитирует левое порождение. Если w — входная строка, соответствие которой проверено до текущего момента, то в стеке хранится последовательность грамматических символов α, такая, что $S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$, где S — стартовый символ. Когда синтаксический анализатор выполняет раскрытие с использованием продукции $A \rightarrow B\ C$, он заменяет A на вершине стека на $B\ C$.

Предположим, что нетерминал C имеет наследуемый атрибут $C.i$. В силу продукции $A \rightarrow B\ C$ наследуемый атрибут $C.i$ может зависеть не только от наследуемых атрибутов A, но и от всех атрибутов B. Таким образом, перед тем, как вычислять атрибут $C.i$, следует полностью обработать B. Значит, все необходимые для вычисления $C.i$ атрибуты должны быть сохранены в записи действий, которые вычисляют $C.i$. В противном случае, когда синтаксический анализатор заменит A на вершине стека на $B\ C$, наследуемые атрибуты A просто исчезнут вместе с соответствующими записями в стеке.

Поскольку лежащее в основе СУО — L-атрибутное, это гарантирует, что значения наследуемых атрибутов A доступны, когда A оказывается на вершине стека. Следовательно, эти значения доступны в момент копирования в записи действий, которые вычисляют наследуемые атрибуты C. Кроме того, нет никаких проблем с памятью для синтезируемых атрибутов A, так как они находятся в записи син-

теза, которая при раскрытии с использованием продукции $A \rightarrow B\ C$ остается в стеке, ниже B и C.

При обработке B можно выполнить действия (с помощью записи, находящейся в стеке непосредственно над B), которые копируют его наследуемые атрибуты для использования нетерминалом C, а после того, как B обработан, запись синтеза для B может скопировать его синтезируемые атрибуты для использования при необходимости нетерминалом C. Аналогично синтезируемые атрибуты A могут потребовать для вычисления их значений временных переменных, которые могут быть скопированы в запись синтеза для A после обработки B, а затем C. Вот общий принцип, который обеспечивает работоспособность всех описанных копирований атрибутов.

- Все копирования выполняются между записями, которые создаются в процессе одного раскрытия одного нетерминала. Таким образом, каждая из этих записей знает, где именно ниже в стеке находится каждая другая запись, и может безопасно копировать в них необходимые значения.

Приведенный далее пример иллюстрирует реализацию наследуемых атрибутов в процессе LL-синтаксического анализа путем аккуратного копирования значений атрибутов. В приведенном примере возможны сокращения и оптимизации, в частности в случае правил копирования, состоящих в присваивании значения одного атрибута другому. Однако пока что мы отложим этот вопрос до примера 5.24, в котором иллюстрируются записи синтеза.

Пример 5.23. В этом примере реализуется представленная на рис. 5.32 СУТ, “на лету” генерирующая код для конструкции while. В этой СУТ нет синтезируемых атрибутов, не считая фиктивных атрибутов, представляющих метки.

На рис. 5.33, *а* показана ситуация, в которой для раскрытия S используется продукция while, поскольку очередным символом во входном потоке оказался символ **while**. Запись на вершине стека — это запись для S, которая содержит только наследуемый атрибут $S.next$, который, будем считать, имеет значение x. Поскольку здесь выполняется нисходящий синтаксический анализ, вершина стека в соответствии с ранее принятыми соглашениями показана на рисунке слева.

На рис. 5.33, *б* показана ситуация непосредственно после раскрытия S. Перед нетерминалами C и S_1 имеются записи действий, соответствующие действиям лежащей в основе СУТ из рис. 5.32. Запись для C содержит место для наследуемых атрибутов *true* и *false*, а запись для S_1, как и все записи для S, — место для атрибута *next*. На рисунке значения этих полей показаны как ?, поскольку пока что значения указанных атрибутов неизвестны.

