МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«СЕВЕРО-КАВКАЗСКАЯ ГОСУДАРСТВЕННАЯ АКАДЕМИЯ»

ИНСТИТУТ ЦИФРОВЫХ ТЕХНОЛОГИЙ

КАФЕДРА ПРИКЛАДНОЙ ИНФОРМАТИКИ

П. А. Кочкарова
М.У. Эркенова

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебно-методическое пособие для обучающихся 4 курсов
по направлению подготовки 09.03.04 Программная инженерия

Черкесск, 2024

УДК 004.43
ББК 32.973-012.1
К 75

Рассмотрено на заседании кафедры «Прикладная информатика»
Протокол № 1 от 31 августа 2023 г.
Рекомендовано к изданию редакционно-издательским советом СКГА.
Протокол № 26 от «29» сентября 2023 г.

Рецензенты: – Бостанова Л.К. – к.п.н., доцент кафедры информатики и информационных технологий

К 75 **Кочкарова, П. А.** Теория языков программирования и методы трансляции: учебно-методическое пособие для обучающихся 4 курсов по направлению подготовки 09.03.04 Программная инженерия / П. А. Кочкарова, М.У. Эркенова.– Черкесск: БИЦ СКГА, 2024. – 36 с.

УДК 004.43
ББК 32.973-012.1

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Предлагаемое учебно-методическое пособие предназначено для обучающихся по направлению подготовки 09.03.04 Программная инженерия.

В нём содержится материал, предназначенный для проведения лабораторных занятий по указанному учебному курсу с целью изучения основных принципов теории формальных языков и грамматик, а также методов лексического и синтаксического анализа современных языков программирования. В пособии по каждой теме имеется теоретический материал, контрольные примеры и задания для самостоятельного выполнения.

Тема 1.Формальные языки и грамматики
Краткие теоретические сведения

Тексты на любом языке (естественном или формальном) представляют собой цепочки символов некоторого алфавита. Цепочкой символов называют произвольную упорядоченную конечную последовательность символов, записанных один за другим.

Понятие символа является базовым в теории формальных языков. Для цепочки символов имеют значение три фактора: состав входящих в цепочку символов, их количество и порядок символов в цепочке. Далее цепочки символов будем обозначать греческими буквами: α, β, γ и др. Цепочки символов α и β равны (α = β), если они имеют один и тот же состав символов, одно и то же их количество и одинаковый порядок следования символов в цепочке. Количество символов в цепочке определяет её длину. Длина цепочки α обозначается как | α |.

Можно выделить следующие операции над цепочками символов: ⎫ конкатенация (объединение, сложение двух цепочек) – это дописывание второй цепочки в конец первой. Конкатенация цепочек α и β обозначается как αβ.

Например: α = аб, β = вг, тогда αβ = абвг. При этом αβ ≠ βα, так как в цепочке важен порядок символов. Но конкатенация обладает свойством ассоциативности: (αβ)γ = α(βγ); ⎫ замена (подстановка) – замена подцепочки символов на любую произвольную цепочку символов. В результате получается новая цепочка символов.

Например: γ = абвг, разобьём эту цепочку символов на подцепочки: α = а, ω = б, β = вг и выполним подстановку цепочки υ = аба вместо подцепочки ω. Получим новую цепочку γ' = ааабавг. Таким образом, 2 подстановка выполняется путём разбиения исходной цепочки на подцепочки и конкатенации; ⎫ обращение – запись символов цепочки в обратном порядке.

Эта операция обозначается как αR. Если α = абвг, то αR = гвба. Для операции обращения справедливо следующее: (αβ) R = βRαR; ⎫ итерация – повторение цепочки n раз, где n > 0 – это конкатенация цепочки с собой n раз, обозначается как αn. Если n = 0, то результатом итерации будет пустая цепочка символов.

Пустая цепочка символов – это цепочка, не содержащая ни одного символа. Будем обозначать такую цепочку как ε. Для пустой цепочки справедливы следующие равенства: |ε| = 0 εα = αε = α ε R = ε ε n = ε, n ≥ 0 α0 = ε Основой любого языка является алфавит, определяющий набор допустимых символов языка.

Алфавит – это счётное множество допустимых символов языка. Будем обозначать это множество как V. Цепочка символов α является цепочкой над алфавитом V: α(V), если в неё входят только символы, принадлежащие множеству символов V. Для любого алфавита V пустая цепочка может как являться, так и не являться цепочкой над алфавитом. Если V – некоторый

алфавит, то: ⎱ V+ – множество всех цепочек над алфавитом V без пустой цепочки; ⎱ V* – множество всех цепочек над алфавитом V, включая пустую цепочку.

Языком L над алфавитом V (L (V)) называется некоторое счётное подмножество цепочек конечной длины из множества всех цепочек над алфавитом V. Множество цепочек языка не обязано быть конечным; хотя каждая цепочка символов, входящая в язык, обязана иметь конечную длину, эта длина может быть сколь угодно большой и формально ничем не ограничена.

Цепочку символов, принадлежащую заданному языку, часто называют предложением языка, а множество цепочек символов некоторого языка L (V) – множеством предложений этого языка. Кроме алфавита язык предусматривает также правила построения допустимых цепочек, поскольку обычно далеко не все цепочки над заданным алфавитом принадлежат языку.

Символы могут объединяться в слова или лексемы – элементарные конструкции языка, на их основе строятся предложения – более сложные конструкции. И те, и другие в общем виде являются цепочками символов, но предусматривают некоторые правила построения. Таким образом, необходимо указать эти правила, или, строго говоря, задать язык. В общем случае язык можно определить тремя способами:

– перечислением всех допустимых цепочек языка;

– указанием способа порождения цепочек языка (заданием грамматики языка);

– определением метода распознавания цепочек языка.

Первый из методов является чисто формальным и на практике не применяется, так как большинство языков содержат бесконечное число допустимых цепочек и перечислить их просто невозможно.

Второй способ предусматривает некоторое описание правил, с помощью которых строятся цепочки языка. Тогда любая цепочка, построенная с помощью этих правил из символов алфавита языка, будет принадлежать заданному языку.

Третий способ предусматривает построение некоторого логического устройства (распознавателя) – автомата, который на входе получает цепочку символов, а на выходе выдаёт ответ, принадлежит или нет эта цепочка заданному языку.

Грамматика – это описание способа построения предложений некоторого языка. Она относится ко второму способу определения языков – порождению цепочек символов. Грамматику языка можно описать различными способами. Например, можно использовать формальное описание грамматики, построенное на основе системы правил (или продукций).

Правило (или продукция) – это упорядоченная пара цепочек символов (α, β). В правилах важен порядок цепочек, поэтому их чаще записывают в виде α → β (или α ::= β). Такая запись читается как «α порождает β» или «α

по определению есть β». Грамматика языка программирования содержит правила двух типов: первые (определяющие синтаксические конструкции языка) довольно легко поддаются формальному описанию; вторые (определяющие семантические ограничения языка) обычно излагаются в неформальной форме.

Поэтому любое описание (или стандарт) языка программирования обычно состоит из двух частей: вначале формально излагаются правила построения синтаксических конструкций, а потом на естественном языке даётся описание семантических правил.

Язык, заданный грамматикой G, обозначается как L (G). Две грамматики, G и G', называются эквивалентными, если они определяют один и тот же язык: L (G) = L (G').

Две грамматики, G и G', называются почти эквивалентными, если заданные ими языки различаются не более чем на пустую цепочку символов: L (G) ∪ {ε} = L (G') ∪ {ε}.

Формально грамматика G определяется как четвёрка G (VT, VN, P, S), где:

– VT – множество терминальных символов, или алфавит терминальных символов;

– VN – множество нетерминальных символов, или алфавит нетерминальных символов;

– P – множество правил (продукций) грамматики вида α → β, где α ∈ (VN ∪ VT) +, β ∈ (VN ∪ VT)*;

– S – целевой (начальный) символ грамматики, S ∈ VN.

Алфавиты терминальных и нетерминальных символов грамматики не пересекаются: VN ∩ VT = ∅. Это значит, что каждый символ в грамматике может быть либо терминальным, либо нетерминальным, но не может быть терминальным и нетерминальным одновременно. Целевой символ грамматики – это всегда нетерминальный символ.

Множество V = VN ∪ VT называют полным алфавитом грамматики G. Множество терминальных символов VT содержит символы, которые входят в алфавит языка, порождаемого грамматикой. Как правило, символы из множества VT встречаются только в цепочках правых частей правил. Множество нетерминальных символов VN содержит символы, которые определяют слова, понятия, конструкции языка. Каждый символ этого множества может встречаться в цепочках как левой, так и правой частей правил грамматики. Во множестве правил грамматики может быть несколько правил, имеющих одинаковые левые части вида: α → β1, α → β2, …, α → βn.

Эти правила можно объединить вместе и записать в виде: α → β1 | β2 | … | βn. Одной строке в такой записи соответствует сразу n правил.

