Введение

В прежние времена под языками подразумевалось исключительно средство общения между людьми, т.е. имелись в виду только естественные языки – русский, немецкий, английский и пр. В начале XX века это представление претерпело серьёзные изменения и в настоящее время под языком понимается всякое средство общения, состоящее из знаковой системы, множества смыслов этой системы и имеющее установленное соответствие между последовательностями знаков и смыслами. Особенно широкий (и все более увеличивающийся) класс составляют языки программирования, изучению которых и посвящен наш курс.

Стремительное развитие вычислительной техники сделало возможной компьютерную обработку текстов, относящихся к самым различным языкам – естественным языкам, языкам формул, языкам программирования.

Существует великое множество программных средств такой обработки. Это и различные редакторы с самыми разными возможностями, и архиваторы, и трансляторы для языков программирования или программы-переводчики для естественных языков.

При обработке текстов возникает ряд задач. Это задачи, связанные с проблемой задания языка, или генерации его цепочек, задачи определения принадлежности некоторого множества слов заданному языку, задачи идентификации цепочек. Для их решения разработано множество методов, причем некоторые из них работают только с определенным классом языков (например, с языками программирования), другие же являются универсальными.

В нашем курсе будут рассматриваться различные методы решения задач генерации и распознавания цепочек языка, основные элементы теории перевода текстов с одного языка на другой, схема трансляции программ и основные этапы трансляции.

Глава 1 Формальные языки и грамматики

1.1 Языки и цепочки символов

1.1.1 Цепочки символов и операции над ними

Сначала дадим все необходимые определения и введём некоторые понятия. Большинство из них достаточно очевидны, но всё же рекомендуется с ними ознакомиться.

- Под *алфавитом* $\sum$ будем понимать непустое конечное множество символов.
- *Цепочка*, или *строка* – это последовательность символов некоторого алфавита, при этом символы в строке могут повторяться. Для цепочки важен её состав, порядок символов и их количество.
- *Длиной* цепочки является количество символов в ней. Например, цепочка x = ’abcab’ имеет длину |x| = 5.
- Две цепочки α и β *совпадают* (равны): $\alpha=\beta$, если они имеют один и тот же состав и порядок символов и их количество. Например, α=’aabc’; β=‘abac‘ – цепочки различны, т.к. порядок символов различен, хотя состав и количество символов в этих цепочках совпадают.
- Цепочка, не содержащая символов, называется *пустой цепочкой*, обозначается через ε или λ.
- Любая последовательно взятая часть цепочки является её *подцепочкой*, для цепочки ‘xy’ подцепочка ‘x’ является *префиксом*, подцепочка ‘y’ – *суффиксом*. Следует заметить, что пустая цепочка является как суффиксом, так и префиксом любой цепочки. Сама цепочка также является своим префиксом и суффиксом. *Собственный суффикс (префикс)* – это *суффикс (префикс)*, не совпадающий со всей цепочкой. Например, для цепочки α=’aabc’ собственными префиксами будут являться цепочки ’a’, ’aa’, ’aab’; собственными суффиксами – цепочки ’c’, ’bc’, ’abc’. Цепочки ’ab’ и ’b’ – просто подцепочки исходной цепочки α.

Над цепочками можно выделить следующие операции:

- *Конкатенацией*, или *сцеплением* цепочек, называется строка, полученная их объединением, например: x = ’abcab’, y = ’cde’, тогда конкатенация цепочек x и y будет иметь вид z = x·y = ’abcabcde’. Свойства: Так как в цепочке важен порядок символов, очевидно, что операция сложения не коммутативна, т.е. в общем случае $\exists\, \alpha,\beta \mid \alpha\beta\neq\beta\alpha$. Конкатенация ассоциативна, т.е. $\forall\alpha,\beta,\gamma\ \ (\alpha\beta)\gamma = \alpha(\beta\gamma)$.
- *Обращение* цепочки x – это запись её символов в обратном порядке, обозначается x^R. Например, для цепочки x = ’abcab’ её обращение x^R = ’bacba’. Свойство операции обращения: $\forall\alpha,\beta\ \ (\alpha\beta)^R= \beta^R\alpha^R$.
- *Итерация* цепочки n раз – это её сцепление (повторение) n раз,

обозначается α^n. Например, если α='ab', то α^3 = 'ababab'.

Для пустой цепочки λ справедливы следующие равенства:

1. $|\lambda| = 0$;
2. $\forall\alpha$: $\lambda\alpha = \alpha\lambda = \alpha$;
3. $\lambda^R = \lambda$;
4. $\forall n \geq 0$: $\lambda^n = \lambda$;
5. $\forall\alpha$: $\alpha^0 = \lambda$.

1.1.2 Понятие языка. Способы задания языков

В общем случае язык – это заданный набор символов и правил, устанавливающих способы комбинации этих символов между собой для записи осмысленных текстов. Основой любого языка является алфавит, определяющий набор допустимых символов.

- Цепочка символов α является *цепочкой над алфавитом* Σ: $\alpha(\Sigma)$, если в неё входят только символы этого алфавита.
- Если $\sum$ – некоторый алфавит, то:
  - $\sum^+$ – множество всех цепочек над алфавитом $\sum$ без пустой цепочки λ.
  - $\sum^*$ – множество всех цепочек над алфавитом $\sum$, включая λ.
- *Языком L* над алфавитом Σ: $L(\Sigma)$ – называют некоторое счётное подмножество цепочек конечной длины из множества всех цепочек над этим алфавитом. Множество цепочек языка не обязано быть конечным и каждая цепочка может иметь сколь угодно большую длину.

Большая часть следующих утверждений основана на теории множеств.

- Для любого языка $L(\Sigma)$ справедливо $L(\Sigma) \subseteq \Sigma^*$.
- Язык $L(\Sigma)$ включает в себя язык $L'(\Sigma)$: $L'(\Sigma) \subseteq L(\Sigma)$, если $\forall\alpha \in L'(\Sigma)$ $\alpha \in L(\Sigma)$. Легко видеть, что это обычное определение включения множеств.
- Два языка совпадают (*эквивалентны*): $L'(\Sigma) = L(\Sigma)$, если $L'(\Sigma) \subseteq L(\Sigma)$ и $L(\Sigma) \subseteq L'(\Sigma)$. И это определение тоже соответствует понятию равенства множеств.
- *Конкатенацией (объединением)* языков L_1 и L_2 называют язык *L*, состоящий из всевозможных сцеплений цепочек языков L_1 и L_2: $L = L_1 \cdot L_2 = \{x \cdot y \mid x \in L_1, y \in L_2\}$.
- *Замыкание Клини*, или *итерация* языка L, обозначается L^* и определяется рекурсивно: 1) $L^0 = \{\lambda\}$, 2) $L^n = L \cdot L^{n-1}$ для $n > 0$, 3) $L^* = \cup L^n$ для всех $n \geq 0$.
- $L^+ = L^* \backslash \{\lambda\}$.
- Язык *L обладает суффиксным (префиксным) свойством*, если никакая цепочка языка не является собственным суффиксом (префиксом) другой цепочки.

Примеры

1. Для языка L={01,11} замыканием Клини будет бесконечное множество цепочек вида L*={λ,01,11,0101,1111,0111,1101, 010101, 010111, 011101, 110101, 110111, 111101, 011111, 111111, …}.

