Глава 1
ВВЕДЕНИЕ

В этой главе будут рассмотрены основные средства, необходимые для реализации компилятора. Программы обычно вводятся в компьютер в виде текста, то есть последовательности символов. Представление программы в виде текста называется *конкретным синтаксисом* (concrete syntax). Конкретный синтаксис используется для записи и анализа программ. Внутри компилятора используются *абстрактные синтаксические деревья* (*AST*, abstract syntax tree), которые формируют представления программ, позволяющие компилятору выполнять необходимые операции. Процесс преобразования конкретного синтаксиса в абстрактный называется *парсингом*, а программный код, в котором он реализуется, — *парсером*. Теория и реализация парсинга программ в книге не рассматривается. Читателям, интересующимся данной процедурой на достаточно серьезном уровне, стоит обратиться к книге (Aho et al., 2006)[1]. Парсер включается в исходный код для преобразования конкретного синтаксиса в абстрактный.

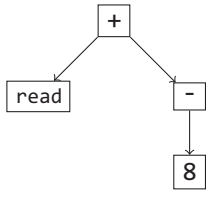
В зависимости от языка программирования, на котором написан компилятор, существует много разных способов представления абстрактных синтаксических деревьев внутри компилятора. Для представления AST в книге будет использоваться конструкция `struct` языка Racket (раздел 1.1). Грамматики будут использоваться для определения абстрактного синтаксиса языков программирования (раздел 1.2), а поиск по шаблону — для поиска отдельных узлов AST (раздел 1.3). Рекурсивные функции будут использоваться для построения и разложения AST (раздел 1.4). В этой главе приводится краткое вводное описание этих компонентов.

[1] Ахо А. и др. «Компиляторы: принципы, технологии и инструменты».

1.1. Абстрактные синтаксические деревья

Компиляторы используют абстрактные синтаксические деревья для представления программ, поскольку, обрабатывая фрагмент программы, они должны знать в том числе, какой языковой конструкции он соответствует, какие элементы содержит. Рассмотрим программу в левой части схемы и диаграмму AST справа (1.1). Программа выполняет операцию сложения, состоящую из двух элементов: операции чтения и операции изменения знака. Операция изменения знака включает еще один элемент – целочисленную константу 8. Используя дерево для представления программы, можно легко переходить от фрагмента программы к его составным элементам.

```
(+ (read) (- 8))
```



(1.1)

Для описания AST будет использоваться стандартная терминология: каждый прямоугольник на схеме выше называется *узлом* (node). Линии со стрелками соединяют узлы с их *потомками*, которые также являются узлами (дочерними). Узел верхнего уровня называется *корневым*. У каждого узла, кроме корневого, имеется *родитель* (то есть узел, по отношению к которому этот узел является дочерним). Если у узла нет дочерних узлов, такой узел называется *листовым*; в противном случае он называется *внутренним* узлом.

Для каждой разновидности узлов мы определим *структуру* (`struct`) Racket. Для этой главы достаточно всего двух разновидностей узлов: для целочисленных констант (литералов) и для примитивных операций. Ниже приведено определение `struct` для целочисленных констант[1].

```
(struct Int (value))
```

Целочисленный узел содержит только целое значение. Мы будем использовать соглашение, в соответствии с которым имена структур (такие, как `Int`) записываются с прописной буквы. Чтобы создать узел AST для целого числа 8, используется запись `(Int 8)`.

```
(define eight (Int 8))
```

Значение, созданное обозначением `(Int 8)`, называется *экземпляром* структуры `Int`.

[1] Все структуры AST определены в файле utilities.rkt в прилагаемом коде.

Определение структуры для примитивных операций выглядит так:

```
(struct Prim (op args))
```

Узел примитивной операции включает обозначение оператора `op` и список дочерних аргументов `args`. Например, при создании AST для изменения знака числа 8 используется следующая конструкция:

```
(define neg-eight (Prim '- (list eight)))
```

У примитивных операций может быть нуль и более дочерних узлов. У оператора `read` их нуль:

```
(define rd (Prim 'read '()))
```

Оператор сложения имеет два дочерних узла:

```
(define ast1_1 (Prim '+ (list rd neg-eight)))
```

В отношении структуры `Prim` было принято проектировочное решение. Вместо того чтобы использовать одну структуру для нескольких разных операций (`read`, `+` и `-`), можно было определить отдельную структуру для каждой операции:

```
(struct Read ())
(struct Add (left right))
(struct Neg (value))
```

Почему же мы используем только одну структуру? Потому что разные части компилятора могут использовать один и тот же код для разных примитивных операций, и этот код с таким же успехом можно написать с использованием одной структуры.