Синтаксический анализатор распознает во входном потоке символы **while** и (и снимает их записи со стека. Теперь первая запись действий находится на вершине стека и должна быть выполнена. Эта запись содержит поле *snext*, предназначенное для хранения копии наследуемого атрибута $S.next$. Когда S снимается

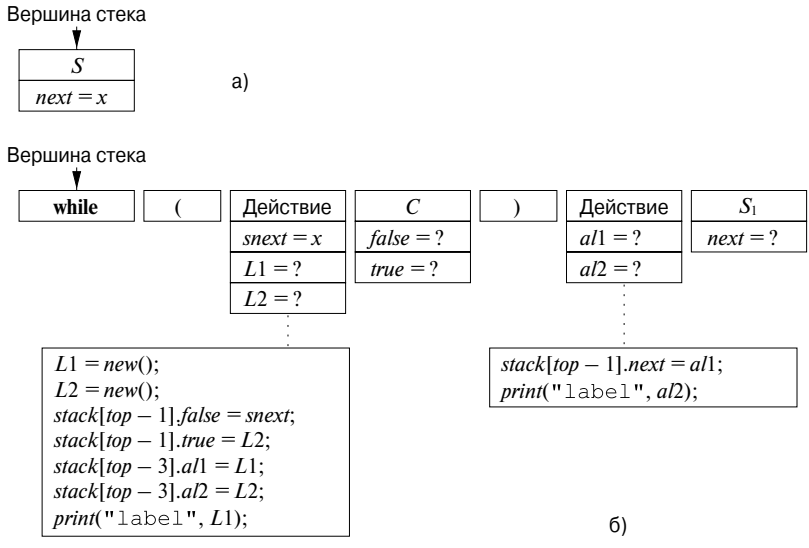

Рис. 5.33. Раскрытие S в соответствии с продукцией для конструкции while

со стека, значение $S.next$ копируется в поле *snext* для использования в процессе вычисления наследуемых атрибутов C. Код первого действия генерирует новые значения для $L1$ и $L2$, которые мы полагаем равными соответственно y и z. Следующий шаг делает z значением $C.true$. Присваивание $stack\,[top-1]\,.true = L2$ записано с учетом знания о том, что оно будет выполняться только в тот момент, когда данная запись действий будет находиться на вершине стека, так что $top - 1$ указывает на запись непосредственно под ней, т.е. запись для C.

Затем первая запись действий копирует $L1$ в поле $al1$ второго действия, где это значение будет использовано при вычислении $S_1.next$. Она также копирует $L2$ в поле $al2$ второго действия; это значение необходимо для корректного вывода, выполняемого этим действием. Наконец, первая запись действий выводит `label` y.

Ситуация после завершения первого действия и снятия со стека его записи показана на рис. 5.34. Значения наследуемых атрибутов в записи C корректно установлены, как и значения временных переменных $al1$ и $al2$ во второй записи действий. В этот момент раскрывается C и мы полагаем, что генерируется код, реализующий этот тест, который содержит переходы к меткам x и z там, где это требуется. После того как запись для C снимается со стека, на его вершине оказывается запись для), что заставляет синтаксический анализатор проверять наличие) во входном потоке.

Когда выполняется действие из записи, находящейся в стеке над записью для S_1, его код устанавливает $S_1.next$ и выводит `label` z. После этого на вершине

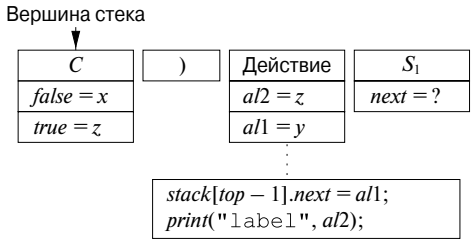

Рис. 5.34. Стек после завершения действия, находившегося в стеке над C

стека оказывается запись для нетерминала S_1, при раскрытии которого, как мы полагаем, выполняется корректная генерация кода, реализующего скрывающиеся за этим нетерминалом инструкции любых типов и переходящего к метке y. □

Пример 5.24. Рассмотрим ту же конструкцию while, но теперь трансляция генерирует код не "на лету", а как синтезируемый атрибут $S.code$. Чтобы лучше понять пояснения, приводимые в данном примере, желательно все время помнить следующий инвариант, который, как мы полагаем, выполняется для каждого нетерминала.

- Каждый нетерминал, который имеет связанный с ним код, оставляет его в виде строки в записи синтеза, находящейся в стеке непосредственно под ним.

Полагая данное утверждение истинным, мы будем работать с while-продукцией так, чтобы поддерживать это утверждение как инвариант.