2. Язык L={01,010,111,100} не обладает префиксным свойством, т.к. цепочка ‘01’ является префиксом цепочки ‘010’. В то же время он обладает суффиксным свойством, т.к. в нём нет ни одной цепочки, которая была бы суффиксом какой-то другой цепочки.

Язык состоит из бесконечного множества цепочек символов над некоторым алфавитом, или слов. Но не любая цепочка символов над этим алфавитом принадлежит языку, т.к. существуют правила построения допустимых цепочек для каждого конкретного языка. Указать эти правила – значит задать язык. Существуют три основных способа задания языка:

1. Перечисление всех допустимых цепочек языка (способ формальный, на практике нереализуем, т.к. в общем случае множество цепочек языка бесконечно и перечислить их невозможно).
2. Указание способа порождения цепочек языка (задание грамматики языка) – применение т.н. генератора.
3. Задание метода распознавания цепочек языка – использование распознавателя.

Генераторы и распознаватели являются основными инструментами задания бесконечного языка конечными средствами. Существует определенная классификация языков по их типу, и для каждого класса языков имеются эквивалентные способы задания с помощью генераторов и распознавателей. Далее рассмотрим эти способы более подробно.

В любом языке можно выделить его синтаксис и семантику. Кроме того, трансляторы имеют дело также с лексическими конструкциями (лексемами), которые задаются лексикой языка. Дадим определения этих понятий.

Синтаксис языка – это набор правил, определяющий допустимые конструкции языка. Чаще всего синтаксис языка можно задать в виде строгого набора правил, но полностью это справедливо только для формальных языков.

Семантика языка – это раздел, определяющий значение предложений языка. Семантика определяет «содержание языка», т.е. задаёт значение всех допустимых цепочек языка. Для большинства языков семантика определяется неформальными методами.

Лексика – это словарный запас языка. Лексическая единица (лексема) – конструкция, которая состоит из элементов алфавита языка и не содержит в себе других конструкций. Т.е. лексема может содержать в себе только элементарные символы алфавита и не может содержать других лексем. Например, лексемами русского языка являются слова, а пробелы и знаки препинания представляют собой разделители. Лексемами алгебры

являются числа, знаки математических операций, обозначения функций и т.п. В языках программирования лексемы – это ключевые слова, идентификаторы, константы, метки, знаки операций.

1.1.3 Особенности языков программирования

Языки программирования занимают промежуточное место между формальными и естественными языками. С формальными языками их объединяют строгие синтаксические правила, на основе которых строятся предложения языка. От естественных языков в языки программирования перешли лексические единицы, представляющие основные ключевые слова (чаще это английские слова, но бывают и слова других языков).

Для задания языка программирования необходимо решить три основных вопроса:

1. определить множество допустимых символов языка;
2. определить множество правильных программ языка;
3. задать смысл каждой правильной программы.

С помощью теории формальных языков удается решить (полностью или частично) только первые два вопроса.

Первый вопрос решается легко. Алфавит языка и представляет собой множество его допустимых символов. Для языков программирования это, как правило, тот набор символов, который можно ввести с клавиатуры.

Второй вопрос в теории формальных языков решается только частично. Для всех языков программирования существуют синтаксические правила, но их недостаточно для строгого определения всех возможных синтаксических конструкций. Дополнительные ограничения накладываются семантикой языка и неформально оговариваются для каждого отдельного языка программирования. Это, например, необходимость предварительного описания переменных и функций, необходимость соответствия типов переменных и констант в выражениях, формальных и фактических параметров в описаниях и вызовах процедур и функций, и т.п.

Отсюда следует, что практически все языки программирования, строго говоря, не являются формальными языками. Поэтому во всех трансляторах, кроме синтаксического разбора и анализа предложений языка, дополнительно предусмотрен семантический анализ.

Третий вопрос вообще не относится к теории формальных языков. Для ответа на него нужно использовать другие подходы, например:

1. изложить смысл программы, написанной на языке программирования, на другом языке, более понятном тому, кому адресована программа;

2. использовать для проверки смысла некоторую «идеальную

машину», которая предназначена для проверки программ, написанных на данном языке.

Любой разработчик программ так или иначе использует первый подход – это комментарии в программе, построение её блок-схемы, разработка документации или любое другое описание алгоритма. Все эти способы ориентированы на человека, но универсального способа проверить, насколько описание в действительности соответствует программе, пока не существует. Для изложения программы на языке, понятном машине – в машинных командах – существуют трансляторы.

Второй подход используется при отладке программы; оценку результатов выполнения программы при отладке осуществляет человек.

Разработчикам компиляторов так или иначе приходится решать вопрос о смысле программ.

Во-первых, компилятору для преобразования исходной программы в последовательность машинных команд необходимо иметь представление о том, какая последовательность команд должна соответствовать той или иной части программы. Обычно такие последовательности сопоставляют базовым конструкциям входного языка – используется первый подход к изложению смысла программы.

Во-вторых, многие современные компиляторы позволяют выявить сомнительные с точки зрения смысла места в исходной программе – например, недостижимые операторы, неиспользуемые переменные, неопределенные результаты функций и т.п. Обычно компилятор указывает такие места в виде предупреждений, на которые разработчик может обращать или не обращать внимание. Для этого компилятор должен иметь представление о том, как программа будет выполняться, т.е. используется второй подход.

Но в любом случае осмысление исходной программы закладывает в компилятор его создатель, который руководствуется неформальными методами – как правило, описанием входного языка. В теории формальных языков вопрос о смысле программ не решается.

Итак, возможности трансляторов по проверке осмысленности исходной программы практически равны нулю, поэтому семантические ошибки остаются на совести авторов программ.

Выше, когда шла речь о различных способах задания языков, как наиболее реальные были выделены второй и третий способ, а именно: задание языка с помощью генераторов и распознавателей. В качестве генераторов в общем случае могут использоваться грамматики, а для более узкого класса языков – регулярные выражения. В качестве распознавателей, как правило, применяются автоматы различных типов (в зависимости от класса языка). В первой главе будет рассмотрен способ задания языков с помощью грамматик.

1.1.4 Контрольные вопросы

1. Может ли некоторый символ встречаться в одной цепочке несколько раз? Например, является ли цепочкой ‘aaabcaab’?
2. По каким признакам различаются цепочки? Разными или одинаковыми будут цепочки ‘aaabaacaab’ и ‘abacab’?
3. Сколько собственных суффиксов можно выделить у цепочки, если известно, что её длина равна 6? А собственных префиксов? А сколько можно выделить различных подцепочек длины 2 в цепочке ‘aaabcaab’?
4. Чему равна длина пустой цепочки?
5. Каким из свойств обладает операция конкатенации – коммутативностью или ассоциативностью?
6. Какие из перечисленных тождеств истинны для любых произвольных цепочек символов, а какие нет? 1) $|\alpha\beta|=|\alpha|+|\beta|=|\beta\alpha|$; 2) $\alpha\beta=\beta\alpha$; 3) $|\alpha^R|=|\alpha|$; 4) $(\alpha^2\beta^2)^R=(\beta^R\alpha^R)^2$; 5) $(\alpha^2\beta^2)^R=(\beta^R)^2(\alpha^R)^2$.
7. Пусть Σ – некоторый алфавит. Какое из следующих трёх утверждений верно? 1) $\Sigma^+\subseteq\Sigma^*$; 2) $\Sigma^*\subseteq\Sigma^+$; 3) $\Sigma^*=\Sigma^+$?
8. Пусть Σ – некоторый алфавит, соответственно L(Σ) – язык над этим алфавитом. Какое из следующих утверждений верно? 1)$L(\Sigma) \subseteq \Sigma^*$; 2) $\Sigma^*\subseteq L(\Sigma)$?
9. Какие из следующих языков обладают префиксным (суффиксным) свойством: 1) $\{a^n \cdot b^n \mid n \geq 1\}$; 2) $\{a^n \cdot b \mid n \geq 1\}$; 3) $\{a \cdot b^n \mid n \geq 1\}$; 4) $\{w \mid w \in \{a,b\}^*$ и количество символов ’a’ и’b’ одинаково$\}$; 5) L*, если L обладает префиксным свойством.
10. Что общего между формальными языками и языками программирования?
11. Какими способами можно задать язык?
12. Почему задание всех цепочек языка перечислением считается способом, не применимым на практике в реальных задачах?
13. Какие вопросы требуется решить при задании языка программирования?
14. Какие способы существуют для определения смысла правильной программы?