Чтобы откомпилировать такую программу, как (1.1), необходимо знать, что с корневым узлом связана операция сложения, и иметь возможность обратиться к двум его дочерним узлам. Для подобных запросов в Racket существует механизм поиска по шаблону, который более подробно рассматривается в разделе 1.3.

Конкретный синтаксис программы часто приводится даже в том случае, когда на самом деле имеется в виду AST, потому что конкретный синтаксис более компактный. Мы рекомендуем всегда рассматривать программы как абстрактные синтаксические деревья.

1.2. Грамматики

Язык программирования можно рассматривать как *множество* программ. Это множество бесконечно (то есть всегда можно создать программу большего

размера), так что язык невозможно описать простым перечислением всех его программ. Вместо этого записывается набор правил построения программ — *контекстно-свободная грамматика*. Грамматики часто используются для определения конкретного синтаксиса языка, но они также могут использоваться для описания абстрактного синтаксиса. Правила записываются как разновидность формы Бэкуса — Наура (BNF) (Backus et al., 1960; Knuth, 1964). Для примера можно описать маленький язык $\mathcal{L}_{Int}$, состоящий из целых чисел и арифметических операций.

Первое правило грамматики для абстрактного синтаксиса $\mathcal{L}_{Int}$ гласит, что экземпляр структуры `Int` является выражением:

$$exp ::= (\texttt{Int}\ int) \tag{1.2}$$

У каждого правила есть левая и правая сторона. Если имеется узел AST, соответствующий правой стороне, это означает, что его можно классифицировать в соответствии с левой стороной. Символы, записанные моноширинным шрифтом (например, `Int`), называются *терминальными*; они должны буквально входить в программу, чтобы правило было применимо. В наших грамматиках не упоминаются *пропуски* (то есть такие символы-ограничители, как пробелы, табуляции и новые строки.) Пропуски могут вставляться между символами для устранения неоднозначности и улучшения удобочитаемости. Имя, определяемое правилами грамматики (такое, как *exp*), является *нетерминальным*. Имя `int` тоже является нетерминальным, но вместо того чтобы определять его правилом грамматики, мы определяем его следующим описанием. `int` представляет собой последовательность десятичных цифр (от 0 до 9), которая может начинаться с необязательного знака «–» (минус, для отрицательных чисел) и позволяет представлять целые числа в диапазоне от -2^{62} до $2^{62} - 1$. Такое определение открывает возможность представления целых чисел в 63-разрядном виде, что немного упрощает вычисления. Таким образом, эти числа соответствуют типу данных Racket `fixnum` на 64-разрядных машинах.

Второе правило грамматики — операция `read`, которая получает целое число от пользователя программы.

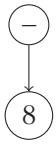
$$exp ::= (\texttt{Prim 'read ()}) \tag{1.3}$$

Третье правило классифицирует изменение знака узла *exp* как *exp*.

$$exp ::= (\texttt{Prim '-}\ (exp)) \tag{1.4}$$

Эти правила можно применить для классификации AST, принадлежащих языку $\mathcal{L}_{Int}$. Например, по правилу (1.2) `(Int 8)` является *exp* (выражением), и тогда по правилу (1.4) следующее AST тоже является *exp*.

```
(Prim '- ((Int 8)))
```



(1.5)

Следующие два правила грамматики предназначены для сложения и вычитания выражений:

$$exp ::= \texttt{(Prim '+ (}exp\ exp\texttt{))} \tag{1.6}$$

$$exp ::= \texttt{(Prim '- (}exp\ exp\texttt{))} \tag{1.7}$$

Теперь можно проверить, что AST (1.1) является *exp* в $\mathcal{L}_{Int}$. Мы знаем, что `(Prim 'read ())` является *exp* согласно правилу (1.3), и мы уже классифицировали `(Prim '- ((Int 8)))` как *exp*, поэтому применение правила (1.6) доказывает, что

```
(Prim '+ ((Prim 'read ()) (Prim '- ((Int 8)))))
```

также является *exp* в языке $\mathcal{L}_{Int}$.