На рис. 5.35, *а* показана ситуация непосредственно перед развертыванием S с использованием продукции для конструкции while. На вершине стека находится запись для S; она содержит поле для своего наследуемого атрибута $S.next$, как в примере 5.23. Непосредственно под этой записью располагается запись синтеза для данного экземпляра S. В ней имеется поле для $S.code$, как и у всех записей синтеза для S. Показаны также некоторые другие поля для локальных данных и действий, поскольку СУТ для продукции while на рис. 5.28, конечно же, является частью большей СУТ.

Наше раскрытие S основано на СУТ, представленной на рис. 5.28, и показано на рис. 5.35, *б*. Для ускорения в процессе раскрытия мы полагаем, что наследуемый атрибут $S.next$ присваивается непосредственно $C.false$, а не размещается в первом действии и затем копируется в запись для C.

Рассмотрим, что делает каждая запись, оказавшись на вершине стека. Сначала запись для **while** заставляет выполнить проверку соответствия токена **while** входному символу (соответствие должно выполняться, иначе для раскрытия S будет использоваться другая продукция). После того как со стека будут сняты **while** и (,

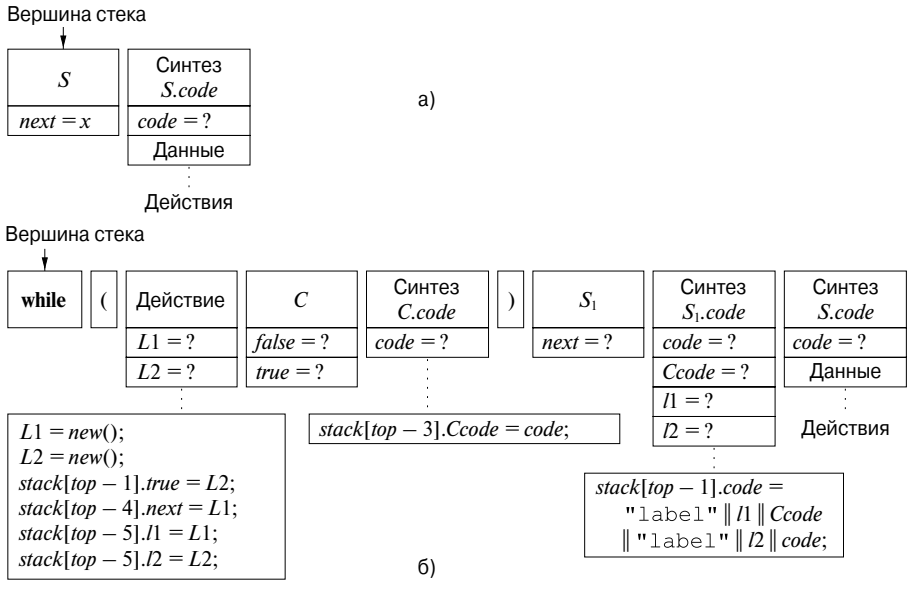

Рис. 5.35. Раскрытие S с синтезируемым атрибутом, конструируемым в стеке

будет выполнен код записи действий. Он генерирует значения $L1$ и $L2$, и мы можем сократить вычисления, копируя их непосредственно в наследуемые атрибуты $S_1.next$ $C.true$. Два последних шага действия копируют значения $L1$ и $L2$ в запись "Синтез $S_1.code$".

Запись синтеза для S_1 выполняет двойную функцию: она не только хранит синтезируемый атрибут $S_1.code$, но и служит в качестве записи действий для завершения вычислений атрибутов всей продукции $S \to$ **while** (C) S_1. В частности, когда эта запись оказывается на вершине стека, она вычисляет синтезируемый атрибут $S.code$ и помещает его значение в запись синтеза для заголовка S.

Когда C находится на вершине стека, оба его наследуемых атрибута оказываются вычисленными. В соответствии со сформулированным ранее инвариантом мы предполагаем, что при этом корректно сгенерирован код для вычисления условия с переходом к соответствующей метке. Мы также считаем, что действие, выполняемое при раскрытии C, корректно размещает этот код в записи ниже в стеке в качестве значения синтезируемого атрибута $C.code$.

После снятия C со стека на его вершине оказывается запись синтеза для $C.code$. Этот код необходим в записи синтеза для $S_1.code$, поскольку именно в ней выполняется конкатенация всех элементов кода, образующих $S.code$. Таким образом, запись синтеза для $C.code$ содержит действие, состоящее в копировании $C.code$ в запись синтеза для $S_1.code$. После этого вершины стека достигает запись для токена), что приводит к выполнению проверки наличия) во входном потоке.