Такую форму записи правил грамматики называют формой Бэкуса-Наура . Форма Бэкуса-Наура (англ. Backus-Naur Form (BNF)), как правило, предусматривает также, что нетерминальные символы берутся в угловые скобки: < >.

Пример грамматики, которая определяет язык целых десятичных чисел со знаком в форме Бэкуса-Наура:

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

Рассмотрим составляющие элементы грамматики **G**:
множество терминальных символов **VT** содержит двенадцать элементов: десять десятичных цифр и два знака;
множество нетерминальных символов **VN** содержит три элемента: символы <*число*>, <*чс*> и <*цифра*>;
множество правил содержит 15 правил, которые записаны в три строки (то есть имеется только три различные левые части правил);
целевым символом грамматики является символ <*число*>.

Та же самая грамматика для языка целых десятичных чисел со знаком, в которой нетерминальные символы обозначены большими латинскими буквами (далее это будет часто применяться в примерах):

G' ({0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, +}, {*S*, *T*, *F*}, **P**, *S*)
P:
$S \rightarrow T \mid +T \mid -T$
$T \rightarrow F \mid TF$
$F \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Здесь изменилось только множество нетерминальных символов. Теперь VN = {S, T, F}. Язык, заданный грамматикой, не изменился – грамматики G и G' эквивалентны.

Особенность рассмотренных выше формальных грамматик в том, что они позволяют определить бесконечное множество цепочек языка с помощью конечного набора правил.

Возможность пользоваться конечным набором правил достигается в такой форме записи за счёт рекурсивных правил. Рекурсия в правилах грамматики выражается в том, что один из нетерминальных символов определяется сам через себя. Рекурсия может быть непосредственной (явной) – символ определяется сам через себя в одном правиле, либо косвенной (неявной) – тогда тоже самое происходит через цепочку правил.

Классификация грамматик и языков

Согласно классификации, предложенной американским лингвистом Ноамом Хомским, профессором Массачусетского технологического института, формальные грамматики классифицируются по структуре их правил. Если все без исключения правила грамматики удовлетворяют некоторой заданной структуре, то такую грамматику относят к определённому типу. Достаточно иметь в грамматике одно правило, не удовлетворяющее требованиям структуры правил, и она уже не попадает в заданный тип.

По классификации Хомского выделяют четыре типа грамматик.

Тип 0: грамматики с фразовой структурой На структуру их правил не накладывается никаких ограничений: для грамматики вида G (VT, VN, P, S), $V = VN \cup VT$, правила имеют вид $\alpha \rightarrow \beta$, где $\alpha \in V+$, $\beta \in V^*$. Это самый общий тип грамматик. В него попадают все без исключения формальные грамматики, но часть из них может быть также отнесена и к другим классификационным типам. Грамматики, которые относятся только к типу 0 и не могут быть отнесены к другим типам, являются самыми сложными по структуре. Практического применения грамматики, относящиеся только к типу 0, не имеют.

Тип 1: контекстно-зависимые (КЗ) и не укорачивающие грамматики В этот тип входят два основных класса грамматик:

Контекстно-зависимые грамматики G (VT, VN, P, S), $V = VN \cup VT$, имеют правила вида $\alpha 1 A \alpha 2 \rightarrow \alpha 1 \beta \alpha 2$, где $\alpha 1, \alpha 2 \in V^*$, $A \in VN$, $\beta \in V+$.

Не укорачивающие грамматики G (VT, VN, P, S), $V = VN \cup VT$, имеют правила вида: $\alpha \rightarrow \beta$, где $\alpha, \beta \in V+$, $|\beta| \geq |\alpha|$.

Структура правил КЗ-грамматик такова, что при построении предложений заданного ими языка один и тот же нетерминальный символ может быть заменён на ту или иную цепочку символов в зависимости от того контекста, в котором он встречается. Именно поэтому эти грамматики называют контекстно-зависимыми. Цепочки $\alpha 1$ и $\alpha 2$ в правилах грамматики обозначают контекст ($\alpha 1$ – левый контекст, а $\alpha 2$ – правый контекст), в общем случае любая из них (или даже обе) может быть пустой. Говоря иными словами, значение одного и того же символа может быть различным в зависимости от того, в каком контексте он встречается.

Не укорачивающие грамматики имеют такую структуру правил, что при построении предложений языка, заданного грамматикой, любая цепочка символов может быть заменена на цепочку символов не меньшей длины. Отсюда и название не укорачивающие. Доказано, что эти два класса грамматик эквивалентны. Это значит, что для любого языка, заданного КЗ-грамматикой, можно построить не укорачивающую грамматику, которая будет задавать эквивалентный язык, и, наоборот: для любого языка, заданного не укорачивающей грамматикой, можно построить КЗ-грамматику, которая будет задавать эквивалентный язык.

Тип 2: контекстно-свободные (КС) грамматики

Не укорачивающие контекстно-свободные (НКС) грамматики G (VT, VN, P, S), $V = VN \cup VT$, имеют правила вида $A \rightarrow \beta$, где $A \in VN$, $\beta \in V+$. Такие грамматики называют НКС-грамматиками, поскольку видно, что в правой части правил у них должен всегда стоять как минимум один символ. Существует также почти эквивалентный им класс грамматик – укорачивающие контекстно-свободные (УКС) грамматики G (VT, N, P, S), $V = VN \cup VT$, правила которых могут иметь вид: $A \rightarrow \beta$, где $A \in VN$, $\beta \in V^*$. Эти два класса составляют тип контекстно-свободных грамматик. Разница между ними заключается лишь в том, что в УКС-грамматиках в правой части

правил может присутствовать пустая цепочка, а в НКС-грамматиках – нет. Отсюда ясно, что язык, заданный НКС-грамматикой, не может содержать пустой цепочки.

Тип 3: регулярные грамматики К типу регулярных относятся два эквивалентных класса грамматик: леволинейные и праволинейные.

Леволинейные грамматики G (VT, VN, P, S), V = VN ∪ VT, могут иметь правила двух видов: A → Bγ или A → γ, где A, B ∈ VN, γ ∈ VT*. 7 В свою очередь, праволинейные грамматики G (VT, VN, P, S), V = VN ∪ VT, могут иметь правила тоже двух видов: A → γB или A → γ, где A, B ∈ VN, γ ∈ VT*.

Языки классифицируются в соответствии с типами грамматик, с помощью которых они заданы. Если язык L может быть задан грамматиками G1 и G2, относящимися к типу 1 (КЗ), грамматикой G3, относящейся к типу 2 (КС), и грамматикой G4, относящейся к типу 3 (регулярные), сам язык должен быть отнесён к типу 3 и является регулярным языком.

Язык L называют КС-языком, т.к. существует КС-грамматика, его описывающая. Но он не является регулярным языком, т.к. не существует регулярной грамматики, описывающей этот язык.

Задачи для самостоятельного выполнения

1. Дана грамматика. Постройте вывод заданной цепочки:

a) $S \to T \mid T{+}S \mid T{-}S$
$T \to F \mid F{*}T$
$F \to a \mid b$
Цепочка $a{-}b{*}a{+}b$

b) $S \to aSBC \mid abC$
$CB \to BC$
$bB \to bb$
$bC \to bc$
$cC \to cc$
Цепочка $aaabbbccc$

2. Какой язык порождается грамматикой с правилами:

а) $S \to aaCFD$
$AD \to D$
$F \to AFB \mid AB$
$Cb \to bC$
$AB \to bBA$
$CB \to C$
$Ab \to bA$
$bCD \to \varepsilon$

б) $S \to A\perp \mid B\perp$
$A \to a \mid Ba$
$B \to b \mid Bb \mid Ab$

3. Построить грамматику, порождающую язык:

а) $\mathbf{L} = \{a^n b^m c^k \mid n, m, k > 0\}$
б) $\mathbf{L} = \{0^n (10)^m \mid n, m \geq 0\}$
в) $\mathbf{L} = \{a_1 a_2 \ldots a_n a_n \ldots a_2 a_1 \mid a_i \in \{0, 1\}\}$

4. К какому типу по Хомскому относится грамматика с правилами:

а) $S \to 0A1 \mid 01$
$0A \to 00A1$
$A \to 01$

б) $S \to Ab$
$A \to Aa \mid ba$

5. Эквивалентны ли грамматики с правилами:

$S \to aSL \mid aL$	и	$S \to aSBc \mid abc$
$L \to Kc$		$cB \to Bc$
$cK \to Kc$		$bB \to bb$
$K \to b$		

6. Построить КС-грамматику, эквивалентную грамматике с правилами:

$S \to AB \mid ABS$
$AB \to BA$
$BA \to AB$
$A \to a$
$B \to b$

7. Построить регулярную грамматику, эквивалентную грамматике с правилами:

$S \to A \,.\, A$
$A \to B \mid BA$
$B \to 0 \mid 1$

Контрольные вопросы

1. Какие операции можно выполнять над цепочками символов?
2. Какие существуют методы задания языков?
3. Что такое грамматика языка?
4. Как выглядит описание грамматики в форме Бэкуса-Наура? Какие ещё формы описания грамматик существуют?
5. На основе какого принципа классифицируются грамматики в классификации Н. Хомского?
6. Какие типы грамматик выделяют по классификации Н. Хомского?
7. Какие типы языков выделяют по классификации Н. Хомского? Как классификация языков соотносится с классификацией грамматик?