Если имеется AST, к которому эти правила неприменимы, то AST не принадлежит $\mathcal{L}_{Int}$. Например, программа `(* (read) 8)` не принадлежит $\mathcal{L}_{Int}$, потому что правила для оператора * не существует. Когда мы определяем язык грамматикой, язык включает только те программы, которые соответствуют правилам грамматики.

Последнее правило грамматики $\mathcal{L}_{Int}$ утверждает, что существует узел `Program` для пометки верхнего уровня всей программы:

$$\mathcal{L}_{Int} ::= \texttt{(Program '()}\ exp\texttt{)}$$

Структура `Program` определяется следующим образом:

```
(struct Program (info body)),
```

где `body` — выражение. В следующих главах часть `info` будет использоваться для хранения дополнительной информации, а пока это просто пустой список.

На практике часто существует множество правил грамматики с одинаковой левой стороной, но разными правыми сторонами, как в правилах *exp* в грамматике $\mathcal{L}_{Int}$. Сокращенная форма записи позволяет объединить несколько правых сторон в одно правило, разделив их символом | (вертикальная черта).

Конкретный синтаксис $\mathcal{L}_{Int}$ представлен на илл. 1.1, а абстрактный синтаксис $\mathcal{L}_{Int}$ — на илл. 1.2. Функция `read-program`, содержащаяся в файле utilities.rkt из прилагаемого кода, читает программу из файла (последовательность символов в конкретном синтаксисе Racket) и преобразует ее в абстрактное синтаксическое

дерево. Чтобы узнать больше, перейдите к описанию `read-program` в приложении A.2.

```
type  ::= Integer
exp   ::= int | (read) | (- exp) | (+ exp exp) | (- exp exp)
L_Int ::= exp
```

Илл. 1.1. Конкретный синтаксис $\mathcal{L}_{Int}$

```
type  ::= Integer
exp   ::= (Int int) | (Prim 'read ())
        | (Prim '- (exp)) | (Prim '+ (exp exp)) | (Prim '- (exp exp))
L_Int ::= (Program '() exp)
```

Илл. 1.2. Абстрактный синтаксис $\mathcal{L}_{Int}$

1.3. Поиск по шаблону

Как упоминалось в разделе 1.1, компиляторам часто требуется доступ к частям узла AST. Для обращения к частям значений Racket предоставляет функциональность поиска по шаблону. Рассмотрим следующий пример:

```
(match ast1_1
  [(Prim op (list child1 child2))
    (print op)])
```

В этом примере форма `match` проверяет, является ли AST (1.1) бинарным оператором, и связывает его части с тремя переменными шаблона `op`, `child1` и `child2`. В общем случае условие `match` состоит из *шаблона* и *тела*. Шаблоны рекурсивно определяются как переменная шаблона, имя структуры, за которой следует шаблон для каждого из аргументов структуры, или S-выражение (символ, список и т. д.) (полное описание `match` см. в разделе 12 документа Racket Guide[1] и главе 9 Racket Reference[2].) Тело условия `match` может содержать произвольный код Racket. Переменные шаблона могут использоваться в области видимости тела, как `op` в `(print op)`.

Форма `match` может иметь несколько условий, как в следующей функции `leaf`, проверяющей, что узел $\mathcal{L}_{Int}$ является листовым в AST. `match` обрабатывает усло-

[1] См. https://docs.racket-lang.org/guide/match.html.

[2] См. https://docs.racket-lang.org/reference/match.html.

