Оглавление

Аннотация....3

1 Структура компилятора....4

1.1 Основные понятия и определения....4

1.2 Этапы процесса компиляции....5

1.3 Ранние методы разбора выражений. Метод Рутисхаузера....7

Контрольные вопросы....9

2 Основные положения теории формальных грамматик....10

2.1 Формальная грамматика и формальный язык....10

2.2 Понятие грамматического разбора....12

2.2.1 Левосторонний восходящий грамматический разбор («слева-направо»)....14

2.2.2 Левосторонний нисходящий грамматический разбор («сверху-вниз»)....15

2.3 Расширенная классификация грамматик Хомского....17

Контрольные вопросы....19

3 Распознавание регулярных грамматик....20

3.1 Конечный автомат и его программная реализация....20

3.2 Построение лексических анализаторов....22

3.3 Построение синтаксических анализаторов....24

Контрольные вопросы....26

4 Распознавание КС-грамматик....27

4.1 Автомат с магазинной памятью....27

4.2 Синтаксические анализаторы LL(k)-грамматик. Метод рекурсивного спуска....30

4.3 Синтаксические анализаторы LR(k)-грамматик. Грамматики предшествования....37

4.4 Польская запись. Алгоритм Бауэра-Замельзона....39

Контрольные вопросы....42

5 Распределение памяти под программы и данные....43

Контрольные вопросы....44

6 Генерация и оптимизация кодов....45

Контрольные вопросы....46

Литература....47

МГТУ им. Н.Э. Баумана

Факультет «Информатика и Системы Управления»

Кафедра ИУ-6 «Компьютерные системы и сети»

ИВАНОВА ГАЛИНА СЕРГЕЕВНА,

НИЧУШКИНА ТАТЬЯНА НИКОЛАЕВНА

Основы конструирования компиляторов

Учебное пособие

МОСКВА

2010 год МГТУ им. Баумана

Аннотация

Учебное пособие содержит сведения из математической логики и теории формальных языков, составляющие основу для построения компилирующих программ. Оно включает:

- математическое определение формального языка и формальной грамматики;
- описание классификации формальных грамматик Хомского;
- определение математических моделей конечного автомата и автомата с магазинной памятью;
- описание и анализ применимости методов грамматического разбора;
- примеры практического применения указанного математического аппарата для выполнения лексического и синтаксического анализа формальных языков регулярного и контекстно-свободного типов.

Пособие предназначено для студентов 2 курса кафедры Компьютерные системы и сети, изучающих дисциплину Системное программное обеспечение.

1 Структура компилятора

1.1 Основные понятия и определения

Транслятор – программа, которая переводит программу, написанную на одном языке, в эквивалентную ей программу, написанную на другом языке.

Компилятор – транслятор с языка высокого уровня на машинный язык или язык ассемблера.

Ассемблер – транслятор с языка Ассемблера на машинный язык.

Интерпретатор – программа, которая принимает исходную программу и выполняет ее, не создавая программы на другом языке.

Макропроцессор (***препроцессор*** – для компиляторов) – программа, которая принимает исходную программу, как текст и выполняет в нем замены определенных символов на подстроки. Макропроцессор обрабатывает программу до трансляции.

Любой язык обязательно подчиняется определенным правилам, которые определяют его синтаксис и семантику. ***Синтаксис*** – это совокупность правил, определяющих допустимые конструкции языка, т. е. его *форму*. ***Семантика*** – это совокупность правил, определяющих логическое соответствие между элементами и значением синтаксически корректных предложений, т. е. *содержание* языка.

1.2 Этапы процесса компиляции

Процесс компиляции предполагает распознавание конструкций исходного языка (анализ) и сопоставление каждой правильной конструкции семантически эквивалентной конструкций другого языка (синтез). Он включает несколько этапов:

- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- распределение памяти;
- генерация и оптимизация объектного кода.

1. ***Лексический анализ*** – процесс преобразования исходного текста в строку однородных символов. Каждый символ результирующей строки – ***токен*** соответствует слову языка – ***лексеме*** и характеризуется набором атрибутов, таких как тип, адрес и т. п., поэтому строку токенов часто представляют таблицей, строка которой соответствует одному токену.

Лексема обозначает простое понятие языка. Всего существует 2 типа лексем:

а) лексемы, соответствующие символам алфавита языка, такие как «Служебные слова» и «Служебные символы»;

б) лексемы, соответствующие базовым понятиям языка, такие как «Идентификатор» и «Литерал».

Пример. При лексическом разборе предложения: **if Sum>5 then pr:= true;** будет получена таблица ***токенов*** (см. таблицу 1) и, возможно, расширены таблицы переменных (см. таблицу 2) и литералов (см. таблицу 3):

Таблица 1 – Пример таблицы токенов

Лексема	**Тип лексемы**	**Значение**	**Ссылка**
if	Служебное слово	Код «if»	-
Sum	Идентификатор	-	Адрес в таблице идентификаторов
>	Служебный символ	Код «>»	-
5	Литерал	-	Адрес в таблице литералов
then	Служебное слово	Код «then»	-
pr	Идентификатор	-	Адрес в таблице идентификаторов
:=	Служебный символ	Код «:=»	-
true	Литерал	-	Адрес в таблице литералов
;	Служебный символ	Код «;»	-

Таблица 2 –Таблица идентификаторов переменных

Имя	**Тип**	**Адрес**
Sum	Integer	∅ (пока не распределена)
pr	Boolean	∅ (пока не распределена)

Таблица 3 – Таблица констант

Имя	**Тип**	**Значение**
«5»	Byte	00000101
«true»	Boolean	00000001

2. ***Синтаксический анализ*** – процесс распознавания конструкций языка в строке токенов. Главным результатом является информация об ошибках в выражениях, операторах и описаниях программы.

Пример. На этом этапе для предыдущего примера должны быть распознаны конструкции: <Логическое выражение>, <Оператор присваивания>, <Оператор *if*>.