1.2 Определение грамматики

1.2.1 Понятие грамматики и формальное определение. Форма Бэкуса-Наура

Грамматика в общем представляет собой описание способа построения цепочек языка. Например, для естественных языков это набор правил построения различных выражений и предложений. Для многих языков (в том числе для языков программирования) возможно использовать формализованное описание – некую математическую систему, описывающую язык. Грамматика задаёт правила порождения цепочек языка, следовательно, является генератором – реализует второй способ задания языка.

Формальное описание грамматики построено на основе системы правил, или продукций. Правило представляет собой упорядоченную пару цепочек символов (α,β), которую обычно записывают как $\alpha \to \beta$ и читают как «α порождает β».

Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальном виде. Поэтому любое описание языка программирования обычно состоит из двух частей: сначала формально излагаются правила построения синтаксических конструкций, затем на естественном языке даётся описание семантических правил. Для компилятора семантические ограничения должны быть представлены в виде алгоритмов проверки правильности программы.

Замечание: Говоря далее о грамматиках языков программирования, будем иметь в виду только правила построения синтаксических конструкций языка.

Формально грамматика G определяется как четвёрка G(VT,VN,P,S), где VT – множество терминальных символов (символы алфавита языка); VN – множество нетерминальных символов (вспомогательные символы, ими могут быть слова, понятия, конструкции языка); причём множества терминальных и нетерминальных символов не пересекаются: $VT \cap VN = \varnothing$; $V = VT \cup VN$ – полный алфавит грамматики G; P – множество правил вида $\alpha \to \beta$, где $\alpha \in V^+$, $\beta \in V^*$; $S \in VN$ – начальный (*целевой*) символ грамматики G, с которого начинается процесс порождения цепочек языка.

Нетерминальные символы участвуют в процессе порождения цепочек языка, но в окончательном виде цепочки языка состоят только из символов терминального алфавита.

В зависимости от вида правил грамматики классифицируются по типам. Эту классификацию рассмотрим далее. Но существуют некоторые общие принципы построения правил любой грамматики:

1. Каждый нетерминальный символ может встречаться в цепочках как левой, так и правой частей правил, но он обязан быть хотя бы один раз в

левой части хотя бы одного правила.

2. В левой части каждого из правил грамматики есть хотя бы один нетерминальный символ.

Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части, например: $\alpha \to \beta_1$, $\alpha \to \beta_2$, …, $\alpha \to \beta_n$. Тогда эти правила записываются в одну строку: $\alpha \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$.

Рассмотренная форма записи правил грамматики *называется формой Бэкуса-Наура*. В ней, как правило, нетерминальные символы заключаются в угловые скобки <>.

Пример: грамматика порождения целых десятичных чисел со знаком.

G({0,1,2,3,4,5,6,7,8,9,–,+}, {<число>,<чс>,<цифра>}, P, <число>).

P: <число> → <чс> | +<чс> | –<чс>
<чс> → <цифра> | <чс><цифра>
<цифра> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

В данной грамматике множество нетерминальных символов содержит 3 элемента (символы <число>,<чс>,<цифра>), множество терминальных символов – 12 элементов (10 цифр и два знака), множество P – 15 правил, записанных в три строки. Начальный символ грамматики – <число>.

В грамматике можно изменить названия нетерминальных символов без какого-то изменения языка, заданного ею. Терминальные символы изменять нельзя! Они соответствуют алфавиту задаваемого языка. При изменении множества терминальных символов изменится и алфавит языка.

Рассмотренную в примере грамматику для целых десятичных чисел со знаком можно переписать в виде:

G′({0,1,2,3,4,5,6,7,8,9,–,+}, {S,T,F}, P, S).

P: S → T | +T | –T
T → F | TF
F → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9.

Формальные грамматики определяют бесконечное множество цепочек языка с помощью конечного набора правил. Это становится возможным благодаря использованию рекурсивных правил, в которых нетерминал выражается сам через себя. Рекурсия может быть непосредственной (явной), когда символ определяется сам через себя в одном правиле; или косвенной (неявной), когда переопределение происходит через цепочку правил.

В грамматиках G и G′ явная рекурсия присутствует во второй части второго правила: <чс> → <чс><цифра>, T → TF.

Чтобы рекурсия не была бесконечной, участвующий в ней нетерминальный символ должен обязательно присутствовать в левой части другого правила, где он определяется, минуя самого себя. В грамматиках G и G′ это достигается в первой части второго правила: <чс> → <цифра>, T → F. Явно или неявно, рекурсия всегда присутствует в грамматиках любых реальных языков программирования. Как правило, в грамматиках реального языка программирования присутствует не одно, а множество правил, построенных с помощью рекурсии.

1.2.2 Другие способы задания грамматик

Кроме формы Бэкуса-Наура, существуют и другие способы записи правил грамматик, а именно запись с использованием метасимволов и графическое представление.

Запись правил грамматик с использованием метасимволов предполагает, что в строке правила могут присутствовать особые символы – т.н. метасимволы, которые трактуются специальным образом. Обычно в этой роли используются () круглые, [] квадратные и {} фигурные скобки, а также «,» запятые и «” ”»кавычки. Они имеют следующий смысл:

- ( ) – в данном месте правила может стоять только одна из всех перечисленных внутри них цепочек;
- [ ] – указанная внутри них цепочка может быть в правиле грамматики один раз или ни одного раза;
- {} – указанная внутри них цепочка может встречаться любое количество раз (в том числе сколь угодно много или ни одного);
- , – разделяет цепочки символов внутри круглых скобок;
- “ ” – используются в том случае, когда какой-то из метасимволов нужно включить в цепочку символов языка.

Правила рассмотренной выше грамматики G порождения целых десятичных чисел со знаком в записи с применением метасимволов будут иметь следующий вид:

<число> → [(+, –)] <цифра>{<цифра>}
<цифра> → (0,1,2,3,4,5,6,7,8,9)

Эти правила означают, что «число есть цепочка символов, которая может начинаться со знака + или – (причём этот знак может и отсутствовать), дальше должна обязательно содержать одну цифру, за которой может следовать (хотя и не обязательно) последовательность цифр любой длины».

Грамматика стала более понятной, кроме того, удалось полностью избежать рекурсии, заменив её символом итерации {}. Эта форма наиболее употребима для одного из типов грамматик, а именно – для регулярных грамматик, которые будут рассмотрены ниже.