вия по порядку, проверяя, существует ли совпадение шаблона во входном AST. После этого выполняется тело первого условия, для которого найдено совпадение. Вывод `leaf` для нескольких AST приведен в правой части:

```
(define (leaf arith)
  (match arith
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e1)) #f]
    [(Prim '+ (list e1 e2)) #f]
    [(Prim '- (list e1 e2)) #f]))

(leaf (Prim 'read '()))                    #t
(leaf (Prim '- (list (Int 8))))            #f
(leaf (Int 8))                             #t
```

При построении выражений `match` мы используем грамматическое определение, чтобы установить, в каких нетерминальных элементах должно быть совпадение, после чего проверяем, что (1) имеется одно условие для каждой альтернативы этого нетерминального элемента и (2) шаблон в каждом условии соответствует аналогичной правой части правила грамматики. Для `match` в функции `leaf` мы обращаемся к грамматике $\mathcal{L}_{Int}$, показанной на илл. 1.2. Нетерминальное имя *exp* имеет четыре альтернативы, поэтому `match` содержит четыре условия. Шаблон в каждом условии соответствует правой части правила грамматики. Например, шаблон `(Prim '+ (list e1 e2))` соответствует правой части `(Prim '+ (`*exp exp*`))`. При переходе от грамматики к шаблонам замените нетерминальные имена (такие, как *exp*) переменными шаблонов по своему выбору (например, `e1` и `e2`).

1.4. Рекурсивные функции

Программы рекурсивны по своей природе. Например, выражение часто состоит из нескольких меньших подвыражений. Таким образом, можно рассматривать рекурсивную функцию как естественный метод обработки всей программы. В качестве первого примера такой рекурсивной функции определим функцию `is_exp`, как показано на илл. 1.3; она получает произвольное значение и определяет, является ли оно выражением в $\mathcal{L}_{Int}$. Говорят, что функция определяется *структурной рекурсией*, если для ее определения используется последовательность условий `match`, соответствующих грамматике, а тело каждого условия содержит рекурсивный вызов к каждому дочернему узлу[1].

[1] Этот принцип структурирования кода в соответствии с определением данных был предложен в книге *How to Design Programs* Фелляйзена и др. (Felleisen et al.) («Как проектировать программы. Введение в программирование и компьютерное вычисление»).

Иллюстрация 1.3 также содержит определение функции `is_Lint`, устанавливающей, является ли AST программой в языке $\mathcal{L}_{Int}$. В общем случае можно написать одну рекурсивную функцию для обработки каждого нетерминального элемента в грамматике. Из двух примеров в нижней части листинга первый принадлежит $\mathcal{L}_{Int}$, а второй нет.

```
(define (is_exp ast)
  (match ast
    [(Int n) #t]
    [(Prim 'read '()) #t]
    [(Prim '- (list e)) (is_exp e)]
    [(Prim '+ (list e1 e2))
      (and (is_exp e1) (is_exp e2))]
    [(Prim '- (list e1 e2))
      (and (is_exp e1) (is_exp e2))]
    [else #f]))

(define (is_Lint ast)
  (match ast
    [(Program '() e) (is_exp e)]
    [else #f]))

(is_Lint (Program '() ast1_1)
(is_Lint (Program '()
      (Prim '* (list (Prim 'read '())
                  (Prim '+ (list (Int 8)))))))
```

Илл. 1.3. Пример рекурсивных функций для $\mathcal{L}_{Int}$. Эти функции проверяют, принадлежит ли AST языку $\mathcal{L}_{Int}$

1.5. Интерпретаторы

Поведение программы определяется спецификацией языка программирования. Например, язык Scheme определяется в докладе Спербера и др. (Sperber et al.) (2009). Язык Racket определяется в его справочном руководстве (Flatt and PLT, 2014). В этой книге для определения каждого рассматриваемого языка будут использоваться интерпретаторы. Интерпретатор, спроектированный как определение языка, называется *определительным интерпретатором* (definitional interpreter) (Reynolds, 1972). Начнем с создания определительного интерпретатора для языка $\mathcal{L}_{Int}$. Этот интерпретатор служит вторым примером структурной рекурсии. Определение функции `interp_Lint` приведено на илл. 1.4. Тело функции содержит вызов `match` для входной программы, за которым следует вызов вспомогательной функции `interp_exp`. В свою очередь, эта функция содержит отдельное условие `match` для каждого правила грамматики для выражений $\mathcal{L}_{Int}$.

```
(define (interp_exp e)
  (match e
    [(Int n) n]
    [(Prim 'read '())
     (define r (read))
     (cond [(fixnum? r) r]
           [else (error 'interp_exp "read expected an integer" r)])]
    [(Prim '- (list e))
     (define v (interp_exp e))
     (fx- 0 v)]
    [(Prim '+ (list e1 e2))
     (define v1 (interp_exp e1))
     (define v2 (interp_exp e2))
     (fx+ v1 v2)]
    [(Prim '- (list e1 e2))
     (define v1 (interp_exp e1))
     (define v2 (interp_exp e2))
     (fx- v1 v2)]))

(define (interp_Lint p)
  (match p
    [(Program '() e) (interp_exp e)]))
```