3. ***Семантический анализ*** – процесс распознавания/проверки смысла конструкции. По результатам распознавания строится последовательность, приближенная к последовательности операторов будущей программы и выполняются предусмотренные проверки правильности программы.

Пример. На этом этапе может быть проверена инициализация переменной *Sum*.

4. ***Распределение памяти*** – процесс назначения адресов для именованных констант и переменных программы.

5. ***Генерация и оптимизация объектного кода*** – процесс формирования программы на выходном языке, которая семантически эквивалентна исходной программе. На этом этапе также обычно выполняется оптимизация генерируемого кода.

Лексический и синтаксический анализ предполагают выполнение грамматического разбора. При их построении используют специальный математический аппарат – формальные грамматики.

1.3 Ранние методы разбора выражений. Метод Рутисхаузера

Первыми компилирующими программами были программы, обеспечивающие перевод формульной записи выражений в машинный язык. В основе такого перевода лежит представление выражения в виде последовательности *троек*, где каждая тройка включает два адреса операндов и операцию:

$\boldsymbol{E_L} \circledcirc \boldsymbol{E_R}$, где E_L, E_R – левый и правый операнды, © – операция.

Основной проблемой при этом является необходимость учета приоритетов операций. Например, для выражения:

$d = a+b*c$ должны быть построены тройки: $T_1 = b*c$, $d = a+T_1$

Исторически первым для решения этой задачи был предложен метод Рутисхаузера.

Метод Рутисхаузера требует, чтобы выражение было записано в полной скобочной записи, когда порядок выполнения операций указывается скобками. Так, выражение $d = a+b*c$ должно быть записано в виде $d = a+(b*c)$, в противном случае сначала будет выполняться операция сложения. Метод заключается в следующем.

1. Каждому символу строки S_i ставится в соответствие индекс N_i по алгоритму:

```
N[0]:=0
J:=1
Цикл-пока  S[J]≠'_'
    Если  S[J] = '(' или S[J] = <операнд>
        то      N[J]:=N[J-1] +1
        иначе  N[J]:=N[J-1] -1
    Все-если
    J:=J+1
Все-цикл
N[J]:=0
```

2. Определяется наибольшее значение индекса в структуре вида $\boldsymbol{k(k-1)k}$ или при наличии скобок $\boldsymbol{(k-1)k(k-1)k(k-1)}$ и строится соответствующая тройка.

3. Обработанные символы вместе со скобками удаляются, и на их место ставится значение $\boldsymbol{N=k-1}$.

4. Операции 2, 3 повторяются до завершения выражения.

Пример. (((a+b)*c)+d)/k

а) **S: (((a + b) * c) + d) / k**

N: 0 1 2 <u>3 4 3 4 3</u> 2 3 2 1 2 1 0 1 0   => T_1 = a+b

b) **S:** **((T_1 * c) + d) / k**

N: **0 1 2 3 2 3 2 1 2 1 0 1 0** => $T_2 = T_1 * c$

c) **S:** **(T_2 + d) / k**

N: **0 1 2 1 2 1 0 1 0** => $T_3 = T_2 + d$

d) **S:** **T_3 / k**

N: **0 1 0 1 0** => $T_4 = T_3 / k$

Недостаток метода – требование полной скобочной структуры. Как правило, для этого используется специальная программа, которая вставляет скобки.

Решение прочих проблем компиляции было не таким простым. Первые компиляторы являлись крайне сложными программами, написание которых требовало неординарных способностей, и содержали большое количество ошибок. Так первый компилятор языка Fortran потребовал 18 человеко-лет работы. С тех пор разработаны систематические технологии решения многих задач компиляции, предложен соответствующий инструментарий, в результате чего небольшой компилятор может стать темой курсового проекта.

Контрольные вопросы

1. Назовите основные типы программ, включающих элементы трансляторов. Поясните их различия.

Ответ.

2. Назовите основные этапы процесса компиляции. Уточните задачи каждого этапа.

Ответ

3. В чем заключается метод Рутисхаузера?

Ответ

2 Основные положения теории формальных грамматик

2.1 Формальная грамматика и формальный язык

Предложения любого языка строятся из символов алфавита.

Алфавит – непустое конечное множество символов, используемых для записи предложений языка. Например, множество арабских цифр, знаки «+» и «-»: A = {0,1,2,3,4,5,6,7,8,9,+,-} – алфавит для записи целых десятичных чисел, например: «-365», «78», «+45».

Строка – любая последовательность символов алфавита, расположенных один за другим. Строки в теории формальных языков обозначаются строчными греческими буквами: α, β, γ … . Строка, содержащая 0 символов, называется пустой и обозначается символами e, ε или λ.

Множество всех составленных из символов алфавита A строк, в которое входит пустая строка, обозначают A*. Множество всех составленных из символов алфавита A строк, в которое не входит пустая строка, обозначают A+. Откуда: $A^* = A^+ \cup \{e\}$.

Формальным языком L в алфавите A называют произвольное подмножество множества A*. Таким образом, язык определяется как множество допустимых предложений.

Задать язык L в алфавите A значит либо перечислить все включаемые в него предложения, либо указать правила их образования. Перечисление бесконечно большого количества предложений невозможно. Определение правил образования предложений осуществляют с использованием абстракций формальных грамматик.

Формальная грамматика – это математическая система, определяющая язык посредством порождающих правил – правил продукции. Она определяется как четверка:

$G = (V_T, V_N, P, S)$,

где V_T – алфавит языка или множество терминальных (незаменяемых) символов;

V_N – множество нетерминальных (заменяемых) символов – вспомогательный алфавит, символы которого обозначают допустимые понятия языка, $V_T \cap V_N = \varnothing$;

$V = V_T \cup V_N$ – словарь грамматики;

P – множество порождающих правил – каждое правило состоит из пары строк (α, β), где $\alpha \in V^+$ – левая часть правила, $\beta \in V^*$ – правая часть правила: $\alpha \rightarrow \beta$, где строка α должна содержать хотя бы один нетерминал;

$S \in V_N$ – начальный символ – аксиома грамматики.