В предположении, что проверка выполнена успешно, на вершину стека поднимается запись для S_1. Согласно инварианту этот нетерминал раскрыт, и в результате соответствующий код корректно генерируется и размещается в поле *code* в записи синтеза для S_1.

Теперь все поля данных записи синтеза для S_1 заполнены, так что, когда оно оказывается на вершине стека, выполнению его действий ничто не препятствует. Действие состоит в конкатенации меток и кода из $C.code$ и $S_1.code$ в правильном порядке. Получающаяся в результате строка размещается ниже, т.е. в записи синтеза для S. Теперь у нас есть корректно вычисленный атрибут $S.code$, и, когда запись синтеза для S окажется на вершине стека, этот код будет доступен для размещения в другой записи ниже в стеке, где в конечном счете он будет собран в большей строке кода, реализующей элемент программы, частью которого является S. □

5.5.4 Восходящий синтаксический анализ L-атрибутных СУО

Любая трансляция, которая может быть выполнена в нисходящем направлении, может быть выполнена и при восходящем синтаксическом анализе. Точнее, для данных L-атрибутного СУО и LL-грамматики можно адаптировать грамматику для вычисления того же самого СУО над новой грамматикой в процессе LR-синтаксического анализа. Этот "трюк" состоит из трех частей.

1. Начнем с СУТ, построенной, как в разделе 5.4.5, которая размещает перед каждым нетерминалом действия по вычислению его наследуемых атрибутов, а в конце продукции — действия по вычислению синтезируемых атрибутов.

2. Введем в грамматику нетерминалы-маркеры на месте каждого вставленного действия. В каждое такое место вставляется свой маркер; для каждого маркера M существует только одна продукция, а именно — $M \to \epsilon$.

3. Модифицируем действие a, заменяемое маркером M в некоторой продукции $A \to \alpha\ \{a\}\ \beta$, и связываем с продукцией $M \to \epsilon$ действие a', которое

   а) копирует в качестве наследуемых атрибутов M любые атрибуты A или символов из α, которые требуются действию a;

   б) вычисляет атрибуты таким же способом, что и a, но делает эти атрибуты синтезируемыми атрибутами M.

   Это изменение выглядит некорректным, поскольку получается, что действие, связанное с продукцией $M \to \epsilon$, должно иметь доступ к атрибутам,

Возможна ли обработка L-атрибутного СУО на основе LR-грамматики

В разделе 5.4.1 мы видели, что каждое S-атрибутное СУО на основе LR-грамматики может быть реализовано в процессе восходящего синтаксического анализа. Из раздела 5.5.3 известно, что каждое L-атрибутное СУО на основе LL-грамматики может быть проанализировано при помощи нисходящего синтаксического анализа. Поскольку LL-грамматики являются истинным подмножеством LR-грамматик, а S-атрибутные СУО — истинным подмножеством L-атрибутных СУО, можно ли работать с любой LR-грамматикой и L-атрибутным СУО в восходящем направлении?

Нет, как показывают следующие интуитивно понятные доводы. Предположим, имеются продукция $A \to B\ C$ из LR-грамматики и наследуемый атрибут $B.i$, который зависит от наследуемых атрибутов A. При выполнении свертки к B мы все еще не видели входного потока, генерируемого C, так что мы не можем быть уверены в том, что имеем дело с телом продукции $A \to B\ C$. Таким образом, мы все еще не можем вычислить значение $B.i$, поскольку нет гарантии, что следует использовать правило, связанное именно с данной продукцией.

Возможно, решение состоит в том, чтобы подождать свертки к C и убедиться в том, что мы должны свернуть $B\ C$ к A. Однако даже тогда мы не знаем наследуемых атрибутов A, поскольку даже после свертки может быть неизвестно, какое именно тело продукции содержит это A. Если опять воспользоваться откладыванием вычисления $B.i$, то в конечном итоге мы придем к тому, что мы не можем принять ни одного решения, пока не будет проанализирована вся входная строка. По сути, это означает применение стратегии “сначала построить дерево разбора, а затем выполнить трансляцию”.

связанным с грамматическими символами, которые отсутствуют в данной продукции. Однако, так как мы реализуем действия с использованием стека LR-синтаксического анализа, необходимые атрибуты всегда будут доступны посредством известных позиций записей ниже в стеке.