При записи правил в графическом виде вся грамматика представляется в виде набора специальным образом построенных диаграмм. Эта форма доступна только для грамматик контекстно-свободных и регулярных типов, но этого достаточно, чтобы её можно было использовать для задания известных языков программирования.

В такой форме записи каждому нетерминальному символу соответствует диаграмма, построенная в виде направленного графа. Граф имеет следующие типы вершин:

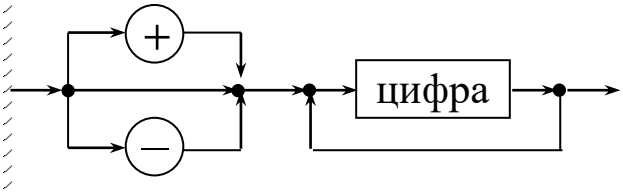
- точка входа (не обозначена – из неё начинается входная дуга графа);
- нетерминальный символ – на диаграмме обозначен прямоугольником, в который вписано обозначение символа;
- цепочка терминальных символов – обозначается овалом, кругом или скругленным прямоугольником;
- узловая точка – обозначена закрашенным кружком;
- точка выхода – в нее входит выходная дуга графа.

Каждая диаграмма имеет только одну точку входа и одну точку выхода, но сколько угодно вершин других типов. Вершины соединяются направленными дугами, из входной точки дуги могут только выходить, в выходную точку – только входить. Все остальные вершины должны иметь как минимум по одной выходной и одной входной дуге.

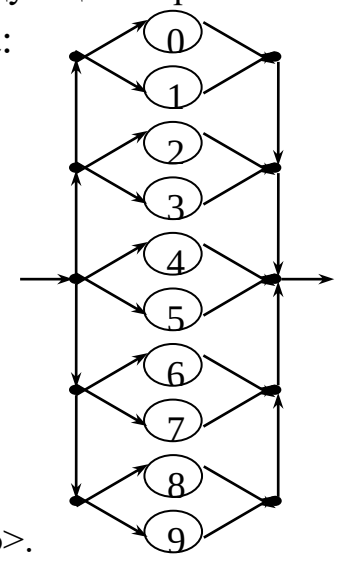
Чтобы построить цепочку символов, соответствующую нетерминальному символу грамматики, нужно рассмотреть диаграмму для этого символа. Начав движение от точки входа, нужно двигаться по дугам до точки выхода, помещая при этом все встречающиеся по пути нетерминальные символы или терминальные цепочки в результирующую цепочку. Через любую вершину графа можно пройти один раз, ни разу или сколь угодно много раз, в зависимости от пути движения. Если результирующая цепочка содержит нетерминальные символы, нужно в свою очередь рассмотреть соответствующие им диаграммы, до тех пор, пока не будет получена цепочка, состоящая полностью из терминальных символов. Для получения цепочки заданного языка процесс порождения необходимо начинать с диаграммы начального символа грамматики.

Описанная ранее грамматика G порождения целых десятичных чисел со знаком в графическом виде будет выглядеть следующим образом:

Число:



Цифра:



При этом начальным символом является <число>.

Данный способ в основном применяется в литературе при изложении грамматик языков программирования. Он удобен для пользователей – разработчиков, но практического применения в компиляторах не имеет.

1.3 Классификация языков и грамматик

От того, к какому типу относится тот или иной язык программирования, зависит сложность распознавателя для этого языка. Для некоторых типов языков в принципе невозможно построить компилятор, который анализировал бы исходные тексты на этих языках за приемлемое время на основе ограниченных вычислительных ресурсов.

1.3.1 Классификация грамматик по Хомскому

Формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют определённой структуре, то её относят к заданному типу. Если хотя бы одно правило грамматики не удовлетворяет требованиям структуры, то она не попадает в заданный тип. Если правила грамматики соответствуют структуре нескольких типов, то она будет отнесена по классификации к самому простому из них.

Пусть грамматика обозначена как **G(VT,VN,P,S)**, **V=VT∪VN.** В соответствии с иерархией Хомского выделяют 4 типа грамматик.

1. **Тип 0 – грамматики *с фразовой структурой*, или *без ограничений***

На структуру их правил не накладывается никаких ограничений, т.е. правила имеют вид: $\alpha \to \beta$, где $\alpha \in V^+$, $\beta \in V^*$. Это самый общий тип грамматик. Грамматики, которые относятся только к этому и не могут быть отнесены ни к какому другому типу, являются самыми сложными.

2. **Тип 1 – *Контекстно-зависимые* (КЗ) и *неукорачивающие* грамматики**

К этому типу относятся два основных класса грамматик.

Контекстно-зависимые грамматики имеют правила вида $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, где $\alpha_1, \alpha_2 \in V^*$, $A \in VN$, $\beta \in V^+$.

Неукорачивающие грамматики имеют правила вида $\alpha \to \beta$, где $\alpha, \beta \in V^+$, $|\beta| \geq |\alpha|$.

В КЗ-грамматиках при построении предложений заданного языка один и тот же нетерминальный символ может быть заменен различными терминальными цепочками в зависимости от контекста, в котором он встречается. Цепочки α_1 и α_2 в правилах обозначают контекст: α_1 – левый контекст, α_2 – правый контекст. В общем случае они могут быть пустыми.

В неукорачивающих грамматиках при построении предложений языка цепочка символов заменяется на цепочку не меньшей длины.

Эти два класса грамматик эквивалентны.

При построении компиляторов такие грамматики не применяются, поскольку языки программирования имеют более простую структуру и могут быть построены с помощью грамматик других типов.

3. Тип 2 – ***Контекстно-свободные*** (КС) грамматики

Контекстно-свободные (КС) грамматики имеют правила вида $A \to \beta$, где $A \in VN$, $\beta \in V^+$. В правой части у них стоит всегда хотя бы один символ. Их отличие от предыдущих типов состоит в том, что левая часть правил должна состоять ровно из одного нетерминального символа. Такие грамматики еще называют неукорачивающими контекстно-свободными (НКС) грамматиками.

Существует почти эквивалентный им класс укорачивающих контекстно-свободных (УКС) грамматик, отличие которого в том, что он допускает пустую цепочку, т.е. правила имеют вид $A \to \beta$, где $A \in VN$, $\beta \in V^*$. В дальнейшем, если возможность наличия в языке пустой цепочки не имеет принципиального значения, будем говорить просто о КС-грамматиках. КС-грамматики широко используются при описании синтаксических конструкций языков программирования.

4. Тип 3 – ***Регулярные*** грамматики

В правой части правил грамматик этого типа может присутствовать не более одного нетерминального символа, причём он должен быть расположен во всех правилах одной грамматики с одной и той же стороны от цепочки терминалов, а требования к левой части правил совпадают с предыдущим типом. К этому типу относятся два эквивалентных класса грамматик: леволинейные и праволинейные (их название определяется местоположением нетерминального символа в правой части правил относительно терминальной цепочки). Для любой праволинейной грамматики можно построить эквивалентную ей леволинейную, задающую тот же язык, и наоборот.

Леволинейные грамматики могут иметь правила двух видов: $A \to B\gamma$, или $A \to \gamma$, где $A,B \in VN$, $\gamma \in VT^*$. Нетерминальный символ в правой части правил размещается слева от цепочки терминалов.

Праволинейные грамматики имеют правила тоже двух видов: $A \to \gamma B$, или $A \to \gamma$, где $A,B \in VN$, $\gamma \in VT^*$. Соответственно нетерминальный символ находится справа от терминальных.