Оглавление

Для описания синтаксиса языков с бесконечным количеством различных предложений используют рекурсию. При этом если нетерминал в порождающем правиле расположен справа, то рекурсию называют правосторонней, если слева, то – левосторонней, а, если между двумя подстроками, то – вложенной.

Пример. Определим грамматику записи десятичных чисел G0:

V_T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, +, -};

V_N = {<целое>, <целое без знака>, <цифра>, <знак>};

P = {<целое> → <знак><целое без знака>, <целое> → <целое без знака>,

<целое без знака> → <цифра><целое без знака>, //правосторонняя рекурсия

<целое без знака> → <цифра>,

<цифра> → 0, <цифра> → 1, <цифра> → 2, <цифра> → 3, <цифра> → 4,

<цифра> → 5, <цифра> → 6, <цифра> → 7, <цифра> → 8, <цифра> → 9,

<знак> → +, <знак> → - };

S = <целое>.

Для записи правил продукции обычно используют более компактные и наглядные формы (модели): форму Бэкуса-Наура или синтаксические диаграммы (см. далее).

Форма Бэкуса-Наура (БНФ). БНФ связывает терминальные и нетерминальные символы, используя две операции: «::=» – «можно заменить на»; «|» – «или». Основное достоинство – группировка правил, определяющих каждый нетерминал. Нетерминалы при этом записываются в угловых скобках. Например, правила продукции грамматики, рассмотренной выше можно записать следующим образом:

<целое> ::= <знак><целое без знака>|<целое без знака>,

<целое без знака> ::= <цифра><целое без знака>|<цифра>(правосторонняя рек.),

<цифра > ::= 0|1|2|3|4|5|6|7|8|9,

<знак> ::= +| - .

Формальная грамматика, таким образом, определяет правила определения допустимых конструкций языка, т. е. его синтаксис.

2.2 Понятие грамматического разбора

Распознавание принадлежности строки конкретному языку осуществляется его выводом из аксиомы. Вывод представляет собой последовательность подстановок, при выполнении которых левая часть правила заменяется правой.

Исходная строка, строки, получаемые в процессе вывода и аксиома, называются ***сентенциальными формами***. Сентенциальная форма, содержащая только терминальные символы, называется ***допустимой*** и представляет собой предложение языка.

Для обозначения подстановки используют символ «⇒».

Пример. Вывод строки «-45»:

<целое> $\Rightarrow_1$

$\Rightarrow_1$ **<знак><целое без знака>** $\Rightarrow_2$

$\Rightarrow_2$ **<знак><цифра><целое без знака>** $\Rightarrow_3$

$\Rightarrow_3$ **<знак><цифра><цифра>** $\Rightarrow_4$

$\Rightarrow_4$ **- <цифра><цифра>** $\Rightarrow_5$

$\Rightarrow_5$ **- 4<цифра>** $\Rightarrow_6$

$\Rightarrow_6$ **- 45**

Вывод можно представить графически, в виде синтаксического дерева, корнем которого является аксиома, а листья – образуют предложение языка (см. рисунок 1).

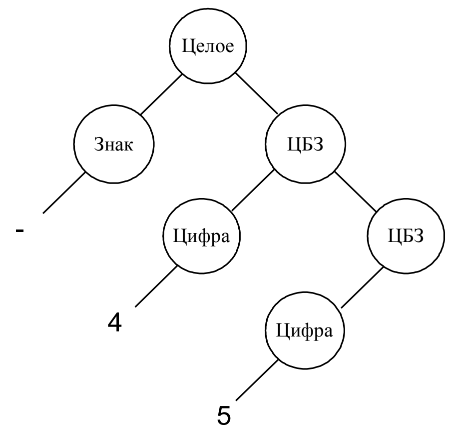


Рисунок 1 – Синтаксическое дерево

Неоднозначные грамматики. Существуют грамматики, в которых для одной строки можно построить несколько деревьев. Такие грамматики называются *неоднозначными*. Разбор неоднозначных грамматик затруднен, поэтому их, если возможно, преобразуют в однозначные или ограничивают правилами.

***Пример* 1.** Строка x+x+x, порождаемая грамматикой с правилами $S \to S + S$ и $S \to x$, имеет два дерева разбора (см. рисунок 2, *а*–*б*). Описывающая тот же язык однозначная грамматика использует правила: $S \to S + x$ и $S \to x$ (см. рисунок 2, *в*).

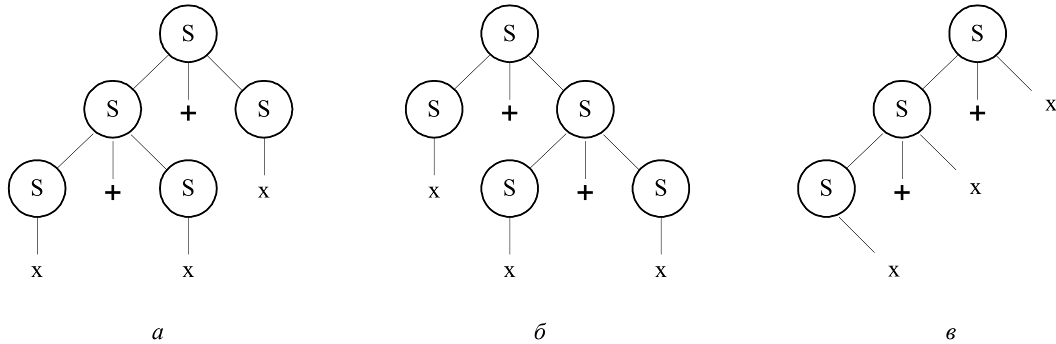


Рисунок 2 – Деревья разбора неоднозначной грамматики (*а*, *б*) и единственное дерево однозначной (*в*)

***Пример* 2.** Конструкция **if** <лог. выр.> **then** <оператор> [**else** <оператор>] при вложении неполного варианта получается неоднозначной. Поскольку ее преобразование невозможно, разбор ограничен правилом вложенности.