Тема 2. Регулярные грамматики и конечные автоматы

Краткие теоретические сведения

Определение. Порождающая грамматика G = <N, T, P, S>, правила которой имеют вид: A $\to$ aB или C $\to$ b, где A, B, C $\in$ N; a, b $\in$ T называется *регулярной* (*автоматной*).

Язык L(G), порождаемый автоматной грамматикой, называется *автоматным* (*регулярным*) языком или языком с конечным числом состояний.

Пример 1. Идентификатором является последовательность, состоящая из букв и цифр, и первым символом идентификатора может быть только буква.

Класс идентификаторов описывается следующей порождающей регулярной грамматикой G = <N, T, P, S>, в которой

N = {I, K}, T = {б, ц}, S = {I},

P = {1.I $\rightarrow$б

2.I $\rightarrow$бк

3.K $\rightarrow$бк

4.K $\rightarrow$цк

5.K $\rightarrow$б

6.K $\rightarrow$ц}

Здесь б, ц — обобщенные терминальные символы для обозначения букв и цифр соответственно.

Процесс порождения идентификатора «ббцбц» описывается следующей последовательностью подстановок:

$$I \overset{2}{\Rightarrow} бK \overset{3}{\Rightarrow} ббK \overset{4}{\Rightarrow} ббцK \overset{3}{\Rightarrow} ббцбK \overset{6}{\Rightarrow} ббцбц$$

Необходимо отметить, что данная грамматика не единственная, которая описывает язык идентификаторов.

Однако основной задачей ЛА является не порождение лексических единиц, а их распознавание. Математической моделью процесса распознавания регулярного языка является вычислительное устройство, которое называется *конечным автоматом* (КА). Термин «конечный» подчеркивает то, что вычислительное устройство имеет фиксированный и конечный объем памяти и обрабатывает последовательность входных символов, принадлежащих некоторому конечному множеству. Существуют различные типы КА, если функцией выхода КА (результатом работы) является лишь указание на то, допустима или нет входная последовательность символов, такой КА называют *конечным распознавателем*. Именно этот тип КА в дальнейшем мы и будем рассматривать.

Определение. *Конечным автоматом* (КА) называется следующая пятерка:

$A = < V, Q, \delta, q_0, F >$,

где

$V = \{a_1, a_2, \ldots, a_m\}$ – входной алфавит (конечное множество символов);

$Q = \{q_0, q_1, \ldots, q_{n-1}\}$ – алфавит состояний (конечное множество символов);

$\delta : Q \times V \rightarrow Q$ – функция переходов;

$q_0 \in Q$ – начальное состояние конечного автомата;

$F \subseteq Q$ – множество заключительных состояний.

На содержательном уровне функционирование КА можно представить следующим образом.

Имеется бесконечная лента, разбитая на ячейки, в каждой из которых может находиться один символ из V. На ленте записана цепочка $\alpha \in V^*$. Ячейки слева и справа от цепочки не заполнены. Имеется конечное устройство управления (УУ) с читающей головкой, которое может последовательно считывать символы с ленты, передвигаясь вдоль ленты

слева направо. При этом УУ может находиться в каком-либо одном состоянии из Q. Начинает свою работу УУ всегда в начальном состоянии $q_0 \in Q$, а завершает в одном из заключительных состояний $F \subseteq Q$; Каждый раз, переходя к новой ячейке на ленте, УУ переходит в новое состояние в соответствии с функцией δ. Схематически конструкция КА показана на рисунке 1.

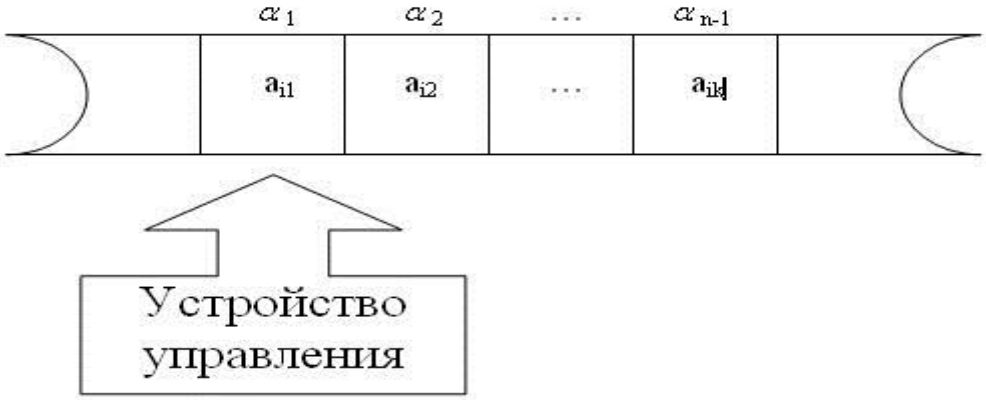

Рисунок 1- Схема конечного автомата

Отображение δ (функцию переходов КА) можно представить различными способами. Основные из них:

· совокупность команд;

· диаграмма состояний;

· матрица переходов.

Команда конечного автомата записывается следующим образом:

$(q_i, a_j) \rightarrow q_k$, где: $q_i, q_k \in Q$; $aj \in V$.

Данная команда обозначает, что КА находится в состоянии qi, читает с ленты символ aj и переходит в состояние qk.

Графически команда представляется в виде дуги графа, идущей из вершины qi в вершину qk и помеченной символом aj входного алфавита:

Графическое представление всего отображения называют диаграммой состояний конечного автомата. Диаграмма состояний в этом случае — это ориентированный граф, вершинам которого поставлены в соответствие символы из множества состояний Q, а дугам — команды отображения δ.

Если КА оказывается в ситуации (qi, aj), которая не является левой частью какой-либо команды, то он останавливается. Если же УУ считает все символы цепочки α, записанной на ленте, и при этом перейдет в заключительное состояние $q_i \in F$, то говорят, что цепочка α допускается конечным автоматом.

Матрица переходов КА строится следующим образом: столбцы матрицы соответствуют символам из входного алфавита, строки — символам из алфавита состояний, а элементы матрицы соответствуют состояниям, в которые переходит КА для данной комбинации входного символа и символа состояния.

Переход от регулярной грамматики к конечному автомату и обратно

Как было указано ранее, КА является хорошей математической моделью для представления алгоритмов распознавания лексем в ЛА, в особенности для лексем из бесконечных классов. При этом источником, по которому строится КА, является регулярная грамматика.

Пусть задана регулярная грамматика G = <N, T, P, S>, правила которой имеют вид:

Ai $\rightarrow$ aj Ak или Ai $\rightarrow$ aj., где Ai, Ak $\in$ N и aj $\in$ T.

Тогда конечный автомат A = <V, Q, δ, q0, F>, допускающий тот же самый язык, что порождает регулярная грамматика G, строится следующим образом:

1) V = T;

2) Q = N $\cup$ {Z}, Z $\notin$ N, Z $\notin$ T, Z – заключительное состояние КА;

3) q_0 = {S};

4) F = {Z};

5) отображение δ строится в виде:

а) каждому правилу подстановки в грамматике G вида $A_i \rightarrow a_j A_k$ ставится в соответствие команда $(A_i, a_j) \rightarrow A_k$;

б) каждому правилу подстановки вида $A_i \rightarrow a_j$ ставится в соответствие команда $(A_i, a_j) \rightarrow Z$.

Пример 2. Построить КА для грамматики из примера 1.