Регулярные грамматики используются при описании простейших конструкций языков программирования: идентификаторов, констант, строк, комментариев и т.д. Они очень просты и удобны в использовании, поэтому в компиляторах на их основе строятся функции лексического анализа входного языка.

Из определения типов видно, что любая регулярная грамматика является также КС-грамматикой, или любая грамматика может быть отнесена к типу 0. В то же время существуют УКС-грамматики, которые не относятся к типу 1, поскольку могут содержать правила вида $A \to \lambda$, недопустимые в этом типе. В общем, сложность грамматики обратно пропорциональна тому максимально возможному номеру типа, к которому может быть отнесена эта грамматика. Самыми простыми являются грамматики типа 3, самыми сложными – типа 0.

1.3.2 Классификация языков

Языки классифицируются согласно иерархии Хомского в соответствии с типами грамматик, с помощью которых они заданы, причем из всех эквивалентных грамматик, задающих один и тот же язык, выбирается грамматика с максимально возможным номером, т.е. самая простая. Сложность языков соответствует сложности грамматик. От классификационного типа языка зависит сложность его распознавателя.

1. **Тип 0 – языки *с фразовой структурой***

Это самые сложные языки, для распознавания которых требуются вычислители, равномощные машине Тьюринга. Для такого языка невозможно построить компилятор, который выполнил бы разбор за ограниченное время на основе ограниченных вычислительных ресурсов.

Практически все естественные языки относятся к этому типу. Одно и то же слово в естественном языке может иметь различный смысл в зависимости от контекста и играть различную роль в предложении. Такие языки далее рассматриваться не будут.

2. **Тип 1 – *контекстно-зависимые* (КЗ) языки**

В общем случае время на распознавание языка типа 1 экспоненциально зависит от длины исходной цепочки символов.

Языки и грамматики этого типа используются в переводе текстов на естественных языках. Распознаватели, построенные на их основе, позволяют анализировать тексты с учётом контекстной зависимости в предложениях входного языка, хотя в общем случае для точного перевода всё же требуется вмешательство человека. Такие грамматики могут использоваться в сервисных функциях проверки орфографии в языковых процессорах.

Однако языки программирования имеют более простую структуру, поэтому в компиляторах КЗ-языки не применяются.

3. **Тип 2 – *контекстно-свободные* (КС) языки**

КС-языки лежат в основе большинства современных языков программирования, на их основе работают некоторые командные процессоры, допускающие управляющие команды цикла и условия.

В общем случае время на распознавание предложений языка этого типа полиномиально зависит от длины цепочки символов (это кубическая или квадратичная зависимость в зависимости от класса языка). Но среди КС-языков существует много классов, для которых эта зависимость линейна, и многие языки программирования можно отнести к одному из таких классов.

4. **Тип 3 – *регулярные* языки**

Это самый простой тип языков, и они являются наиболее широко распространенным типом, используемым в вычислительных системах. Время на распознавание цепочек языка линейно зависит от их длины. Поэтому иногда эти языки ещё называют *линейными*. Для работы с такими языками используются регулярные множества и выражения, конечные автоматы.

КС-языки и регулярные языки будут рассматриваться подробно.

1.3.3 Контрольные вопросы

1. К какому типу будет относиться язык, если его можно задать как регулярной, так и контекстно-свободной грамматикой?
2. Что общего у грамматик регулярного и КС типов? Каковы их различия?
3. В чём отличие КС-грамматики от грамматики типа 1?
4. Какие типы языков используются в теории языков программирования?
5. Построить регулярную грамматику: 1) в форме Бэкуса-Наура; 2) с использованием метасимволов; 3) в графическом виде – для генерации следующих языков:
   а) Множество всех цепочек из {0,1,a,b,c}*, начинающихся с 0 или 1.
   б) Множество всех цепочек из {0,1}*, длины которых делятся на три.
   в) Множество всех цепочек из {0,1}*, содержащих подцепочку ’101’.
   г) Множество цепочек из {0,1,a,b}*, заканчивающихся цепочкой ’01a’.
   д) Множество всех цепочек из {0,1}*, содержащих четное число нулей и четное число единиц.
6. Построить грамматику, порождающую множество строк из нулей и единиц, т.е. язык {0,1}*: 1) праволинейную; 2) контекстно-свободную.
7. Построить КС-грамматики, порождающие следующие языки: 1) $\{0^n \mid n \geq 0\}$; 2) $\{0^n \mid n \geq 1\}$; 3) $\{0^{3n} \mid n \geq 1\}$; 4) $\{a^n b^n \mid n \geq 1\}$; 5) {w | w ∈ {0,1}* и количество нулей и единиц одинаково}; 6) {w | w ∈ {0,1}* и количество как нулей, так и единиц четно}; 7) {w | w ∈ {0,1,a,b}* и начинающиеся с ’a’ или ’b’}; 8) всевозможные последовательности правильно расставленных скобок.
8. Определить тип Хомского грамматики G = (Σ, N, P, S), где Σ = {x, y, z}, N = {S, A, B, T}, а правила вывода имеют вид: P: (1) S → xB | xTB; (2) T → xA | xTA; (3) B → yz; (4) Ay → yA; (5) Az → yzz. Принадлежат ли L(G) цепочки x^2yxz, $x^2y^2z^2$, xyxz ?
9. Дана грамматика G = (Σ, N, P, S), где Σ = {a, b, c}, N = {S, B, C}, P: (1) S → aSBC | abC; (2) CB → BC; (3) bB → bb; (4) bC → bc; (5) cC → cc. Определить, какого вида цепочки порождаются данной грамматикой, записать полученный язык. Какого типа эта грамматика?

1.4 Вывод и выводимость

1.4.1 Цепочки вывода. Сентенциальная форма

Выводом называется процесс порождения цепочек языка на основе правил определяющей язык грамматики.

Цепочка $\beta=\delta_1\gamma\delta_2$ называется *непосредственно выводимой* из цепочки $\alpha=\delta_1\omega\delta_2$ в грамматике **G(VT,VN,P,S)**, **V=VT∪VN,** $\delta_1,\gamma,\delta_2 \in \mathbf{V}^*$, $\omega \in \mathbf{V}^+$, если в

грамматике G существует правило $\omega \to \gamma \in \mathbf{P}$. Непосредственная выводимость обозначается $\alpha \Rightarrow \beta$. Т.е. цепочка β *непосредственно выводима* из цепочки α в том случае, если можно взять несколько символов в цепочке α, заменить их на другие символы согласно правилу грамматики и получить при этом цепочку β. В формальном определении любая из цепочек δ_1, δ_2 (или обе) может быть пустой. В пределе вся цепочка α может быть заменена на цепочку β, тогда в грамматике должно существовать правило $\alpha \to \beta \in \mathbf{P}$.

Цепочка β называется *выводимой из цепочки* α – обозначается $\alpha \Rightarrow^* \beta$, если выполняется одно из двух следующих условий:

- β непосредственно выводима из α: $\alpha \Rightarrow \beta$;
- $\exists\gamma$, такая, что: γ выводима из α и β непосредственно выводима из γ ($\alpha \Rightarrow^* \gamma$ и $\gamma \Rightarrow \beta$).