Грамматический разбор – процедура построения синтаксического дерева для конкретного предложения языка. Построение такого дерева позволяет однозначно доказать, что анализируемая строка языка является *допустимой*, т.е. принадлежит конкретному языку.

Грамматический разбор может осуществляться в разной последовательности, причем дерево можно строить как «сверху» – от аксиомы, так и «снизу» – от предложения. Соответственно существуют нисходящий и восходящий методы разбора. При этом предложения языка можно рассматривать слева направо и наоборот. Соответственно различают левосторонний и правосторонний разбор. Левосторонний разбор используют чаще, т. к. мы читаем слева направо.

2.2.1 Левосторонний восходящий грамматический разбор («слева-направо»)

Метод разбора «слава-направо» применим, если *грамматика не содержит правил с правосторонней рекурсией*.

При осуществлении грамматического разбора сентенциальные формы просматриваются слева направо и последовательно «свертываются» посредством замены подстроки, совпадающей с правой частью правила («основы») на левую часть того же правила.

Правила подстановки должны проверяться, начиная с самого сложного, иначе сложные правила никогда не будут применены, и разбор не удастся.

В общем случае при разборе возможны возвраты, поскольку может быть выбрано неподходящее правило подстановки.

Пример. Разобрать строку «-45»

Правила грамматики:

<целое> ::= <знак><целое без знака>|<целое без знака>,

<целое без знака> ::= <целое без знака><цифра>|<цифра>,

<цифра > ::= 0|1|2|3|4|5|6|7|8|9,

<знак> ::= +| - .

Последовательность получения сентенциальных форм данным методом:

<знак> 45

<знак> <цифра>5

<знак> <цбз>5

<целое> 5 – **Тупик! Возвращаемся и ищем другой вариант подстановки!**

<знак><целое>5 – **Тупик! Возвращаемся и ищем другой вариант подстановки!**

<знак> <цбз><цифра>

<целое> <цифра> – **Тупик! Возвращаемся и ищем другой вариант подстановки!**

<знак> <целое> <цифра> – **Тупик! Возвращаемся и ищем другой вариант!**

<знак> <цбз>

<целое>

Необходимость предусмотреть возможность возвратов требует хранить всю информацию о каждом шаге разбора, что требует много памяти и существенно усложняет процесс написания программы разбора.

2.2.2 Левосторонний нисходящий грамматический разбор («сверху-вниз»)

Метод разбора «сверху-вниз» применим, если *грамматика не содержит правил с левосторонней рекурсией*.

Метод предполагает последовательное выдвижение гипотез, начиная с аксиомы, их доказательство посредством подбора правил, которым удовлетворяет разбираемый фрагмент, и выдвижения новых гипотез относительно не разобранной части предложения.

Правила подстановки также должны проверяться, начиная с самого сложного, иначе цель разбора не будет достигнута.

При наличии правил с левосторонней рекурсией процедура разбора становится бесконечной.

В общем случае при разборе возможны возвраты, поскольку может быть выбрано неподходящее правило подстановки.

Пример. Разобрать строку «-45»

Правила грамматики:

а) **<целое> ::= <знак><целое без знака>|<целое без знака>**

б) **<целое без знака> ::= <цифра><целое без знака>|<цифра>,**

в) **<цифра > ::= 0|1|2|3|4|5|6|7|8|9,**

г) **<знак> ::= +| - .**

Последовательность разбора:

	Распознано	Текущий символ	Цель или подцель	Правило	Истина?
1		-	Целое	а1	**да**
2		-	Знак	г1	нет
3				г2	да
4	-	4	Цбз	б1	**да**
5		4	Цифра	в1-в4	нет
6				в5	да
7	- 4	5	Цбз	б1	**нет**
8		5	Цифра	в1-в5	нет
9				в6	да
10	-45	_	Цбз	б1	**нет**
11		_	Цифра	в1-в10	нет
12		_	Цбз	б2	**нет**
13		_	Цифра	в1-в10	нет
14	- 4	5	Цбз	б2	**да**

Оглавление

15		5	Цифра	в1-в5	нет
16				в6	да

Недостаток метода, как и в предыдущем случае, заключается в том, что в программе разбора следует хранить всю информацию о каждом его шаге.

Выбор метода разбора для грамматики определяется видом правил продукции языка.

2.3 Расширенная классификация грамматик Хомского

В зависимости от вида правил грамматики различают:

тип 0 – грамматики фразовой структуры или грамматики «без ограничений»:

$\alpha \to \beta$, где $\alpha \in V^{+}$, $\beta \in V^{*}$– в таких грамматиках допустимо наличие любых правил вывода, что свойственно грамматикам естественных языков;

тип 1 – контекстно-зависимые (неукорачивающие) грамматики:

$\alpha \to \beta$, где $\alpha \in V^{+}$, $\beta \in V^{*}$, $| \alpha | \le | \beta |$ – в этих грамматиках для правил вида $\alpha \boldsymbol{X} \beta \to \alpha \boldsymbol{x} \beta$ возможность подстановки строки x вместо символа X определяется присутствием подстрок α и β, т. е. *контекста*, что также свойственно грамматикам естественных языков;

(расширение допускает не более одного правила вида $A \to \boldsymbol{e}$, где $A \in V_N$);

тип 2 – контекстно-свободные грамматики:

$A \to \beta$, где $A \in V_N$, $\beta \in V^{*}$ – поскольку в левой части правила стоит нетерминал, подстановки не зависят от контекста;

тип 3 – регулярная грамматика:

$A \to \alpha$, $A \to \alpha B$, где $A, B \in V_N$, $\alpha \in V_T$;

(расширение допускает не более одного правила вида $S \to \boldsymbol{e}$, но в этом случае аксиома не должна появляться в правых частях правил).