Имеем A=<V,Q, δ, q0, F>,

1) где V = T = {б, ц}

2) Q = N $\cup$ {Z} = {I, K, Z}

3) q0 = {S} = {I}

4) F = {Z}

1) δ:

а) в виде совокупности команд:

(I, б) $\rightarrow$ Z

(I, б) $\rightarrow$ K

(K, б) $\rightarrow$ K

(K, ц) $\rightarrow$ K

(K, б) $\rightarrow$ Z

(K, ц) $\rightarrow$ Z

б) в виде диаграммы состояний (рисунок 2)

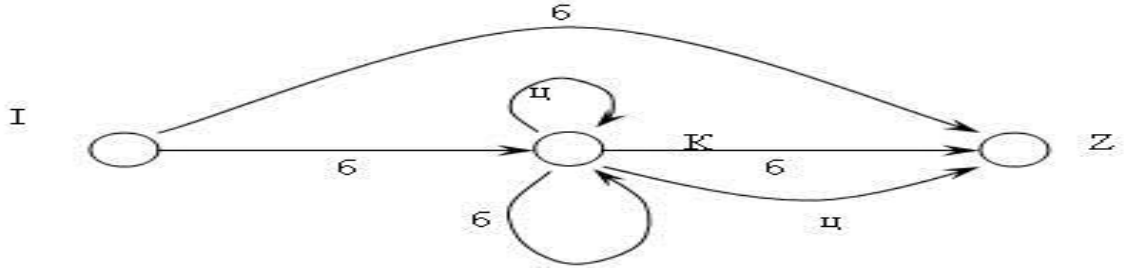

Рисунок 2 – Диаграмма состояний КА для идентификаторов

Допустим и обратный переход. Так конечному автомату A = <V, Q, δ, q0, F> можно поставить в соответствие регулярную грамматику G = <N, T, P, S>, у которой:

1) T = V;

2) N = Q;

3) S = {q0};

4) множество правил подстановки P строится следующим образом:

каждой команде автомата (qi,aj)$\rightarrow$qk ставится в соответствие правило подстановки qi $\rightarrow$ ajqk, если qk $\in$ Q, либо еще одно правило qi $\rightarrow$ aj, если qk $\in$ F.

Задачи для самостоятельного выполнения

1. Какой язык допускается конечным автоматом $M = (\{q_0\}, \{a, b\}, \varnothing, q_0, \{q_0\})$?

2. Построить недетерминированный конечный автомат, допускающий цепочки в алфавите {1, 2}, у которых последний символ цепочки уже появлялся в ней раньше. Построить эквивалентный детерминированный конечный автомат. Построить аналогичные конечные автоматы в алфавите {1, 2, 3}.

3. Построить конечный автомат, допускающий язык $\{xy\} \cup \{yx\}$, где $x \in \{a\}^* \setminus \varepsilon y \in \{b\}^* \setminus \varepsilon$

4. Построить детерминированный конечный автомат, допускающий язык L всех слов в алфавите {0, 1}, содержащих чётное число единиц и не чётное число нулей;

Контрольные вопросы

1. Какие грамматики относятся к регулярным? Назовите два класса регулярных грамматик.

2. Можно ли граф переходов конечного автомата использовать для однозначного определения данного автомата?

3. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду?

4. Всякая ли регулярная грамматика является однозначной?

5. Если язык задан КА, то можно ли для него построить регулярное выражение?

6. Если язык задан КА, то может ли он быть задан КС-грамматикой?

Тема 3. КС – языки и автоматы с магазинной памятью

Краткие теоретические сведения

Описание КС-языка с помощью порождающей КС-грамматики не является описанием алгоритма порождения предложений этого языка.

Правила подстановки грамматики — это не последовательность предписаний, а совокупность разрешений, причем порядок применения правил в грамматике произволен, тогда как в алгоритме должен быть задан жесткий порядок применения отдельных инструкций.

Получение алгоритмического описания процесса распознавания языка является одной из первоочередных задач при разработке блока синтаксического анализа транслятора.

Одним из способов описания алгоритма распознавания языка является задание его в виде некоторого распознающего устройства. Для КС-языков такими устройствами являются магазинные автоматы (или, иначе, автоматы с магазинной памятью, МП-автоматы).

Определение 1. Недетерминированным МП-автоматом называется семерка $M = <A, Q, Г, \delta, q_0, z_0, F>$,

где A — конечное множество входных символов (входной алфавит);

Q — конечное множество внутренних состояний (алфавит состояний);

Г — конечное множество магазинных символов;

$q_0 \in Q$ — начальное состояние автомата;

$z_0 \in Г$ — начальный (первый) символ в магазинной памяти;

$F \subseteq Q$ — множество заключительных состояний;

δ — отображение QxAxГ в множество подмножеств Qx Г*, т.е. отображение вида $\delta : QxAxГ \rightarrow (QxГ^*)$.

Схема МП-автомата показана на рис. 3.4. Автомат имеет конечное множество состояний, конечное множество входных символов и неограниченную снизу вспомогательную ленту (называемую лентой магазинной памяти или магазинной памятью). «Дно» магазина (самый нижний символ) отмечается специальным символом, называемым маркером дна. Пусть это будет символ #. Магазинная память определяется свойством «первым введен — последним выведен». При записи символа в магазин, его содержимое сдвигается на одну ячейку «вниз», а на освободившееся место записывается требуемый символ. В этом случае говорят, что символ «вталкивается» в магазин.

Рисунок 3 – Схема автомата с магазинной памятью

Для чтения доступен только самый последний (т.е. верхний) символ магазина. Этот символ после чтения может либо остаться в магазине, либо быть удален из него, т.е. «вытолкнут» из магазина. За один такт работы МП-автомата из магазина можно удалять не более одного символа.

Каждый шаг работы МП-автомата задается множеством правил перехода автомата из одних состояний в другие. Переходы в общем случае определяются:

– состоянием МП-автомата;

– верхним символом магазина;

– текущим входным символом.

Множество правил перехода называется *управляющим устройством*. В зависимости от получаемой информации, управляющее устройство реализует один такт работы МП-автомата, который включает в себя три операции (в скобках указано обозначение операций):

1) операции над магазином:

– втолкнуть в магазин определенный символ (ВТОЛКНУТЬ (А), где А — магазинный символ);

– вытолкнуть верхний символ из магазина (ВЫТОЛКНУТЬ);

– оставить содержимое магазина без изменений;

2) операции над состоянием:

– перейти в заданное новое состояние (СОСТОЯНИЕ (S), где S — следующее состояние;

– остаться в прежнем состоянии;

3) операции над входом:

– перейти к следующему входному символу и сделать его текущим (СДВИГ);

– оставить данный входной символ текущим, т.е. держать его до следующего шага (ДЕРЖАТЬ).

Обработку входной цепочки МП-автомат начинает в некотором выделенном состоянии при определенном содержимом магазина. Затем автомат выполняет операции, задаваемые его управляющим устройством. При этом выполняется либо завершение обработки, либо переход в новое состояние. Если выполняется переход, то он дает новый верхний символ магазина, новый текущий символ, автомат переходит в новое состояние и цикл работы автомата повторяется.

МП-автомат называется МП-распознавателем, если у него два выхода ДОПУСТИТЬ и ОТВЕРГНУТЬ.

Рассмотрим МП-автомат, распознающий слова языка $L = \{0^n1^n \mid n \geq 1\}$, т.е. слова, состоящие из n нулей, вслед за которыми идет ровно такое же количество единиц. Для этого автомата $A = \{0, 1, \vdash\}$, $Q = \{q_1, q_2\}$, $\Gamma = \{z, \#\}$, $\vdash$ — символ конца цепочки на входной ленте, # — маркер дна магазина, q_1 — начальное состояние, # — начальное содержимое магазина. Работа управляющего устройства для данного автомата (а это МП-распознаватель) осуществляется в соответствии с управляющими таблицами, изображенными на рисунке 4, а и б для состояний q_1, и q_2 соответственно. В таблицах показаны действия автомата для каждого сочетания входного символа и верхнего символа магазина.

Работа МП-автомата при распознавании конкретной цепочки описывается обычно в виде последовательности конфигураций МП-автомата. При линейном представлении конфигураций слева изображается магазин, в середине – состояние, справа – необработанная часть входной цепочки. Ниже показана такая последовательность конфигураций при анализе МП-автоматом входного слова 000111.

Таблица 1 – Управляющая таблица для состояния q_1 (а) и q_2 (б) МП-распознавателя языка $L = \{0^n1^n \mid n \geq 1\}$

МАГАЗИН	СОСТОЯНИЕ	ВХОДНАЯ ЛЕНТА
#	q_1	000111⊢
#z	q_1	00111⊢
#zz	q_1	0111⊢
#zzz	q_1	111⊢
#zz	q_2	11⊢
#z	q_2	1⊢
#	q_2	⊢
ДОПУСТИТЬ		

а)

Состояние q_1

Магазин	Вход		
	0	1	⊢
Z	СОСТОЯНИЕ (q_1) ВТОЛКНУТЬ (z) СДВИГ	СОСТОЯНИЕ (q_2) ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
#	СОСТОЯНИЕ (q_1) ВТОЛКНУТЬ (z) СДВИГ	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ

Состояние q_2

Магазин	Вход		
	0	1	⊢
Z	ОТВЕРГНУТЬ	СОСТОЯНИЕ (q_2) ВЫТОЛКНУТЬ СДВИГ	ОТВЕРГНУТЬ
#	ОТВЕРГНУТЬ	ОТВЕРГНУТЬ	ДОПУСТИТЬ

Другой способ описания (более формальный) МП-автомата — в виде команд. Команда записывается в виде:

$(q, a, z) \rightarrow \{(q_1, \gamma_1), \ldots, (q_m, \gamma_m)\}$,

где $q, q_1, \ldots, q_m \in Q$; $a \in A$; $z \in \Gamma$; $\gamma_1, \gamma_2, \ldots, \gamma_m \in \Gamma^*$.