Пример 5.25. Предположим, имеется LL-грамматика с продукцией $A \to B\ C$ и наследуемый атрибут $B.i$ вычисляется с использованием наследуемого атрибута $A.i$ по формуле $B.i = f(A.i)$. Значит, интересующий нас фрагмент СУТ имеет вид

$$A \to \{B.i = f(A.i);\}\ B\ C$$

Почему работают маркеры

Маркеры — это нетерминалы, порождающие только ϵ и встречающиеся только один раз в телах всех продукций. Мы не приводим здесь формального доказательства того, что нетерминалы-маркеры могут быть добавлены в любые позиции в телах продукций LL-грамматики и получившаяся в результате грамматика останется LR-грамматикой. Интуитивно это поясняется следующим образом. Если грамматика принадлежит классу LL, то можно определить, что строка w во входном потоке порождается нетерминалом A, в порождении, начинающемся с продукции $A \to \alpha$, просматривая только один первый символ w (или следующий символ, если $w = \epsilon$). Таким образом, при восходящем синтаксическом анализе w тот факт, что префикс w должен быть свернут в α, а затем в S, становится известен, как только во входном потоке появляется начало строки w. В частности, если мы вставим маркер где угодно в α, то LR-состояния включат тот факт, что в некотором месте должен иметься данный маркер, и выполнят свертку ϵ в маркер в соответствующей точке входного потока.

Введем маркер M с наследуемым атрибутом $M.i$ и синтезируемым атрибутом $M.s$. Первый является копией $A.i$, а последний — $B.i$. СУТ записывается как

$$A \to M\ B\ C$$
$$M \to \{M.i = A.i;\ M.s = f\,(M.i)\,;\}$$

Заметим, что атрибут $A.i$ не доступен правилу для M, но в действительности можно так расположить наследуемые атрибуты нетерминалов, таких как A, чтобы они находились в стеке непосредственно под тем местом, где позже будет выполнена свертка в A. Таким образом, при свертке ϵ в M $A.i$ находится непосредственно под ним, откуда и может быть считано. Значение $M.s$, находящееся в стеке слева вместе с M, в действительности представляет собой $B.i$ и находится ниже, сразу за тем местом, где позже будет выполнена свертка в B. □

Пример 5.26. Давайте превратим СУТ на рис. 5.28 в СУТ, которая может использоваться при LR-синтаксическом анализе переделанной грамматики. Введем маркер M перед C и маркер N перед S_1, так что лежащая в основе СУТ грамматика принимает вид

$$S \to \textbf{while}\ (M\ C)\ N\ S_1$$
$$M \to \epsilon$$
$$N \to \epsilon$$

Перед тем как рассмотреть действия, связанные с маркерами M и N, сформулируем "гипотезу индукции" о том, где хранятся атрибуты.

1. Ниже всего тела while-продукции — т.е. ниже **while** в стеке — находится наследуемый атрибут $S.next$. Мы можем не знать, какой нетерминал или состояние синтаксического анализатора связано с этой записью в стеке, но мы можем быть уверены, что в фиксированной позиции этой записи имеется поле, хранящее $S.next$ до того, как мы начнем распознавать, что же именно порождено этим S.

2. Наследуемые атрибуты $C.true$ и $C.false$ находятся в стеке в записи, расположенной непосредственно под записью для C. Поскольку предполагается, что грамматика принадлежит классу LL, наличие **while** во входном потоке гарантирует, что может быть распознана единственная продукция, а именно — while-продукция, так что можно гарантировать, что M будет находиться в стеке непосредственно под C, а запись M будет содержать наследуемые атрибуты C.

3. Аналогично наследуемый атрибут $S_1.next$ должен находиться в стеке непосредственно под S_1, так что можно разместить этот атрибут в записи для N.

4. Синтезируемый атрибут $C.code$ будет находиться в записи для C. Как всегда, когда в качестве значения атрибута выступает длинная строка, на практике в записи хранится указатель на (объект, представляющий) строку, в то время как сама строка располагается вне стека.

5. Аналогично синтезируемый атрибут $S_1.next$ находится в записи для S_1.