Классификация построена по правилам иерархии, т. е. грамматики типа 3 являются частным случаем грамматик типа 2 и т. п. (см. рисунок 3).

Грамматики большинства существующих языков программирования относятся к типу 2, что связано с рекурсивно-вложенной структурой большинства правил продукции таких языков.

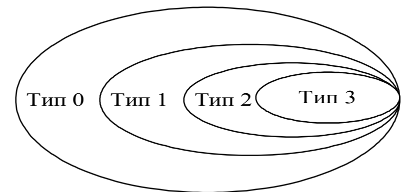


Рисунок 3 – Вложение типов

Один и тот же язык может быть описан грамматиками различных типов, поэтому тип языка определяется максимально возможным для него типом грамматики. Так грамматика языка описания целых десятичных чисел, приведенная в примере выше, относится к типу 2, хотя этот язык можно описать грамматикой типа 3:

**<целое>::= + <цбз>| - <цбз>|0<цбз>| 1<цбз>| 2<цбз>| 3<цбз>|4<цбз>|
5<цбз>| 6<цбз>| 7<цбз>|8<цбз>| 9<цбз>|0|1|2|3|4|5|6|7|8|9**

**<цбз>::= 0<цбз>| 1<цбз>| 2<цбз>| 3<цбз>|4<цбз>|
5<цбз>| 6<цбз>| 7<цбз>|8<цбз>| 9<цбз>|0|1|2|3|4|5|6|7|8|9.**

Оглавление

Следовательно, язык описания целых десятичных чисел относится к типу 3.

Примечание. Существует простой метод проверки регулярности языка, основанный на *лемме о разрастании*, смысл которой заключается в следующем: в строке регулярного языка всегда можно найти непустую подстроку, повторение которой произвольное количество раз порождает новые строки того же языка.

Пример. Язык описания целых чисел. Из строки «+23» можно построить строки: 22222, 23232323 и т. д. того же языка.

Еще одно свойство языка позволяет сразу определить, что язык не относится к классу регулярных – это *самовложение*, т. е. наличие правил вида $A \Rightarrow^{+} \alpha_1 A \alpha_2$, где $\alpha_1, \alpha_2 \in V^{+}$.

Пример. Язык скобок, описываемый правилами: $S \rightarrow (S)$, $S \rightarrow SS$, $S \rightarrow e$. Грамматика этого языка является грамматикой с самовложением, принадлежит классу КС и не приводима к регулярной.

Лемма о разрастании КС языков формулируется следующим образом: в строке, принадлежащей КС-языку, всегда можно найти две подстроки с ненулевой суммарной длиной, одновременное повторение которых произвольное количество раз порождает новую строку того же языка.

Пример. Язык скобок. Из строки «()» можно построить строку: ((())) и т. д. того же языка.

Проверка соответствия строк языку осуществляется специальной программой – ***распознавателем***. Распознаватели могут использоваться для определения языка также как и грамматики. Чем шире класс распознаваемых грамматик, тем сложнее класс соответствующих распознавателей. Доказано, что:

- грамматики типа 3 распознаются *конечными автоматами*;
- грамматики типа 2 распознаются *автоматами с магазинной памятью*;
- грамматики типа 1 распознаются *линейными ограниченными автоматами*;
- грамматики типа 0 распознаются *машинами Тьюринга*.

Контрольные вопросы

1. Определите, что такое алфавит языка.
   Ответ.
2. Дайте определение формального языка. Почему формальные языки обычно не определяют перечислением допустимых предложений?
   Ответ.
3. Дайте определение формальной грамматики.
   Ответ.
4. Что такое «Форма Бэкуса-Наура»? Для чего она используется?
   Ответ.
5. Определите цель грамматического разбора предложений языка.
   Ответ.
6. В чем заключается левосторонний восходящий грамматический разбор? Назовите, в каких случаях он не применим и определите его основной недостаток.
   Ответ.
7. В чем заключается правосторонний нисходящий грамматический разбор? Назовите, в каких случаях он не применим и определите его основной недостаток.
   Ответ.
8. Назовите основные типы грамматик по Хомскому. Какие грамматики используют в языках программирования?
   Ответ.

3 Распознавание регулярных грамматик

3.1 Конечный автомат и его программная реализация

Распознаватели регулярных грамматик строятся на конечных автоматах. ***Конечный автомат*** – это математическая модель, свойства и поведение которой полностью определяются пятеркой: $M = (Q, \Sigma, \delta, q_0, F)$,

где Q – конечное множество состояний;

Σ – конечное множество входных символов;

$\delta(q_i, c_k)$ – функция переходов (q_i – текущее состояние, c_k – очередной символ);

q_0 – начальное состояние;

$F = \{q_j\}$ – подмножество допускающих состояний.

Конечные автоматы – одна из базовых моделей, используемых при проектировании программного и аппаратного обеспечения средств вычислительной техники.

Пример. Автомат «Чет-нечет» описывается следующим образом:

Q = {Чет, Нечет};

Σ = {0, 1};

δ(Чет, 0) = Чет, δ(Нечет, 0) = Нечет, δ(Чет, 1) = Нечет, δ(Нечет, 1) = Чет;

q_0 = Чет;

F = {Чет}

Строка «10110» приведет такой автомат в состояние Нечет, т. е. будет отвергнута, а строка «110011» – в состояние Чет, т. е. будет принята.

Для представления функции переходов конечного автомата помимо аналитической формы могут использоваться: таблица (см. таблицу 4), граф переходов состояний (см. рисунок 4) и синтаксическая диаграмма (см. рисунок 5).

Таблица 3 – Функция переходов

q / σ	0	1
Чет	Чет	Нечет
Нечет	Нечет	Чет

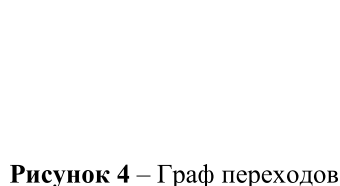


Рисунок 4 – Граф переходов

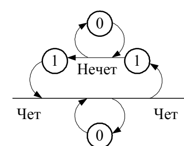


Рисунок 5 – Синтаксическая диаграмма

Разработчик автомата обычно использует представление автомата графом или синтаксической диаграммой. При реализации в виде программы автомат задают таблицей переходов.