Интерпретируется следующим образом: МП-автомат находится в состоянии **q**, считывает входной символ **a** и верхний символ магазина **z,** переходит в состояние $\mathbf{q_i}$, заменяя в магазине символ **z** на символ γ_i, $1 \le i \le m$ и продвигает входную головку на один символ.

Если же при выполнении команды сдвига входной ленты не выполняется, то команда записывается в виде:

$(q, \lambda, z) \rightarrow \{(q_1, \gamma_1), \ldots, (q_m, \gamma_m)\}$,

и используется лишь для изменения магазина.

Различают детерминированные и недетерминированные МП-автоматы. Если среди команд МП-автомата нет двух таких, у которых совпадают части, стоящие слева от стрелки, и не совпадают части, стоящие справа от стрелки, то МП-автомат называют *детерминированным*. В противном случае, т.е. когда для текущих входном и магазинном символов и заданного состояния, возможны переходы автомата в различные состояния, его называют *недетерминированным*.

Задачи для самостоятельного выполнения

5. Какой язык допускается конечным автоматом $M = (\{q_0\}, \{a, b\}, \varnothing, q_0, \{q_0\})$?

6. Построить недетерминированный конечный автомат, допускающий цепочки в алфавите {1, 2}, у которых последний символ цепочки уже появлялся в ней раньше. Построить эквивалентный детерминированный конечный автомат. Построить аналогичные конечные автоматы в алфавите {1, 2, 3}.

7. Построить конечный автомат, допускающий язык $\{xy\} \cup \{yx\}$, где $x \in \{a\}^* \setminus \varepsilon y \in \{b\}^* \setminus \varepsilon$

8. Построить детерминированный конечный автомат, допускающий язык L всех слов в алфавите {0, 1}, содержащих чётное число единиц и нечётное число нулей;

Контрольные вопросы

1. Какие грамматики относятся к регулярным? Назовите два класса регулярных грамматик.

2. Можно ли граф переходов конечного автомата использовать для однозначного определения данного автомата?

3. Всегда ли недетерминированный КА может быть преобразован к детерминированному виду?

4. Всякая ли регулярная грамматика является однозначной?

5. Если язык задан КА, то можно ли для него построить регулярное выражение?

6. Если язык задан КА, то может ли он быть задан КС-грамматикой?

Тема 4. Преобразование грамматик

Преобразования контекстно-свободных грамматик

Для одного и того же КС-языка могут существовать различные грамматики, поэтому возникает проблема выбора грамматики, наиболее подходящей по тем или иным свойствам, или проблема эквивалентного преобразования грамматики к нужному виду. Универсальных методов эквивалентных преобразований КС-грамматик не существует из-за неразрешимости алгоритмических проблем распознавания эквивалентности КС-грамматик и существенной неоднозначности КС-языков.

Рассмотрим несколько эквивалентных преобразований, которые с некоторой точки зрения улучшают грамматику.

Символ $X \in VT \cup VN$ называется полезным в КС-грамматике $G = VT, VN, P, S$, если X появляется в процессе вывода из аксиомы S некоторого слова $w \in L(G)$, т.е. $S \; X \; w \Rightarrow * \Rightarrow * \alpha \beta$, где $* \; w \in VT$. Если символ X не является полезным, то он называется бесполезным. Очевидно, что исключение бесполезных символов из грамматики не изменяет порождаемого языка, поэтому все бесполезные символы можно удалить.

Символ $X \in VN$ называется порождающим, если $X \; w \Rightarrow *$ для некоторой терминальной цепочки w.

Символ $X \in VT \cup VN$ называется достижимым, если существует вывод $S \; \alpha X \beta \Rightarrow *$ для некоторых α и β. Очевидно, что полезный нетерминальный символ грамматики является одновременно и порождающим, и достижимым. Можно доказать, что если сначала удалить из грамматики непорождающие символы, а затем недостижимые, то останутся только полезные.

Алгоритм 1. Устранение непорождающих символов.

Шаг 1. Положить $N_0 = \varnothing$ и $i = 1$.

Шаг 2. $Ni \{ = A | \; A \rightarrow \alpha \in P \; \alpha \in (VT \cup N_{i-1})^*\} \cup N_{i-1}$.

Шаг 3. Если $N_i \neq N_{i-1}$, то положить $i = i + 1$ и перейти к шагу 2. В противном случае положить $Ne = N_i$ - множество порождающих символов.

Шаг 4. Если $N_e \subset VN$, т.е. имеются непорождающие символы, следует перейти к грамматике $G_1 = <VT, VN', P', S>$, в которой $VN = Ne \cap VN'$, и P' состоит из правил множества P, содержащих нетерминалы только из Ne.

С помощью алгоритма может быть решена проблема пустоты для КС-языков. Язык L(G) непуст тогда и только тогда, когда S - порождающий символ, т. е. когда имеется вывод $S \Rightarrow * \; w$ для некоторого слова $w \in V^*T$. Другими словами, язык L(G) непуст тогда и только тогда, когда $S \in Ne$.

Пример 1. Рассмотрим грамматику $G = <\{0,1\},\{S, A,B\},P, S>$, где P состоит из правил

$S \to 0|A$

$A \to AB$

$B \to 1$.

Применяя алгоритм устранения непорождающих символов к грамматике G , получаем множество $Ne = \{S, B\}$. Так как $S \in Ne$, то язык L(G) непуст. Нетерминальный символ $A \notin Ne$, значит, он является непорождающим. Удалим из исходной грамматики все правила, содержащие символ A, в результате получим грамматику $G1 = <\{0,1\},\{S,B\},\{S \to 0, B \to 1\}, S>$, которая эквивалентна G и не содержит непорождающих символов.

Алгоритм 2. Устранение недостижимых символов.

Шаг 1. Положить $V0 = \{S\}$ и $i = 1$.

Шаг 2. $V_i = \{X \mid A \to \alpha X \beta \in P$ и $A \in V_{i-1}\} \cup V_{i-1}$.

Шаг 3. Если $V_i \neq V_{i-1}$, то положить $i = i + 1$ и перейти к шагу 2. В противном случае положить $Ve = V_i$ - множество достижимых символов.

Шаг 4. Если $V_e \subset (VT \cup VN)$, т.е. имеются недостижимые символы, следует перейти к грамматике $G' = <VT', VN', P', S>$, в которой $VT' = Ve \cap VT$, $VN' = Ve \cap VN$, и P' состоит из правил множества P , содержащих только символы из Ve .

Пример 2. Применив алгоритм устранения недостижимых символов к грамматике $G1 = <\{0,1\},\{S,B\},\{S \to 0, B \to 1\}, S>$, рассмотренной в примере 1, получим $Ve = V_2 = \{S,0\}$, а это означает, что терминальный символ 1 и нетерминальный символ B - недостижимы. После удаления этих символов получим грамматику $G' = <\{0\},\{S\},\{S \to 0\}, S>$, эквивалентную исходной грамматике G и не содержащую бесполезных символов.

Пример 3. Снова рассмотрим грамматику $G = <\{0,1\},\{S, A,B\},P, S>$ из примера 1, где P состоит из правил

$S \to 0|A$

$A \to AB$

$B \to 1$.

Теперь изменим порядок удаления бесполезных символов и начнем с алгоритма устранения недостижимых символов. В этом случае окажется, что все символы достижимы. Затем, применив алгоритм устранения непорождающих символов, получим грамматику $G1 = <\{0,1\},\{S,B\},\{S \to 0, B \to 1\}, S>$, отличную от $G' = <\{0\},\{S\},\{S \to 0\}, S>$.

Алгоритм 3. Устранение бесполезных символов.

Шаг 1. Применив к грамматике $G = <VT, VN, P, S>$ алгоритм 1 устранения непорождающих символов, получить $G1 = <VT, VN', P', S>$.

Шаг 2. Применив к грамматике $G1 = <VT, VN', P', S>$ алгоритм 2 устранения недостижимых символов, получить $G' = <VT', VN', P', S>$.

Теорема 1. Грамматика G' , которую строит алгоритм 3 по грамматике G , не содержит бесполезных символов, и $L(G) = L(G')$.

λ -правилом называется правило вида $A \to \lambda$, где $A \in VN$ и λ - пустое слово.

Контекстно-свободная грамматика G = <VT ,VN , P, S> называется грамматикой без λ -правил (или неукорачивающей), если либо

(1) P не содержит λ -правил (в случае, когда $\lambda \notin L(G)$), либо

(2) есть точно одно λ -правило $S \to \lambda$, и S не встречается в правых частях остальных правил из P (в случае, когда $\lambda \in L(G)$).

Алгоритм 4. Преобразование в грамматику без λ -правил.

Пусть G = <VT ,VN , P, S> - КС-грамматика с λ -правилами. Символ $X \in VN$ называется λ -порождающим, если $X \Rightarrow *\lambda$.