Проследим теперь за процессом синтаксического анализа конструкции while. Предположим, что запись, в которой хранится $S.next$, находится на вершине стека, а очередной входной символ — терминал **while**. Перенесем этот терминал в стек. Теперь определенно известно, что распознаваемая продукция представляет собой while-продукцию, так что LR-синтаксический анализатор может перенести (и определить, что следующим шагом должна быть свертка ϵ в M. Состояние стека в этот момент показано на рис. 5.36. На этом рисунке приведены также действия, связанные со сверткой в M. Здесь создаются значения $L1$ и $L2$, которые хранятся в полях M-записи. В этой же записи располагаются поля для $C.true$ и $C.false$. Эти атрибуты должны находиться во втором и третьем полях записи для согласованности с другими записями стека, которые могут находиться ниже C в других контекстах и также предоставлять эти атрибуты C. Действие завершается присваиванием значений атрибутам $C.true$ и $C.false$, одному — только что сгенерированного значения $L2$, а другому — значения из записи, располагающейся ниже в стеке, в которой, как мы знаем, может быть найдено значение $S.next$.

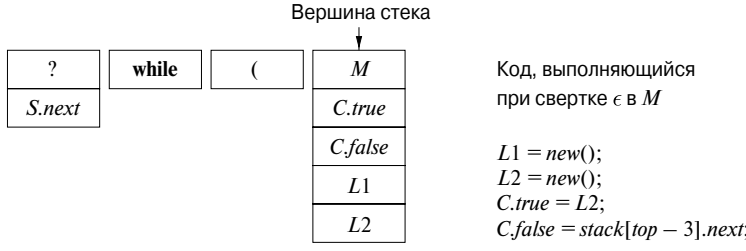

Рис. 5.36. Стек LR-синтаксического анализа после свертки ϵ в M

Мы предполагаем, что все очередные символы входного потока корректно сворачиваются в C. Следовательно, синтезированный атрибут $C.code$ размещается в записи для C. Это изменение в стеке показано на рис. 5.37; на нем показаны также несколько записей, которые позже оказываются в стеке над записью C.

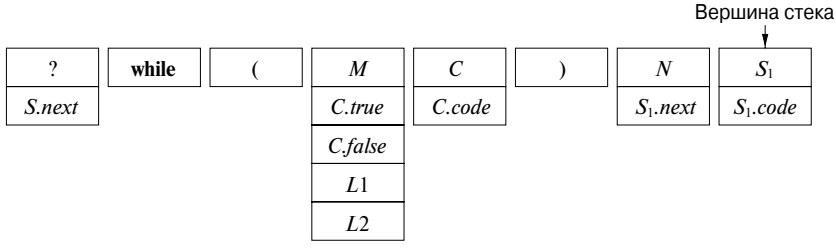

Рис. 5.37. Стек непосредственно перед сверткой тела while-продукции в S

Продолжая распознавание конструкции while, синтаксический анализатор должен обнаружить во входном потоке символ), который он поместит в стек в соответствующую запись. В этот момент синтаксический анализатор, в силу принадлежности грамматики к классу LL знающий, что он работает с конструкцией while, выполнит свертку ϵ в N. Единственные данные, связанные с N, — это наследуемый атрибут $S_1.next$. Заметим, что этот атрибут должен находиться в записи для N, потому что она находится непосредственно под записью для S_1. Выполняемый при свертке код вычисляет значение $S_1.next$ следующим образом:

$$S_1.next = stack\,[top - 3]\,.L1;$$

Это действие обращается к третьей записи под N (которая находится в момент выполнения кода на вершине стека) и получает значение $L1$.

Затем синтаксический анализатор выполняет свертку некоторого префикса остающегося входного потока в нетерминал S (который мы везде указываем с индексом, как S_1, чтобы отличать его от нетерминала S в заголовке продукции). Вычисленное при этом значение $S_1.code$ находится в записи стека для S_1. Этот шаг приводит нас в состояние, показанное на рис. 5.37.