Для идентификации завершения строки обычно применяют специальный символ или множество символов, которые включают во входной алфавит и учитывают в таблице. Кроме этого, в таблице также предусматривают реакцию автомата на символы, не включенные в алфавит и не являющиеся завершающими. Появление таких символов в строке однозначно означает, что предложение языка содержит ошибку (см. таблицу 5).

Таблица 4 – Дополненная таблица конечного автомата

q *c*	0	1	**Символы завершения**	**Другие символы**
Чет	Чет	Не-чет	**Конец**	**Ошибка**
Нечет	Не-чет	Чет	**Ошибка**	**Ошибка**

Пусть *S* – строка на входе автомата;

Ind – номер очередного символа;

q – текущее состояние автомата;

Table – таблица, учитывающая символы завершения и другие символы.

Тогда алгоритм работы автомата можно представить следующим образом.

***Ind* := 1**

$q := q_0$

Цикл-пока *q* ≠ «Ошибка» и *q* ≠ «Конец»

q = *Table* [*q*, *S*[*Ind*]]

***Ind* := *Ind* +1**

Все-цикл

Если *q* = «Конец»

то «Строка принята»

иначе «Строка отвергнута»

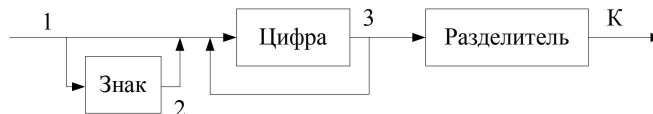
Все-если

3.2 Построение лексических анализаторов

При построении лексических анализаторов конечные автоматы используют для распознавания лексем второго типа (базовых понятий языка). Эти понятия можно описать в рамках самых простых, регулярных грамматик и предложить для них эффективную технику разбора. Отделение сканера от синтаксического анализатора позволяет также сократить время распознавания лексем, так как при этом появляется возможность использования оптимизированных ассемблерных команд, специально предназначенных для сканирования строк.

Алфавит автомата лексического анализатора – все множество однобайтовых (*ANSI*) или двухбайтовых (*Unicode*) символов. При записи правил обычно используются обобщающие нетерминалы вида «Буквы», «Цифры». В процессе распознавания может формироваться описываемый объект, например, литерал или идентификатор.

Пример. Распознаватель целых чисел. Синтаксическая диаграмма синтаксиса языка описывается синтаксической диаграммой (см. рисунок 6).



1 – начало разбора; 2 – распознан знак; 3 – целое; *К* – завершение разбора

Рисунок 6 – Синтаксическая диаграмма

По диаграмме строим таблицу переходов (см. таблицу 5), обозначая состояние ошибки символом «Е». В таблице переходов указываем подпрограммы обработки, которые должны быть выполнены при осуществлении указанного перехода. При выполнении этих подпрограмм формируются указанные значения.

Таблица 5 – Таблица переходов

	Знак	Цифра	Разделитель	Другие
1	2, А1	3, А2	Е, Д1	Е, Д4
2	Е, Д2	3, А2	Е, Д3	Е, Д4
3	К, А3	3, А2	К, А3	Е, Д4

а) Подпрограммы обработки:

А0: Инициализация: Целое := 0; Знак_числа := «+».

А1: Знак_числа := Знак

А2: Целое := Целое*10 + Цифра

А3: <u>Если</u> Знак_числа = «-» <u>то</u> Целое := -Целое <u>Все-если</u>

б) Диагностические сообщения:

Д1: «Строка не является десятичным числом»;

Д2: «Два знака рядом»;

Д3: «В строке отсутствуют цифры»,

Д4: «В строке встречаются недопустимые символы»

Обозначим: *S* – строка на входе автомата; *Ind* – номер очередного символа; *q* – текущее состояние автомата; *Table* – таблица, учитывающая символы завершения и другие символы.

Тогда алгоритм сканера-распознавателя можно представить следующим образом.

***Ind* := 1**

***q* := 1**

Выполнить *A0*

Цикл-пока *q* ≠ «*E*» и *q* ≠ «*K*»

Если *S*[*Ind*] = «+» или *S*[*Ind*] = «-»,

то j := 1

иначе Если *S*[*Ind*] ≥ «0» и *S*[*Ind*] ≤ «9»,

то *j* := 2,

иначе *j* := 3

Все-если

Все-если

Выполнить A_i := *Table* [*q*, *j*]. *A*()

***q* := *Table* [*q*, *j*]**

***Ind* := *Ind* +1**

Все-цикл

Если *q* = «К»

то Выполнить А3

Вывести сообщение «Это число»

иначе Вывести сообщение Дi

Все-если

3.3 Построение синтаксических анализаторов

Синтаксические анализаторы регулярных языков на вход получают строку лексем.

Пример. Синтаксический анализатор списка описания целых скаляров, массивов и функций (упрощенный вариант), например: int xaf, y22[5], zrr[2][4], re[N], fun(), *g;

После лексического анализа входная строка представлена в алфавите:

V – идентификатор; *N* – целочисленная константа; служебные символы: «[ ] ( ) , ; *».

Функцию переходов зададим синтаксической диаграммой (см. рисунок 7).