Шаг 1. Положить $N_0 = \{A | A \to \lambda \in P\}$ и i = 1.

Шаг 2. $N_i = \{B | B \to \alpha \in P\ \alpha \in N_{i-1}\} \cup N_i$ и i=1/

Шаг 3. Если $N_i \neq N_{i-1}$, то положить i = i +1 и перейти к шагу 2. В противном случае положить $N_\lambda = N_i$ - множество λ -порождающих символов.

Шаг 4. Построить множество правил P′ :

(1) Если $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \ldots \alpha_{k-1} B_k \alpha_k \in P$, где $k \geq 0$ и $B_i \in N_\lambda$ для i = 1, k , но ни один символ в словах (j =0, k) не принадлежит N_λ , то включить в P′ все правила вида $A \to \alpha_0 X_1 \alpha_1 X_2 \alpha_2\ \alpha_{k-1} X_k \alpha_k$, где X_i - либо B_i , либо λ , но не включать правило $A \to \lambda$ (это могло бы произойти, если все αi равны λ).

(2) Если $S \in N_\lambda$, то включить в P′ правила $S' \to \lambda | S$, где S′ - новый символ, и положить V′N = {S′}∪ VN. В противном случае положить V′N =VN и S′ = S .

Шаг 5. Положить G′ = <V′T,V′N,P′,S′> ; G' - эквивалентная КС-грамматика без λ -правил.

Теорема 2. Грамматика G′ , которую строит алгоритм 4 по грамматике G , является грамматикой без λ -правил, и L(G) = L(G′).

Пример .4. Рассмотрим грамматику $G = \langle \{a, b\}, \{S, A, B\}, P, S \rangle$, где P состоит из правил

$$S \to AB$$
$$A \to aAA \mid \lambda$$
$$B \to bBB \mid \lambda.$$

Сначала найдем λ-порождающие символы. $N_0 = \{A, B\}$ и $N_\lambda = \{S, A, B\}$. Построим теперь множество правил P'. Так как $S \to AB \in P$, то в P' следует включить следующие три правила: $S \to AB \mid A \mid B$. Правило $A \to aAA \in P$, значит в P' следует включить следующие правила: $A \to aAA \mid aA \mid a$. Аналогично, правило $B \to bBB \in P$ влечет правила в P': $B \to bBB \mid bB \mid b$. Символ $S \in N_\lambda$, поэтому следует включить в P' правила $S' \to \lambda \mid S$. Таким образом, грамматика $G' = \langle \{a, b\}, \{S', S, A, B\}, P', S' \rangle$, где P' состоит из правил

$$S' \to \lambda \mid S$$
$$S \to AB \mid A \mid B$$
$$A \to aAA \mid aA \mid a$$
$$B \to bBB \mid bB \mid b,$$

является грамматикой без λ-правил, причем $L(G) = L(G')$.

Цепным правилом называется правило вида $A \to B$, где $A, B \in VN$, .

Алгоритм 5. Устранение цепных правил.

Пусть G = <VT ,VN , P, S> - КС-грамматика без λ -правил.

Шаг 1. Для каждого $A \in VN$ построить $N_A = \{B \mid A\} \Rightarrow {}^*B\}$ следующим образом:

(1) Положить $N_0 = \{ A\}$ и i = 1.

(2) Положить $N_i = \{C \mid B \to C \in P$ и $B \in N_{i-1}\} \cup N_{i-1}$.

(3) Если $N_i \neq N_{i-1}$, то положить i = i +1 и перейти к шагу 2. В противном случае положить $N_A = N_i$.

Шаг 2. Построить множество правил P′ : если $B \to \alpha \in P$ и не является цепным правилом, то включить в P′ правило $A \to \alpha$ для всех таких A , что $B \in NA$.

Шаг 3. Положить G′ = <VT ,VN , P′ , S> ; G' - эквивалентная КС-грамматика без λ -правил и без цепных правил.

Теорема 3. Грамматика G′, которую строит алгоритм 5 по грамматике G , не имеет цепных правил, и L(G) = L(G′).

Пример 5. Рассмотрим грамматику $G = \langle \{a,b\}, \{S,A,B,C\}, P, S \rangle$, где P состоит из правил

$$S \to aB \mid bA \mid B$$
$$A \to aa \mid bBb \mid C$$
$$B \to ab \mid C \mid aAb$$
$$C \to aBa \mid ab.$$

Для нетерминального символа S находим множество $N_S = \{S,B,C\}$, для A - множество $N_A = \{A,C\}$, для B - множество $N_B = \{B,C\}$, для C - множество $N_C = \{C\}$. Построим новое множество правил P':

$$S \to aB \mid bA \mid ab \mid aBa \mid aAb$$
$$A \to aa \mid bBb \mid aBa \mid ab$$
$$B \to ab \mid aBa \mid aAb$$
$$C \to aBa \mid ab.$$

Грамматика $G' = \langle \{a,b\}, \{S,A,B,C\}, P', S \rangle$ не содержит цепных правил и эквивалентна исходной грамматике G.

КС-грамматика G = VT ,VN , P, S называется приведенной грамматикой (или грамматикой в приведенной форме), если множество ее правил вывода P не содержит λ -правил, цепных правил и бесполезных символов. Рассмотренные выше алгоритмы 1 – 5 любую КС-грамматику позволяют преобразовать к эквивалентной приведенной КС-грамматике.

Задачи для самостоятельного выполнения

1.Найти грамматику, не содержащую бесполезных символов и эквивалентную грамматике с правилами

S → A|C

A → aB |bS b

B → AB |Ba

C → AS |b .

1.2.3. Найти грамматику, не содержащую бесполезных символов и эквивалентную грамматике с правилами

S →aBCa |bADb

A→aAa |bEa

B → aBa |bSBb

C →cB cE

D → aEa |bD bBa

E →bS aba.

3. Найти грамматику, не содержащую бесполезных символов и эквивалентную грамматике с правилами

S → aSb|bA aDEa aFE

A → aB |Bba aa

B →Ca |bC

C → bE |Bb Da

D → ab |ba

E → aEA F → ab .

4. Найти приведенную грамматику, эквивалентную грамматике с правилами

S → ASB |λ

A→aAS |a

B → SbS | A bb.

5.Найти приведенную грамматику, эквивалентную грамматике с правилами

S → AAA |B

A → aA|B

B → λ .

6.Найти приведенную грамматику, эквивалентную грамматике с правилами

S → ABa |aAb

A → aA|ab λ

B →ba |Ba λ .

7. Найти приведенную грамматику, эквивалентную грамматике с правилами

S → aB |bA B

A → bBb |C aa
B → aAb |C ab
C →aBa ab.

8. Найти приведенную грамматику, эквивалентную грамматике с правилами

S → aAa |bBb BB
A →C
B → S | A
C →S λ

Контрольные вопросы

1. Эквивалентные грамматики
2. Непорождающие символы, их устранение
3. Недостижимые символы, их устранение
4. Бесполезные символы, их устранение

Тема 5. Трансляторы

Каждая вычислительная машина имеет свой собственный язык программирования – язык команд или машинный язык – и может исполнять программы, записанные только на этом языке. С помощью машинного языка, в принципе, можно описать любой алгоритм, но затраты на программирование будут чрезвычайно велики. Это обусловлено тем, что машинный язык позволяет описывать и обрабатывать лишь примитивные структуры данных – бит, байт, слово. Программирование в машинных кодах требует чрезмерной детализации программы и доступно лишь программистам, хорошо знающим устройство и функционирование ЭВМ. Преодолеть эту трудность и позволили языки высокого уровня (Фортран, ПЛ/1, Паскаль, Си, Ада и др.) с развитыми структурами данных и средствами их обработки, не зависящими от языка конкретной ЭВМ.

Алгоритмические языки высокого уровня дают возможность программисту достаточно просто и удобно описывать алгоритмы решения многих прикладных задач. Такое описание называют *исходной программой*, а язык высокого уровня — *входным языком*.

Языковым процессором называют программу на машинном языке, позволяющую вычислительной машине понимать и выполнять программы на входном языке. Различают два основных типа языковых процессоров: интерпретаторы и трансляторы.

Интерпретатор – это программа, которая в качестве входа допускает программу на входном языке и по мере распознавания конструкций входного языка реализует их, выдавая на выходе результаты вычислений, предписанные исходной программой.

Транслятор – это программа, которая допускает на входе исходную программу и порождает на своем выходе программу, функционально-

эквивалентную исходной, называемую *объектной*. Объектная программа записывается на объектном языке. В частном случае, объектным языком может служить машинный язык, и в этом случае, полученную на выходе транслятора программу можно сразу же выполнить на ЭВМ (проинтерпретировать). При этом ЭВМ является интерпретатором объектной программы в машинных кодах. В общем случае объектный язык не обязательно должен быть машинным или близким к нему (автокодом). В качестве объектного языка может служить некоторый *промежуточный язык* – язык, лежащий между входным и машинным языками.