В этот момент синтаксический анализатор выполняет свертку всей части стека от **while** до S_1 в S. Вот код, который выполняется при этой свертке:

$$
\begin{aligned}
&\mathit{tempCode} = \textbf{label} \parallel \mathit{stack}[\mathit{top}-4].L1 \parallel \mathit{stack}[\mathit{top}-3].\mathit{code} \parallel \\
&\quad \textbf{label} \parallel \mathit{stack}[\mathit{top}-4].L2 \parallel \mathit{stack}[\mathit{top}].\mathit{code};\\
&\mathit{top} = \mathit{top} - 5;\\
&\mathit{stack}[\mathit{top}].\mathit{code} = \mathit{tempCode};
\end{aligned}
$$

Иначе говоря, мы собираем значение $S.code$ в переменной *tempCode*. Это обычный код, состоящий из меток $L1$ и $L2$, кода C и кода S_1. Затем выполняется снятие со стека, так что S оказывается в стеке на том месте, где находился терминал **while**. Код S размещается в поле *code* указанной записи, где он может рассматриваться как синтезируемый атрибут $S.code$. Заметим, что в процессе рассмотрения нашего примера мы нигде не показывали работу с LR-состояниями, которые также должны находиться в стеке в поле, в которое мы вносим грамматические символы. □

5.5.5 Упражнения к разделу 5.5

Упражнение 5.5.1. Реализуйте каждое из ваших СУО из упражнения 5.4.4 в виде синтаксического анализатора, работающего методом рекурсивного спуска, как это было сделано в разделе 5.5.1.

Упражнение 5.5.2. Реализуйте каждое из ваших СУО из упражнения 5.4.4 в виде синтаксического анализатора, работающего методом рекурсивного спуска, как это было сделано в разделе 5.5.2.

Упражнение 5.5.3. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LL-синтаксического анализатора, как это было сделано в разделе 5.5.3, с генерацией кода “на лету”.

Упражнение 5.5.4. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LL-синтаксического анализатора, как это было сделано в разделе 5.5.3, но в этом случае код (или указатель на код) хранится в стеке.

Упражнение 5.5.5. Реализуйте каждое из ваших СУО из упражнения 5.4.4 при помощи LR-синтаксического анализатора, как это было сделано в разделе 5.5.4.

Упражнение 5.5.6. Реализуйте ваше СУО из упражнения 5.2.4 так, как это было сделано в разделе 5.5.1. Будет ли чем-то отличаться реализация в стиле раздела 5.5.2?

5.6 Резюме к главе 5

- *Наследуемые и синтезируемые атрибуты*. Синтаксически управляемые определения могут использовать два типа атрибутов. Синтезируемые атрибуты в узле дерева разбора вычисляются с использованием атрибутов

в дочерних узлах. Наследуемый атрибут в узле вычисляется с использованием атрибутов в родительском узле и/или в узлах-братьях.

- *Графы зависимостей*. Для данного дерева разбора и СУО мы проводим дуги между экземплярами атрибутов, связанными с каждым узлом дерева разбора, для указания того, что атрибут, на который указывает стрелка дуги, вычисляется с использованием значения атрибута, из которого выходит данная дуга.

- *Циклические определения*. В проблематичных СУО можно найти деревья разбора, для которых не существует порядка, в котором можно вычислить все атрибуты всех узлов. Такие деревья разбора содержат циклы в связанных с ними графах зависимостей. Выяснение, имеет ли СУО такие циклические графы зависимостей, — очень сложная задача.

- *S-атрибутные определения*. В S-атрибутном СУО все атрибуты синтезируемые.

- *L-атрибутные определения*. В L-атрибутном СУО атрибуты могут быть наследуемыми или синтезируемыми. Однако наследуемые атрибуты в узле дерева разбора могут зависеть только от наследуемых атрибутов в родительском узле и от любых атрибутов в братских узлах, находящихся слева от рассматриваемого.

- *Синтаксические деревья*. Каждый узел синтаксического дерева представляет конструкцию; дочерние узлы представляют значащие компоненты конструкции.

- *Реализация S-атрибутных СУО*. S-атрибутное СУО может быть реализовано при помощи СУТ, в которой все действия находятся в конце продукций (постфиксная СУТ). Действия вычисляют синтезируемые атрибуты заголовков продукций с использованием синтезируемых атрибутов символов их тел. Если лежащая в основе грамматика принадлежит классу LR, то такая СУТ может быть реализована в стеке LR-синтаксического анализатора.

- *Устранение левой рекурсии из СУТ*. Если СУТ содержит только побочные действия (без вычисления атрибутов), то применим стандартный алгоритм устранения левой рекурсии из грамматики, при использовании которого действия рассматриваются так, как если бы это были терминалы. При вычислении атрибутов устранение левой рекурсии возможно, если СУТ является постфиксной.