Илл. 1.4. Интерпретатор для языка $\mathcal{L}_{Int}$

Рассмотрим результат интерпретации нескольких программ $\mathcal{L}_{Int}$. Следующая программа складывает два целых числа:

```
(+ 10 32)
```

Результат равен 42 – ответ на главный вопрос жизни, Вселенной и всего остального: 42![1] Мы записали эту программу в конкретном синтаксисе, тогда как разобранный абстрактный синтаксис имеет вид:

```
(Program '() (Prim '+ (list (Int 10) (Int 32))))
```

Следующая программа демонстрирует, что выражения могут быть вложенными — в примере ниже используются вложенные операции сложения и изменения знака.

```
(+ 10 (- (+ 12 20)))
```

Как выглядит результат выполнения этой программы?

[1] Адамс Д. «Автостопом по Галактике».

Как уже говорилось, язык $\mathcal{L}_{Int}$ не поддерживает произвольные большие числа, а работает только с 63-разрядными целыми числами, поэтому арифметические операции $\mathcal{L}_{Int}$ интерпретируются с использованием арифметики `fixnum` в Racket. Допустим, имеется число

$$n = 999999999999999999,$$

которое действительно помещается в 63 бита. Что произойдет при выполнении следующей программы в интерпретаторе?

```
(+ (+ (+ n n) (+ n n)) (+ (+ n n) (+ n n)))))
```

Программа выдает ошибку:

```
fx+: result is not a fixnum
```

Будем считать, что если при запуске определительного интерпретатора для программы выдается ошибка, то поведение этой программы *не определено* (кроме ошибок `trapped-error`). Компилятор для языка не имеет никаких обязательств в отношении программ с неопределенным поведением; он не обязан создавать исполняемый файл, а если такой файл все же создается, то он может делать все что угодно. С другой стороны, для ошибок `trapped-error` компилятор должен построить исполняемый файл и сообщать о возникновении ошибки. Сигналом об ошибке является завершение работы с кодом возврата 255. Интерпретаторы в главах 9 и 10, а также в разделе 6.10 используют `trapped-error`.

Последняя функция языка $\mathcal{L}_{Int}$ — операция `read` — запрашивает у пользователя программы целое число. Вспомните, что программа (1.1) запрашивает целое число и вычитает из него 8. Таким образом, при выполнении

```
(interp_Lint (Program '() ast1_1))
```

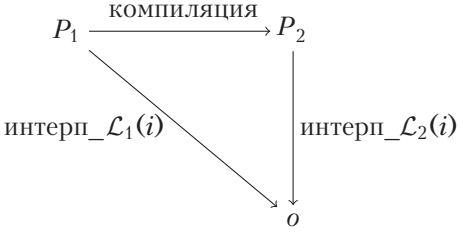
для ввода `50` результат будет равен `42`.

Мы включили в $\mathcal{L}_{Int}$ операцию `read`, чтобы какой-нибудь хитрый студент не мог реализовать компилятор $\mathcal{L}_{Int}$, который просто запускает интерпретатор во время компиляции для получения вывода, а затем генерирует тривиальный код для создания такого вывода[1].

Задача компилятора – преобразовать программу на одном языке в программу на другом языке, чтобы выходная программа обладала таким же поведением, как исходная. Этот принцип представлен на следующей диаграмме. Допустим,

[1] Да, один сообразительный студент в первой версии курса именно так и поступил!