Это рекурсивное определение выводимости цепочки. Т.е. β выводима из цепочки α, если она либо непосредственно выводима, либо можно построить последовательность непосредственно выводимых цепочек от α к β: $\alpha \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots \Rightarrow \gamma_i \Rightarrow \gamma_{i+1} \ldots \Rightarrow \gamma_n \Rightarrow \beta$. Такая последовательность непосредственно выводимых цепочек называется *выводом* или *цепочкой вывода*, а каждый переход – *шагом вывода*. Если β непосредственно выводима из α ($\alpha \Rightarrow \beta$), то имеется всего один шаг вывода.

Если вывод содержит два или более шагов, то говорят, что β *нетривиально выводима* из α: $\alpha \Rightarrow^+ \beta$. Если количество шагов вывода известно, его можно указать у знака выводимости цепочек: $\alpha \Rightarrow^4 \beta$ означает, что β выводима из α за 4 шага. Запись $\alpha \Rightarrow^0 \beta$ означает, что цепочки равны. Целесообразно при записи цепочки вывода на каждом шаге над знаком выводимости указывать номер правила, согласно которому выполнен этот шаг.

Грамматика, в которой для любой цепочки порождаемого языка существует единственная цепочка вывода, называется *однозначной*.

Вывод называется *законченным*, если из полученной цепочки нельзя сделать более ни одного шага, т.е. если полученная цепочка пустая или содержит только терминальные символы грамматики: $\beta \in \mathbf{VT}^*$. Цепочка, полученная в результате законченного вывода, называется *конечной цепочкой вывода*.

Цепочка символов $\alpha \in \mathbf{V}^*$ называется *сентенциальной формой грамматики* G**(VT,VN,P,S)**, если она выводима из целевого символа грамматики S: $S \Rightarrow^* \alpha$. Если цепочка получена в результате законченного вывода, она называется *конечной сентенциальной формой*.

Вывод называется *левосторонним*, если в нём на каждом шаге вывода правило грамматики применяется к самому левому нетерминальному символу в цепочке. Аналогично определяется *правосторонний* вывод.

Если символы заменяются в произвольном порядке, вывод нельзя отнести ни к какому из типов. Для КС - грамматик для любой сентенциальной формы всегда можно построить левосторонний или правосторонний вывод. Для грамматик более сложных типов это не всегда возможно (структура правил не всегда позволяет заменять крайний левый или крайний правый нетерминальные символы в цепочке).

Рассмотрим снова пример грамматики целых десятичных чисел со знаком.

G ({0,1,2,3,4,5,6,7,8,9,–,+}, {S,T,F}, P, S).

P: $S \to T \mid +T \mid -T$

$T \to F \mid TF$

$F \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9.$

Построим в ней несколько цепочек вывода, причём в качестве начала будем брать не только целевой символ, но и другие символы или их сочетания.

(1) $S \Rightarrow -\underline{T} \Rightarrow -\underline{T}F \Rightarrow -\underline{F}F \Rightarrow -1\underline{F} \Rightarrow -19$ (левосторонний);

(2) $T \Rightarrow T\underline{F} \Rightarrow \underline{T}0 \Rightarrow T\underline{F}0 \Rightarrow \underline{T}30 \Rightarrow \underline{F}30 \Rightarrow 730$; (правосторонний)

(3) $S \Rightarrow T \Rightarrow \underline{T}F \Rightarrow T\underline{F}F \Rightarrow \underline{T}6F \Rightarrow F6\underline{F} \Rightarrow \underline{F}63 \Rightarrow 263$; (–)

(4) $S \Rightarrow T \Rightarrow T\underline{F} \Rightarrow \underline{T}3 \Rightarrow T\underline{F}3 \Rightarrow \underline{T}63 \Rightarrow \underline{F}63 \Rightarrow 263$; (правосторонний);

(5) $S \Rightarrow T \Rightarrow \underline{T}F \Rightarrow T\underline{F}F \Rightarrow \underline{T}8F \Rightarrow F8F$ (не законченный);

(6) $TF\underline{T} \Rightarrow TFT\underline{F} \Rightarrow TFF\underline{F} \Rightarrow TF\underline{F}0 \Rightarrow T\underline{F}10 \Rightarrow \underline{T}210 \Rightarrow \underline{F}210 \Rightarrow 1210$ (правост.)

Здесь (1) – (4) – конечные сентенциальные формы, поскольку цепочка в (2) *<u>может быть</u>* получена из целевого символа грамматики (хотя в этом примере и получена из нетерминала T); (5) – просто сентенциальная форма (не конечная). В выводе (6) в явном виде не присутствует сентенциальная форма, хотя цепочка 1210 и является конечной сентенциальной формой. Для того, чтобы это подтвердить, достаточно построить другой вывод этой цепочки из целевого символа. А цепочка TFT не является сентенциальной формой, т.к. её невозможно получить из целевого символа. Все выводы, за исключением (5), являются законченными. На примере (3), (4) видно, что одна и та же цепочка может быть получена посредством разных выводов. Выводы (2), (4), (6) – правосторонние, (1) – левосторонний, (3), (5) – ни то, ни другое.

1.4.2 Дерево вывода

Деревом вывода грамматики G называется ориентированное дерево (граф), которое соответствует некоторой цепочке вывода и удовлетворяет следующим условиям:

- каждая вершина дерева обозначается символом грамматики $A \in V$;
- корнем дерева является вершина, обозначенная целевым символом грамматики S;
- листьями дерева являются вершины, обозначенные терминальными

символами грамматики или символом λ;

- если некоторый узел обозначен символом $A \in VN$, а связанные с ним узлы – символами $b_1, b_2, \ldots, b_n \mid n > 0,\ 0 < i \le n,\ b_i \in V \cup \{\lambda\}$, то в грамматике существует правило $A \to b_1 b_2 \ldots b_n \in \mathbf{P}$.

По структуре правил дерево вывода всегда можно построить для грамматик типа 2 и 3. Для строго формализованного построения дерева удобнее пользоваться левосторонним либо правосторонним выводом. Дерево вывода можно построить двумя способами: сверху вниз (обычно левосторонний вывод) и снизу вверх (обычно правосторонний).

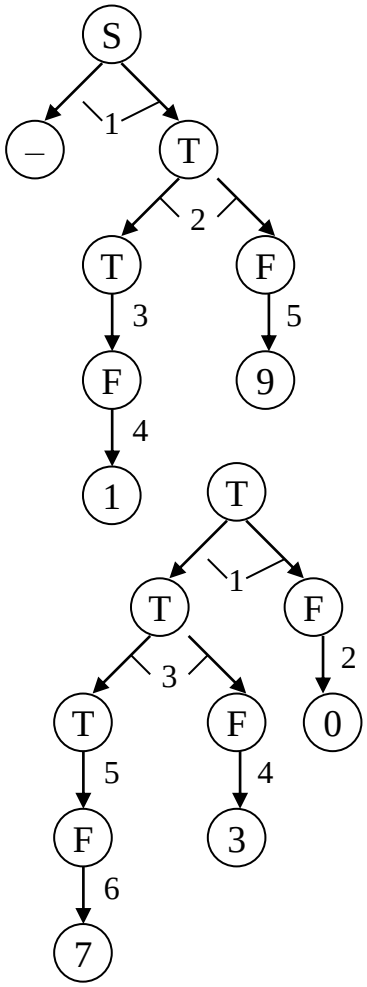
Алгоритм построения дерева сверху вниз:

1. целевой символ грамматики помещается в вершину дерева (корень);
2. в грамматике выбирается необходимое правило и целевой символ раскрывается на несколько символов первого уровня;
3. среди всех концевых вершин дерева выбирается крайняя (левая для левостороннего вывода, правая – для правостороннего вывода) вершина, обозначенная нетерминальным символом;
4. для неё выбирается нужное правило и она снова раскрывается на несколько вершин следующего уровня;
5. если все концевые вершины обозначены терминальными символами, процесс закончен. Иначе – возврат на шаг 3.