- *Реализация L-атрибутного СУО в процессе синтаксического анализа методом рекурсивного спуска*. Если имеется L-атрибутное определение на основе грамматики, к которой применим нисходящий синтаксический анализ, то для реализации трансляции можно построить синтаксический анализатор, работающий методом рекурсивного спуска без возврата. Наследуемые атрибуты становятся аргументами функций для их нетерминалов, а синтезируемые атрибуты этими функциями возвращаются.

- *Реализация L-атрибутного СУО на основе LL-грамматики*. Каждое L-атрибутное определение с лежащей в его основе LL-грамматикой может быть реализовано одновременно с синтаксическим анализом. Записи для хранения синтезируемых атрибутов нетерминалов размещаются в стеке под записями нетерминалов, в то время как наследуемые атрибуты нетерминалов хранятся в стеке вместе с нетерминалами. Записи действий также размещаются в стеке для выполнения вычислений атрибутов в соответствующие моменты времени.

- *Реализация L-атрибутного СУО на основе LL-грамматики при восходящем синтаксическом анализе*. L-атрибутное определение с лежащей в его основе LL-грамматикой может быть преобразовано в трансляцию на основе LR-грамматики и трансляцию, выполняемую в процессе восходящего синтаксического анализа. Преобразование грамматики включает нетерминалы-маркеры, которые появляются в стеке восходящего синтаксического анализатора и хранят наследуемые атрибуты нетерминалов, находящихся в стеке над ними. Синтезируемые атрибуты хранятся в стеке вместе с их нетерминалами.

5.7 Список литературы к главе 5

Синтаксически управляемые определения представляют собой вид индуктивных определений, в которых индукция проявляется в синтаксической структуре. Как таковые они давно неформально используются в математике. Их применение к языкам программирования началось с Algol 60.

Идея синтаксического анализатора, который вызывает семантические действия, может быть найдена у Сеймельсона (Samelson) и Бауэра (Bauer) [8], а также у Брукера (Brooker) и Морриса (Morris) [1]. Айронс (Irons) [2] создал один из первых синтаксически управляемых компиляторов с использованием синтезируемых атрибутов. Класс L-атрибутных определений введен в [6].

Наследуемые атрибуты, графы зависимостей и проверка цикличности СУО (т.е. выяснение, существует ли некоторое дерево разбора, для которого не существует порядка вычисления атрибутов) обязаны своим происхождением Кнуту

(Knuth) [5]. Джазаери (Jazayeri), Огден (Ogden) и Раундс (Rounds) [3] показали, что проверка цикличности требует времени, экспоненциально зависящего от размера СУО.

Генераторы синтаксических анализаторов, такие как `Yacc` [4] (см. также список литературы к главе 4), поддерживают вычисление атрибутов в процессе синтаксического анализа.

Обзор Паакки (Paakki) [7] может служить хорошей отправной точкой для литературного поиска по синтаксически управляемым определениям и трансляциям.

1. Brooker, R. A. and D. Morris, “A general translation program for phrase structure languages”, *J. ACM* **9**:1 (1962), pp. 1–10.

2. Irons, E. T., “A syntax directed compiler for Algol 60”, *Comm. ACM* **4**:1 (1961), pp. 51–55.

3. Jazayeri, M., W. F. Ogden, and W. C. Rounds, “The intrinsic exponential complexity of the circularity problem for attribute grammars”, *Comm. ACM* **18**:12 (1975), pp. 697–706.

4. Johnson, S. C., “Yacc — Yet Another Compiler Compiler”, Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975. Доступно по адресу `http://dinosaur.compilertools.net/yacc/`.

5. Knuth, D. E., “Semantics of context-free languages”, *Mathematical Systems Theory* **2**:2 (1968), pp. 127–145. См. также *Mathematical Systems Theory* **5**:1 (1971), pp. 95–96.

6. Lewis, P. M. II, D. J. Rosenkrantz, and R. E. Stearns, “Attributed translations”, *J. Computer and System Sciences* **9**:3 (1974), pp. 279–307.

7. Paakki, J., “Attribute grammar paradigms — a high-level methodology in language implementation”, *Computing Surveys* **27**:2 (1995), pp. 196–255.

8. Samelson, K. and F. L. Bauer, “Sequential formula translation”, *Comm. ACM* **3**:2 (1960), pp. 76–83.