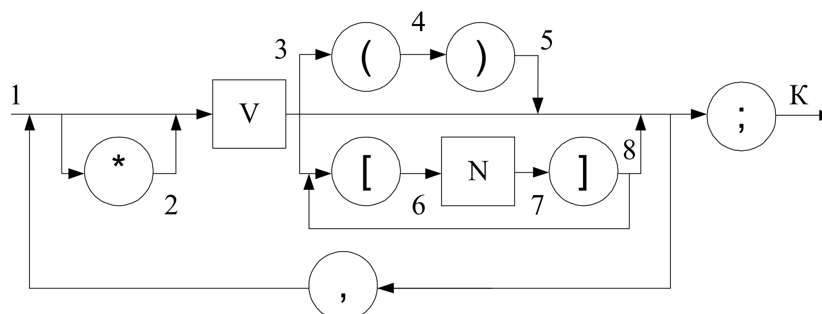


Рисунок 7 – Синтаксическая диаграмма

По диаграмме построим таблицу автомата (см. таблицу 6).

Таблица 6 –Таблица переходов

	V	N	*	(	)	[	]	,	;	Другие
1	**3**	**E**	**2**	**E**	**E**	**E**	**E**	**E**	**E**	**E**
2	**3**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**
3	**E**	**E**	**E**	**4**	**E**	**6**	**E**	**1**	**К**	**E**
4	**E**	**E**	**E**	**E**	**5**	**E**	**E**	**E**	**E**	**E**
5	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**1**	**К**	**E**
6	**E**	**7**	**E**	**E**	**E**	**E**	**E**	**E**	**E**	**E**
7	**E**	**E**	**E**	**E**	**E**	**E**	**8**	**E**	**E**	**E**
8	**E**	**E**	**E**	**E**	**E**	**6**	**E**	**1**	**К**	**E**

Алгоритм распознавателя:

***Ind* := 1**

***q* := 1**

Цикл-пока *q* ≠ «E» и *q* ≠ «К»

  ***q* := *Table* [*q*, *Pos*(*S*[*Ind*], «VN*()[];»)]**

  ***Ind* := *Ind* +1**

Все-цикл

Если *q* = «К»

  то Выполнить А3

Вывести сообщение «Это число»

иначе Вывести сообщение Ді

Все-если

Контрольные вопросы

1. Определите, что конечный автомат. В чем особенность его использования при грамматическом разборе.
   Ответ.
2. Какие способы описания конечных автоматов существуют? Какой способ используется при программной реализации и почему?
   Ответ.
3. Как построить конечный автомат для лексического анализа? В чем заключается его особенность?
   Ответ.
4. Как построить конечный автомат для синтаксического анализа?
   Ответ.

4 Распознавание КС-грамматик

4.1 Автомат с магазинной памятью

Распознавание КС-грамматик выполняется автоматом с магазинной памятью. ***Автомат с магазинной памятью*** определяется семеркой:

$P_M = (Q, \Sigma, \Gamma, \delta, q_0, z_0, F)$,

где Q – конечное множество состояний автомата;

Σ – конечный входной алфавит;

Γ – конечное множество магазинных символов;

$\delta\ (q, c_k, z_j)$ – функция переходов;

$q_0 \in Q$ – начальное состояние автомата;

$z_0 \in \Gamma$ – символ, находящийся в магазине в начальный момент,

$F \subseteq Q$ – множество заключительных (допускающих) состояний.

Пример. Синтаксический анализатор выражений.

Алфавит языка записи выражений: Σ = {<Ид>, +, –, *, /, (,), ◄, ►}. Грамматика описывается синтаксическими диаграммами, представленными на рисунке 8.

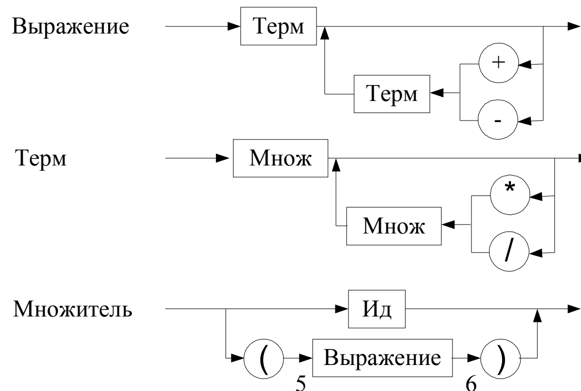


Рисунок 8 – Синтаксические диаграммы

Подставляем диаграммы одна в другую, исключая промежуточные нетерминалы Терм и Множитель. В результате получаем полную синтаксическую диаграмму (см. рисунок 9), которая состоит из двух частей: основной и рекурсивной, что связано с наличием рекурсивного вложения правил продукции.

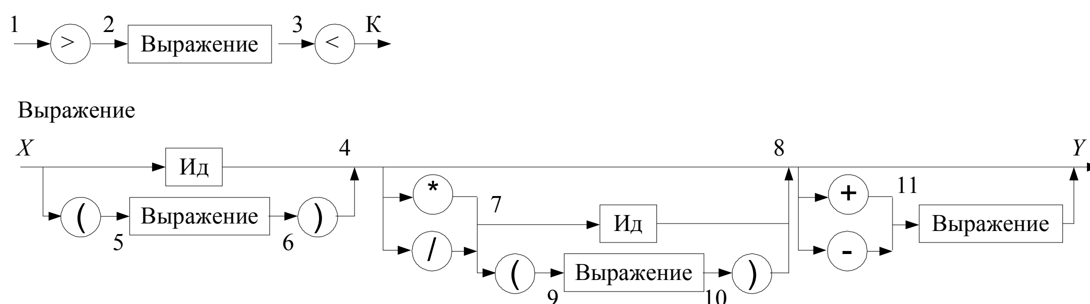


Рисунок 9 – Синтаксическая диаграмма

По диаграмме строим таблицу автомата, которая также будет состоять из двух частей (см. таблицу 7), которые можно объединить в одну.