При построении дерева снизу вверх процесс построения начинается с листьев, в качестве которых выбираются терминальные символы цепочки языка. Они образуют последний уровень дерева; далее в грамматике выбирается соответствующее правило, согласно которому один или несколько символов цепочки могут быть «свёрнуты» в нетерминальный символ – (при левостороннем или правостороннем выводе это крайние символы в слое дерева) они соединяются с новой вершиной; и так до тех пор, пока все вершины не окажутся соединены в корневой вершине.

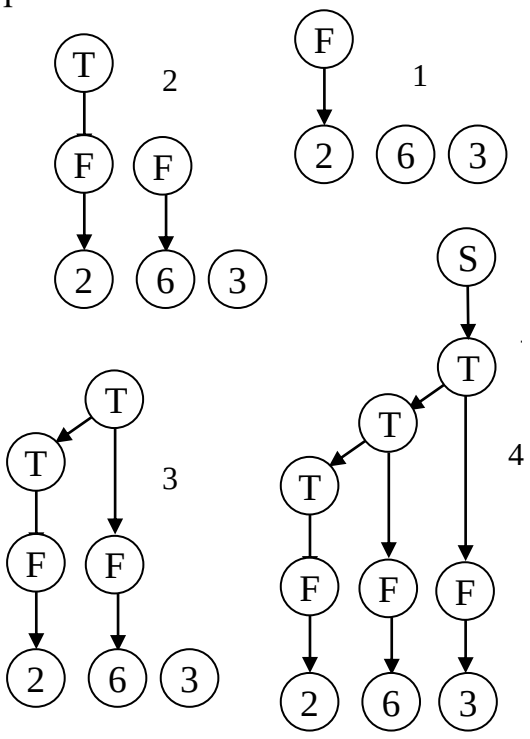
Поскольку компилятор читает программу сверху вниз и слева направо, для построения дерева сверху вниз обычно используется левосторонний вывод.

Пример: Рассмотрим деревья вывода для сентенциальных форм (1) и (2). Подробно рассмотрим построение для формы (1).



Дерево строится сверху вниз. Номера шагов обозначены цифрами. Процесс начинается с корня (он помечен целевым символом), далее при выводе применялось правило $S \to -T$ (первый шаг), следовательно, в следующем уровне будут две вершины («–» и «T»). Вершина «–» помечена терминальным символом, значит, это лист. Вершина «T» представляет собой нетерминал и вновь раскрывается по правилу $T \to TF$ согласно ранее построенному выводу (второй шаг). Поскольку вывод левосторонний, каждый раз в цепочке вывода очередное правило применяется к крайнему левому нетерминалу. Поэтому следующим заменяется нетерминал T, как самый левый в цепочке вывода (она в настоящий момент имеет вид –TF), и это третий шаг… Процесс продолжается до тех пор, пока все вершины не получат в качестве меток терминальные символы.

Для формы (2) построение отличается тем, что в качестве начального символа вместо целевого S взят нетерминал T, а вывод использован правосторонний.



Построение дерева снизу вверх является более неоднозначным процессом. Рассмотрим его для вывода формы (4). При этом будем двигаться по цепочке вывода от её конца к началу. Сначала строим листья «2», «6» и «3». Последним шагом было правило $F \to 2$. Оно позволяет заменить терминальный символ ‘2’ на ‘F’ (схема 1). Следующие правила $T \to F$ и $F \to 6$ (схема 2). Далее по правилу $T \to TF$ нетерминалы T и F сворачиваются в T (схема 3). И наконец после применения правил $F \to 3$, $T \to TF$ и $S \to T$ получается итоговое дерево (схема 4).

1.4.3 Контрольные вопросы

1. За сколько шагов выводима цепочка β из цепочки α, если в правилах грамматики есть правило $\alpha \to \beta$?
2. В чём заключается различие понятий выводимости и непосредственной выводимости?
3. Какая грамматика является однозначной?
4. Какой вывод называется законченным?
5. Что такое сентенциальная форма грамматики?
6. В чём состоит различие левостороннего и правостороннего выводов для регулярной грамматики?
7. Чем отличается дерево вывода от самого вывода?
8. Каким символом помечают корень дерева вывода и какими –листья?

1.5 Распознаватели. Задача разбора

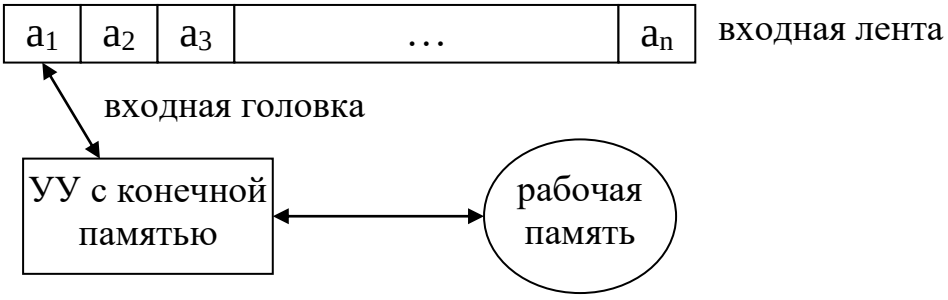
1.5.1 Общая схема распознавателя

В числе прочих задач компилятор должен определить принадлежность некоторого текста к конкретному языку. В отношении исходной программы компилятор выступает в роли распознавателя, а человек, создавший программу – в роли генератора цепочек этого языка.

Распознаватель – это специальный алгоритм, позволяющий для некоторой цепочки символов определить, принадлежит ли она заданному языку. Это один из способов задания языка.

Распознаватель входит в состав компилятора и является частью программного обеспечения компьютера.

Условная схема распознавателя имеет следующий вид:



Основные компоненты распознавателя:

- *входная лента* – линейная последовательность клеток, или ячеек, каждая из которых содержит ровно один символ входного алфавита;
- *входная* (считывающая) *головка* обозревает одну входную ячейку; на каждом шаге работы может сдвигаться на одну ячейку вправо, влево или оставаться на месте;
- *устройство управления* (УУ), которое координирует работу распознавателя, имеет некоторое множество состояний и конечную

память;

- *внешняя* (рабочая) *память* может хранить некоторую информацию в процессе работы распознавателя и может иметь неограниченный объем.

Алфавит распознавателя конечен; он включает в себя все допустимые символы входных цепочек, а также некоторый дополнительный алфавит символов, которые могут обрабатываться УУ и храниться в рабочей памяти распознавателя.

В процессе своей работы распознаватель может выполнять некоторые элементарные операции, такие как чтение входного символа, сдвиг головки, доступ к рабочей памяти для чтения или записи информации, изменение состояния УУ.

Работа распознавателя состоит из последовательности шагов, или тактов. То, каким должен быть этот такт, определяется текущим входным символом, состоянием УУ и символом, извлеченным из памяти. Итак,

Такт состоит из следующих моментов:

- входная головка распознавателя сдвигается на одну ячейку вправо, влево или остается на месте;
- в память помещается некоторая информация;
- изменяется состояние УУ.