Если в качестве объектного языка используется промежуточный язык, то возможны два варианта построения транслятора.

•Первый вариант – для промежуточного языка имеется (или разрабатывается) другой транслятор с промежуточного языка на машинный, и он используется в качестве последнего блока проектируемого транслятора.

•Второй вариант построения транслятора с использованием промежуточного языка – построить интерпретатор команд промежуточного языка и использовать его в качестве последнего блока транслятора. Преимущество интерпретаторов проявляется в отладочных и диалоговых трансляторах, обеспечивающих работу пользователя в диалоговом режиме, вплоть до внесений изменений в программу без ее повторной полной перетрансляции.

Интерпретаторы используются также и при эмуляции программ – исполнении на технологической машине программ, составленных для другой (объектной) машины. Данный вариант, в частности, используется при отладке на универсальной ЭВМ программ, которые будут выполняться на специализированной ЭВМ.

Транслятор, использующий в качестве входного языка язык, близкий к машинному (автокод или ассемблер), традиционно называют *ассемблером*. Транслятор для языка высокого уровня называют *компилятором*.

В построении компилятора за последние годы достигнуты значительные успехи. Первые компиляторы использовали так называемые *прямые методы трансляции* – это преимущественно эвристические методы, в которых на основе общей идеи для каждой конструкции языка разрабатывался свой алгоритм перевода в машинный эквивалент. Эти методы были медленные и не носили структурного характера.

В основе методики проектирования современных компиляторов лежит *композиционный синтаксически-управляемый метод обработки языков*. Композиционный в том смысле, что процесс перевода исходной программы в объектную реализуется композицией функционально независимых отображений с явно выделенными входными и выходными структурами данных. Отображения эти строятся из рассмотрения исходной программы, как композиции основных аспектов (уровней) описания входного языка: лексики, синтаксиса, семантики и прагматики, и выявления

этих аспектов из исходной программы в ходе ее компиляции. Рассмотрим эти аспекты с целью получения упрощенной модели компилятора.

Основой любого естественного или искусственного языка является *алфавит* – набор допустимых в языке элементарных знаков (букв, цифр и служебных знаков). Знаки могут объединяться в *слова* – элементарные конструкции языка, рассматриваемые в тексте (программе) как неделимые символы, имеющие определенный смысл.

Словом может быть и одиночный символ. Например, в языке Паскаль словами являются идентификаторы, ключевые слова, константы, и разделители, в частности знаки арифметических и логических операций, скобки, запятые и другие символы. Словарный состав языка вместе с описанием способов их представления составляют *лексику* языка.

Слова в языке объединяются в более сложные конструкции – предложения. В языках программирования простейшим предложением является оператор. Предложения строятся из слов и более простых предложений по правилам синтаксиса. *Синтаксис* языка представляет собой описание правильных предложений. Описание смысла предложений, т.е. значений слов и их внутренних связей, составляет *семантику* языка. В дополнение отметим, что конкретная программа несет в себе некоторое воздействие на транслятор – *прагматизм*. В совокупности синтаксис, семантика и прагматизм языка образуют *семиотику* языка.

Перевод программы с одного языка на другой, в общем случае состоит в изменении алфавита, лексики и синтаксиса языка программы с сохранением ее семантики. Процесс трансляции исходной программы в объектную обычно разбивается на несколько независимых подпроцессов (фаз трансляции), которые реализуются соответствующими блоками транслятора. Удобно считать основными фазами трансляции лексический анализ, синтаксический анализ, семантический анализ и

синтез объектной программы. Тем не менее, во многих реальных компиляторах эти фазы разбиваются на несколько подфаз, могут также быть и другие фазы (например, оптимизация объектного кода). На рис. 1.1 показана упрощенная функциональная модель транслятора.

В соответствии с этой моделью входная программа, прежде всего, подвергается лексической обработке. Цель лексического анализа – перевод исходной программы на внутренний язык компилятора, в котором ключевые слова, идентификаторы, метки и константы приведены к одному формату и заменены условными кодами: числовыми или символьными, которые называются дескрипторами. Каждый дескриптор состоит из двух частей: класса (типа) лексемы и указателя на адрес в памяти, где хранится информация о конкретной лексеме. Обычно эта информация организуется в виде таблиц. Одновременно с переводом исходной программы на внутренний язык на этапе лексического анализа проводится *лексический контроль* - выявление в программе недопустимых слов.

Синтаксический анализатор воспринимает выход лексического анализатора и переводит последовательность образов лексем в форму промежуточной программы. Промежуточная программа является, по существу, представлением синтаксического дерева программы. Последнее отражает структуру исходной программы, т.е. порядок и связи между ее операторами. В ходе построения синтаксического дерева выполняется *синтаксический контроль* – выявление синтаксических ошибок в программе.

Фактическим выходом синтаксического анализа может быть последовательность команд, необходимых для того, чтобы строить промежуточную программу, обращаться к таблицам справочника, выдавать, когда это требуется, диагностическое сообщение.

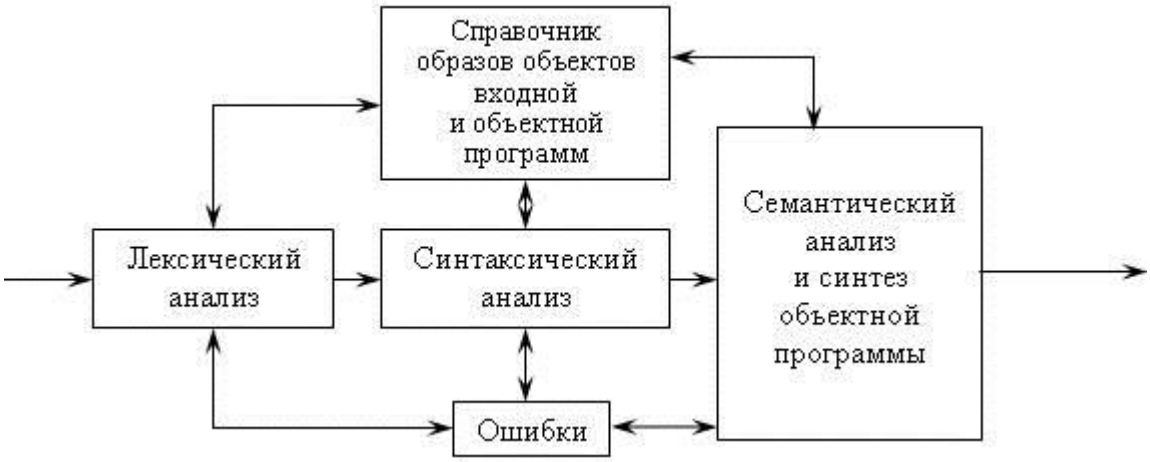

Рисунок 5 - Упрощенная функциональная модель транслятора

Синтез объектной программы начинается, как правило, с распределения и выделения памяти для основных программных объектов. Затем производится исследование каждого предложения исходной программы и генерируется семантически эквивалентные предложения объектного языка. В качестве входной информации здесь используется синтаксическое дерево программы и выходные таблицы лексического анализатора – таблица идентификаторов, таблица констант и другие. Анализ дерева позволяет выявить последовательность генерируемых команд объектной программы, а по таблице идентификаторов определяются типы команд, которые допустимы для значений операндов в генерируемых командах (например, какие требуется породить команды: с фиксированной или плавающей точкой и т.д.).

Непосредственно генерации объектной программы часто предшествует *семантический анализ*, который включает различные виды семантической обработки. Один из видов – проверка семантических соглашений в программе. Примеры таких соглашений: единственность описания каждого идентификатора в программе, определение переменной производится до ее использования и т.д. Семантический анализ может выполняться на более поздних фазах трансляции, например, на фазе оптимизации программы, которая тоже может включаться в транслятор. Цель

оптимизации – сокращение временных ресурсов или ресурсов оперативной памяти, требуемых для выполнения объектной программы.

Таковы основные аспекты процесса трансляции с языков высокого уровня. Подробнее организация различных фаз трансляции и связанные с ними практические способы их математического описания рассматриваются ниже.

Контрольные вопросы

1. Понятие транслятора
2. Понятие компилятора
3. Функциональная модель транслятора

Тема 6. Лексический анализ

Этап лексической обработки текста исходной программы выделяется в отдельный этап работы компилятора, как с методическими целями, так и с целью сокращения общего времени компиляции программы. Последнее достигается за счет того, что исходная программа, представленная на входе компилятора в виде непрерывной последовательности символов, на этапе лексической обработки преобразуется к некоторому стандартному виду, что облегчает дальнейший анализ. При этом используются специализированные алгоритмы преобразования, теория и практика построения которых проработана достаточно глубоко.