имеются два языка, $\mathcal{L}_1$ и $\mathcal{L}_2$, и определительный интерпретатор для каждого языка. Если имеется компилятор, преобразующий программу из языка $\mathcal{L}_1$ в $\mathcal{L}_2$, и некоторая программа P_1 на $\mathcal{L}_1$, компилятор должен преобразовать ее в программу P_2 такую, что интерпретация P_1 и P_2 соответствующими интерпретаторами с одинаковыми входными данными приведет к одинаковым результатам o.



(1.8)

В следующем разделе рассматривается первый пример компилятора.

1.6. Пример компилятора: частичный вычислитель

В этом разделе рассматривается компилятор, который преобразует программы $\mathcal{L}_{Int}$ в более эффективные программы $\mathcal{L}_{Int}$. Компилятор немедленно вычисляет части программы, которые не зависят от ввода, – этот процесс называется *частичным вычислением* (Jones, Gomard, and Sestoft, 1993). Например, следующую программу:

```
(+ (read) (- (+ 5 3)))
```

компилятор преобразует в такую программу:

```
(+ (read) -8)
```

На илл. 1.5 приведен код простого частичного вычислителя для языка $\mathcal{L}_{Int}$. Выводом частичного вычислителя является программа на $\mathcal{L}_{Int}$. На илл. 1.5 структурная рекурсия по *exp* отражена в функции `pe_exp`, тогда как код частичного вычисления операций изменения знака и сложения распределяется по трем вспомогательным функциям: `pe_neg`, `pe_add` и `pe_sub`. Вводом для этих функций является вывод частичного вычисления дочерних узлов. Функции `pe_neg`, `pe_add` и `pe_sub` проверяют, являются ли их аргументы целыми числами, и если да, выполняют соответствующие арифметические вычисления. В противном случае они создают узел AST для арифметической операции.

```
(define (pe_neg r)
  (match r
    [(Int n) (Int (fx- 0 n))]
    [else (Prim '- (list r))]))

(define (pe_add r1 r2)
  (match* (r1 r2)
    [((Int n1) (Int n2)) (Int (fx+ n1 n2))]
    [(_ _) (Prim '+ (list r1 r2))]))

(define (pe_sub r1 r2)
  (match* (r1 r2)
    [((Int n1) (Int n2)) (Int (fx- n1 n2))]
    [(_ _) (Prim '- (list r1 r2))]))

(define (pe_exp e)
  (match e
    [(Int n) (Int n)]
    [(Prim 'read '()) (Prim 'read '())]
    [(Prim '- (list e1)) (pe_neg (pe_exp e1))]
    [(Prim '+ (list e1 e2)) (pe_add (pe_exp e1) (pe_exp e2))]
    [(Prim '- (list e1 e2)) (pe_sub (pe_exp e1) (pe_exp e2))]))

(define (pe_Lint p)
  (match p
    [(Program '() e) (Program '() (pe_exp e))]))
```

Илл. 1.5. Частичный вычислитель для $\mathcal{L}_{Int}$

Чтобы убедиться, что частичный вычислитель работает правильно, можно проверить, генерируют ли программы на его выходе тот же результат, что и программы на входе. Иначе говоря, можно проверить, удовлетворяют ли они схеме на диаграмме (1.8). Следующий код запускает частичный вычислитель для нескольких примеров и тестирует выходную программу. Функции `parse-program` и `assert` определены в приложении A.2.

```
(define (test_pe p)
  (assert "testing pe_Lint"
    (equal? (interp_Lint p) (interp_Lint (pe_Lint p)))))

(test_pe (parse-program `(program () (+ 10 (- (+ 5 3))))))
(test_pe (parse-program `(program () (+ 1 (+ 3 1)))))
(test_pe (parse-program `(program () (- (+ 3 (- 5))))))
```

УПРАЖНЕНИЕ 1.1. Создайте три программы на языке $\mathcal{L}_{Int}$ и проверьте, будет ли результат, полученный при их частичном вычислении функцией `pe_Lint` и последующей интерпретации функцией `interp_Lint`, совпадать с результатом их прямой интерпретации функцией `interp_Lint`.