В процессе работы распознавателя происходит смена конфигураций. *Конфигурация* распознавателя (*мгновенное описание*) определяется следующими параметрами:

- состояние УУ;
- содержимое входной ленты и положение считывающей головки в ней;
- содержимое внешней памяти.

Конфигурация называется *начальной*, если УУ находится в начальном состоянии, входная головка обозревает самый левый символ на входной ленте, а память имеет заранее установленное начальное содержимое.

Конфигурация называется *заключительной*, если УУ находится в одном из множества заключительных состояний, а входная головка обозревает правый концевой маркер или сошла с ленты.

Распознаватель *допускает входную цепочку символов*, если, находясь в начальной конфигурации, в которой данная цепочка записана на входной ленте, он может проделать конечную последовательность шагов, заканчивающуюся одной из его заключительных конфигураций.

Некоторые виды распознавателей могут из начальной конфигурации проделать различные последовательности шагов, из которых, может быть, лишь некоторые (или даже одна) приведут к заключительной конфигурации. В таком случае входная цепочка является допущенной.

Язык, определяемый распознавателем – это множество всех цепочек, которые допускает этот распознаватель.

1.5.2 Виды распознавателей и их классификация

Распознаватели классифицируют в зависимости от вида составляющих их компонентов: считывающего устройства, устройства управления и внешней памяти.

По видам *считывающего устройства* распознаватели могут быть двусторонними и односторонними. *Односторонние* распознаватели допускают перемещение считывающей головки по ленте только в одном направлении, обычно слева направо (такой распознаватель называется левосторонним). Такие распознаватели не возвращаются назад к уже прочитанной части цепочки. *Двусторонние* распознаватели допускают перемещение считывающей головки в любом направлении.

По видам *устройства управления* распознаватели бывают детерминированные и недетерминированные. Распознаватель является *детерминированным*, если для любой допустимой конфигурации существует не более одной следующей конфигурации. Если для любой допустимой конфигурации единственно возможна ровно одна следующая конфигурация, то такой распознаватель является детерминированным полностью определённым, иначе он неполностью определённый. Для *недетерминированного* распознавателя на некотором шаге возможно совершить несколько альтернативных переходов в различные конфигурации. При этом может оказаться, что только одна из возможных последовательностей шагов приводит в заключительную конфигурацию.

По видам *внешней памяти* распознаватели бывают следующих типов:

- распознаватели *без внешней памяти*;
- распознаватели *с ограниченной внешней памятью*;
- *с неограниченной* внешней памятью.

Распознаватели *без внешней памяти* моделируются конечными автоматами и используют в процессе работы только конечную память УУ.

Размер внешней памяти распознавателей второго типа ограничен. Эти ограничения могут носить характер некоторой функциональной зависимости от длины исходной цепочки символов – линейной, полиномиальной, экспоненциальной и т.п. Кроме того, может быть указан способ организации внешней памяти – стек, очередь, список и т.п. Например, широко распространены распознаватели с памятью магазинного типа, которая организована по стековому принципу.

В распознавателях последнего типа предполагается, что для их работы может потребоваться внешняя память неограниченного объема, вне зависимости от длины входной цепочки. У таких распознавателей используется память с произвольным методом доступа.

Все три рассмотренных составляющих организуют общую классификацию распознавателей. Например, в ней возможен такой тип: «односторонний детерминированный полностью определённый

распознаватель с линейно ограниченной стековой памятью». Тип распознавателя в классификации определяет его сложность в целом. Сложность распознавателя напрямую связана с типом языка, цепочки которого он должен определять.

Например, для языков типа 1 (КЗ) распознавателями являются двусторонние недетерминированные автоматы с линейно ограниченной внешней памятью. Такой алгоритм уже может быть реализован в ПО компьютера. Но экспоненциальная зависимость времени разбора от длины входной цепочки существенно ограничивает применение распознавателей для КЗ-языков. Такие распознаватели, как правило, применяются для автоматизированного перевода текстов на естественных языках, когда временные ограничения на разбор текста несущественны.

Для КС-языков распознавателями являются односторонние недетерминированные автоматы с магазинной (стековой) внешней памятью. Кроме того, среди всего множества КС-языков можно выделить подкласс детерминированных языков, для которых распознавателями являются детерминированные автоматы с магазинной памятью – ДМПА. Этот класс языков наиболее интересен для построения компиляторов.

Для языков программирования более целесообразно строить компиляторы на основе КС-языков, дополняя их семантическим анализатором, чем использовать в качестве основы КЗ-языки – такая комбинация имеет более высокую скорость работы.

Для регулярных языков распознавателями являются односторонние недетерминированные распознаватели без внешней памяти – конечные автоматы (КА). Кроме того, любой недетерминированный КА всегда может быть преобразован в детерминированный (ДКА). В компиляторах распознаватели на основе регулярных языков используются для лексического анализа текста исходной программы. Регулярные языки находят применение также во многих областях, связанных с разработкой ПО вычислительных систем.

1.5.3 Задача разбора

Грамматики и распознаватели – два независимых метода, которые реально могут быть использованы для определения языка. Однако при разработке компилятора для некоторого языка программирования возникает задача, которая требует связать между собой эти два метода.

Разработчики компилятора имеют дело с конкретным языком программирования, т.е. грамматика языка уже известна. Задача разработчиков состоит в построении распознавателя данного языка, который явится основой синтаксического анализатора в компиляторе.

Итак, *задача разбора* формулируется следующим образом: на основе имеющейся грамматики некоторого языка необходимо построить

распознаватель для этого языка. Заданная грамматика и распознаватель должны быть эквивалентны, т.е. определять один и тот же язык.

В общем виде эта задача может быть решена не для всех типов языков, хотя для КС и регулярных языков задача разбора разрешима и существуют некоторые формальные методы её решения.

Компилятор должен не просто дать ответ, принадлежит ли входная цепочка данному языку, но и определить её смысл. Фактически работа распознавателя в составе компилятора заключается в построении дерева разбора, которое затем используется для генерации кода. Кроме того, если исходная цепочка не принадлежит к заданному языку, компилятор должен не просто установить факт ошибки, но и по возможности определить её тип и местоположение.

1.5.4 Контрольные вопросы

1. Что такое распознаватель? Какова его роль при задании языка?
2. Какие операции может выполнять распознаватель в процессе работы?
3. Чем определяется каждый такт распознавателя? Какие действия может совершать считывающая головка во время одного такта?
4. Какими параметрами определяется конфигурация распознавателя?
5. Какая конфигурация является начальной?
6. Что обозревает входная головка, когда распознаватель находится в заключительной конфигурации?
7. Может ли существовать несколько заключительных конфигураций, или же такая конфигурация единственна?
8. В каком случае считается, что распознаватель допускает цепочку символов?
9. В чём состоит различие детерминированного и недетерминированного распознавателей?
10. Может ли распознаватель являться полностью определённым и недетерминированным одновременно?
11. Какую внешнюю память имеет распознаватель, моделью которого является конечный автомат?
12. Какую память использует конечный автомат?
13. Какие автоматы являются распознавателями КС-языков?
14. Какова взаимосвязь грамматик и распознавателей при разработке компилятора?
15. В чём состоит задача разбора?
16. Пусть недетерминированный распознаватель при разборе некоторой цепочки может проделать 5 различных последовательностей шагов, только одна из которых приводит к заключительной конфигурации. Будет ли эта цепочка допущена?