Задачи и функционирование блока лексического анализа

Под *лексическим анализом* понимается процесс предварительной обработки исходной программы, в котором основные лексические единицы программы —*лексемы:* ключевые (служебные) слова, идентификаторы, метки, константы приводятся к единому формату и заменяются условными кодами или ссылками на соответствующие таблицы, а комментарии исключаются из текста программы.

Выходами лексического анализа являются поток образов лексем-дескрипторов и таблицы; в последних хранятся значения выделенных в программе лексем.

Дескриптор — это пара вида: (<тип лексемы>, <указатель>),

где <тип лексемы> — это, как правило, числовой код класса лексемы, который означает, что лексема принадлежит одному из конечного множества классов слов, выделенных в языке программирования;

<указатель> — это может быть либо начальный адрес области основной памяти, в которой хранится адрес этой лексемы, либо число, адресующее элемент таблицы, в которой хранится значение этой лексемы.

Количество классов лексем (т.е. различных видов слов) в языках программирования может быть различным. Наиболее распространенными классами являются:

– идентификаторы;
– служебные (ключевые) слова;
– разделители;
– константы.

Могут вводиться и другие классы. Это обусловлено, в первую очередь, той ролью, которую играют различные виды слов при написании исходной программы и, соответственно, при переводе ее в машинную программу. При этом наиболее предпочтительным является разбиение всего множества слов, допускаемого в языке программирования, на такие классы, которые бы не пересекались между собой. В этом случае лексический анализ можно выполнить более эффективно. В общем случае все выделяемые классы являются либо конечными (ключевые слова, разделители и др.) — классы фиксированных для данного языка программирования слов, либо бесконечными или очень большими (идентификаторы, константы, метки) — классы переменных для данного языка программирования слов.

С этих позиций коды образов лексем (дескрипторов) из конечных классов всегда одни и те же в различных программах для данного компилятора. Коды же образов лексем из бесконечных классов различны для разных программ и формируются каждый раз на этапе лексического анализа.

В ходе лексического анализа значения лексем из бесконечных классов помещаются в таблицы соответствующих классов. Конечность таблиц объясняет ограничения, существующие в языках программирования на длины (и соответственно число) используемых в программе идентификаторов и констант. Необходимо отметить, что числовые константы перед помещением их в таблицу могут переводиться из внешнего символьного во внутреннее машинное представление. Содержимое таблиц, в особенности таблицы идентификаторов, в дальнейшем пополняется на этапе семантического анализа исходной программы и используется на этапе генерации объектной программы.

Рассмотрим основные идеи, которые лежат в основе построения лексического анализатора, проблемы, возникающие при его работе, и подходы к их устранению.

Первоначально в тексте исходной программы лексический анализатор выделяет последовательность символов, которая по его предположению должна быть словом в программе, т.е. лексемой. Может выделяться не вся последовательность, а только один символ, который считается началом лексемы. Это наиболее ответственная часть работы лексического анализатора. Более просто она реализуется для тех языков программирования, в которых слова в программе отделяются друг от друга специальными разделителями (например, пробелами), либо запрещается использование служебных слов в качестве переменных, либо классы лексем опознаются по вхождению первого (последнего) символа лексемы.

После этого (или параллельно с этим) проводится идентификация лексемы. Она заключается в сборке лексемы из символов, начиная с

выделенного на предыдущем этапе, и проверки правильности записи лексемы данного класса.

Идентификация лексемы из конечного класса выполняется путем сравнения ее с эталонным значением. Основная проблема здесь — минимизация времени поиска эталона. В общем случае может понадобиться полный перебор слов данного класса, в особенности для случая, когда выделенное для опознания слово содержит ошибку. Уменьшить время поиска можно, используя различные методы ускоренного поиска:

- метод линейного списка;
- метод упорядоченного списка;
- метод расстановки и другие.

Для идентификации лексем из бесконечных (очень больших) классов используются специальные методы сборки лексем с одновременной проверкой правильности написания лексемы. При построении этих алгоритмов широко

применяется формальный математический аппарат — теория регулярных языков, грамматик и конечных распознавателей.

При успешной идентификации значение лексемы из бесконечного класса помещается в таблицу идентификации лексем данного класса. При этом необходимо предварительно проверить: не хранится ли там уже значение данной лексемы, т.е. необходимо проводить просмотр элементов таблицы. Если ее там нет, то значение помещается в таблицу. При этом таблица должна допускать расширение. Опять же для уменьшения времени доступа к элементам таблицы она должна быть специальным образом организована, при этом должны использоваться специальные методы ускоренного поиска элементов.

После проведения успешной идентификации лексемы формируется ее образ — дескриптор, он помещается в выходной поток лексического анализатора. В случае неуспешной идентификации формируются сообщения об ошибках в написании слов программы.

В ходе лексического анализа могут выполняться и другие виды лексического контроля, в частности, проверяется парность скобок и других парных символов.

Выходной поток с лексического анализатора в дальнейшем поступает на вход синтаксического анализатора. Имеется две возможности их связи:

– раздельная связь, при которой выход лексического анализатора формируется полностью и затем передается синтаксическому анализатору;

– нераздельная связь, когда синтаксическому анализатору требуется очередной образ лексемы, он вызывает лексический анализатор, который генерирует требуемый дескриптор и возвращает управление синтаксическому анализатору.

Второй вариант характерен для однопроходных трансляторов. Таким образом, процесс лексического анализа может быть достаточно простым, но в смысле времени компиляции оказывается довольно долгим. больше

половины времени, затрачиваемого компилятором на компиляцию, приходится на этап лексического анализа.

Тема 7. Синтаксический анализ

Синтаксический анализ – это базовый и в настоящее время наиболее формализованный этап трансляции. Существует масса методов синтаксического анализа.

Синтаксический анализатор для автоматного языка можно тривиально построить, основываясь на автоматной грамматике или конечном автомате, описывающем этот язык.

Но, как известно, класс автоматных грамматик очень узок. Любой язык, допускающий вложенность конструкций не является автоматным языком, и даже язык арифметических выражений с вложенными скобками нельзя описать с помощью А-грамматики. Эти грамматики и анализаторы автоматных языков используются только на первой фазе трансляции - лексическом анализе.

Большинство языков программирования можно описать с помощью **контекстно-свободных (КС-) грамматик** или **автоматов с магазинной памятью**(**МП-автоматов**).

Но построить алгоритм анализа по таким грамматикам в общем случае не так просто и эти алгоритмы, как правило, неэффективны.

Один из путей анализа цепочек КС-языка следует из теоремы о разрешимости этих языков. Любой КС-язык можно описать с помощью КС-грамматики в удлиняющей форме, по которой любая цепочка заданного языка длины **n** выводится не более чем за **n** шагов.

Для того чтобы определить принадлежность некоторой цепочки с длиной **n** данному языку необходимо по заданной грамматике вывести все бесповторные цепочки, которые выводятся не более чем за **n** шагов, и сравнить их с исходной цепочкой. Если мы обнаружим совпадение, то исходная цепочка принадлежит языку, в противном случае - нет. Совершенно очевидно, что подобный потенциально осуществимый алгоритм не имеет практического смысла.

Во-первых, необходимо строить вывод миллиардов цепочек, во-вторых, такой алгоритм может ответить только на вопрос принадлежности языку. Локализовать и идентифицировать ошибку, а тем более осуществить трансляцию в процессе синтаксического анализа текста он не способен.

Итак, основная задача синтаксического анализатора сводится к тому, чтобы по заданной КС-грамматике и произвольной цепочке, он мог бы ответить на вопрос о принадлежности цепочки языку.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Э. А. Опалева, В. П. Самойленко. Языки программирования и методы трансляции. СПб.: БХВ-Петербург, 2005. – 480 с.: ил.

2. Пратт Т., Зелковиц М. Языки программирования: разработка и реализация. 4-е изд. - СПб.: Питер, 2002. - 688 с.: ил.

3. Е. Н. Ишакова Теория формальных языков, грамматик и автоматов. Методические указания к лабораторному практикуму. – Оренбург: ГОУ ОГУ, 2005. – 54 с.

4. А.Н. Мелихов, В. И. Кодачигов. Теория алгоритмов и формальных языков: Учебное пособие. Таганрог: ТРТУ, 2006

КОЧКАРОВА Паризат Ахматовна
ЭРКЕНОВА Мадина Умаровна

ТЕОРИЯ ЯЗЫКОВ ПРОГРАММИРОВАНИЯ И МЕТОДЫ ТРАНСЛЯЦИИ

Учебно-методическое пособие для обучающихся 4 курсов
по направлению подготовки 09.03.04 Программная инженерия

Корректор Чагова О.Х.
Редактор Чагова О.Х.

Сдано в набор 26.08.2024 г.
Формат 60х84/16
Бумага офсетная.
Печать офсетная.
Усл. печ. л. 2,09
Заказ № 4955
Тираж 100 экз.

Оригинал-макет подготовлен
в Библиотечно-издательском центре СКГА
369000, г. Черкесск, ул. Ставропольская, 36