Таблица 7 – Таблица переходов автомата

	<Ид>	+	-	*	/	(	)	▶	◀
1								2	
2	↓*S*=3					↓ *S* =3			
3									К

	<Ид>	+	-	*	/	(	)	▶	◀
X	4					5			
4		11	11	7	7		↑ *S*		↑ *S*
5	↓ *S* =6					↓ *S* =6			
6							4		
7	8					9			
8		11	11				↑ *S*		↑ *S*
9	↓ *S* =10					↓ *S* =10			
10							8		
11	↓ *S* =Y					↓ *S* =Y			
Y							↑ *S*		↑ *S*

Условные обозначения:

К – конец разбора, *E* – состояние ошибки (все пустые ячейки должны содержать *E*),

↓ *Y* = <состояние> – рекурсивный вызов распознавателя, справа указан номер состояния после возврата из данного вызова;

↑*Y* – возврат из рекурсии, следующее состояние определяется значением *Y*,

При разработке алгоритма используем следующие обозначения:

Mag – стек (магазин) распознающего автомата, элементы, записываемые в стек, – терминальные и нетерминальные символы; ↓ – запись в стек, ↑ – чтение из стека;

q – текущее состояние;

Оглавление

Table (<Текущее состояние>, <Анализируемый символ>) – элемент таблицы (матрицы) переходов, состоящий из следующих полей: *q* – следующее состояние (в том числе «↓» – рекурсивный вызов, «↑» – возврат из рекурсии), *S* – состояние после возврата из рекурсии, если необходим рекурсивный вызов, или ∅;

String – исходная строка;

Ind – номер символа исходной строки.

Алгоритм программы, реализующей автомат, будет выглядеть следующим образом:

_q_:=1

Ind := 1

Mag :=∅

Цикл-пока _q_ ≠ «Е» и _q_ ≠ «К»

  _q_ := _Table_ [_q_, _String_[_Ind_]]._q_

  Если _q_ = ′↓′

    то _Mag_ ↓ _Table_ [_q_, _String_[_Ind_]]._S_

      _q_ := _X_

    иначе Если _q_ = ′↑′

          то _Mag_ ↑_q_

          иначе _Ind_ := _Ind_ +1

        Все-если

  Все-если

  Если _q_ = «К»

    то «Строка принята»

    иначе «Строка отвергнута»

  Все-если

Однако построение такого автомата для сложного языка – задача не простая. Поэтому в литературе предлагаются другие способы. Обычно выделяют подклассы грамматик, обладающих определенными свойствами, основываясь на которых можно строить конкретные распознаватели.

4.2 Синтаксические анализаторы LL(k)-грамматик. Метод рекурсивного спуска

LL(k)-грамматики – это класс КС-грамматик, гарантирующих существование детерминированных *нисходящих* распознавателей (*L* – левосторонний просмотр исходной строки, *L* – левосторонний разбор, *k* – количество символов, просматриваемых для определения очередного правила). В грамматиках этого класса для однозначного выбора правил должны:

1) отсутствовать правила с левосторонней рекурсией;

2) различаться *k* первых символов разных правил.

Реализация нисходящего разбора заключается в следующем. В исходный момент времени в стек записывают аксиому. Затем выбирают правило подстановки аксиомы, сравнивая *k* символов исходной строки с тем же количеством символов правых частей правил. Правую часть правила подставляют в стек вместо левой части. Одинаковые первые символы стека и исходной строки удаляют. Затем вновь выбирают правило подстановки уже для нетерминала, который оказался в начале стека и т. д. Количество просматриваемых символов определяется конкретными правилами грамматики.

Пример. Дана грамматика записи выражений:

1) <Строка> ::= <Выр>◄

2) <Выр> ::= <Терм> <Слож>

3) <Слож> ::= e | + <Терм> <Слож> | - <Терм><Слож>

4) <Терм> ::= <Множ><Умн>

5) <Умн> ::= e | *<Множ><Умн> | / <Множ><Умн>

6) <Множ> ::= <Ид> | (<Выр>)

В данной грамматике в тех случаях, когда есть несколько правил для нетерминала выбор определяется одним терминальным символом, поэтому данная грамматика относится к классу *LL*(1). Следовательно, для распознавания можно использовать нисходящий метод (см. таблицу 8).

Таблица 8 – Результат распознавания

Распознаваемая строка	Стек	Правило
<Ид>*(<Ид>+<Ид>)+<Ид>◀	<Выр> ◀	2
<Ид>*(<Ид>+<Ид>)+<Ид>◀	**<Терм>**<Слож>◀	4
<Ид>*(<Ид>+<Ид>)+<Ид>◀	<**Множ**><Умн><Слож>◀	6а
<Ид>*(<Ид>+<Ид>)+<Ид>◀	<Ид><Умн><Слож>◀	Удалить символ
*(<Ид>+<Ид>)+<Ид>◀	<**Умн**><Слож>◀	5б
*(<Ид>+<Ид>)+<Ид>◀	*<Множ><Умн><Слож>◀	Удалить символ
(<Ид>+<Ид>)+<Ид>◀	<**Множ**><Умн><Слож>◀	6б
(<Ид>+<Ид>)+<Ид>◀	(<Выр>)<Умн><Слож>◀	Удалить символ
<Ид>+<Ид>)+<Ид>◀	**<Выр>**)<Умн><Слож>◀	2
<Ид>+<Ид>)+<Ид>◀	**<Терм>**<Сложе>)<Умн><Слож>	4
<Ид>+<Ид>)+<Ид>◀	<**Множ**><Умн><Слож>)<Умн><Слож>◀	6а
<Ид>+<Ид>)+<Ид>◀	<**Ид**><Умн><Слож>)<Умн><Слож>◀	Удалить символ
+<**Ид**>)+<Ид>◀	<Умн><Слож>)<Умн><Слож>◀	5а (е)
+<**Ид**>)+<Ид>◀	<Слож>)<Умн><Слож>◀	2а
+<**Ид**>)+<Ид>◀	+<Терм><Слож>)<Умн><Слож>◀	Удалить символ
<**Ид**>)+<Ид>◀	<Терм><Слож>)<Умн><Слож>◀	4
<**Ид**>)+<Ид>◀	**<Множ><Умн>**<Слож>)<Умн><Сложе>◀	6а
<**Ид**>)+<Ид>◀	<**Ид**>**<Умн><Слож>**)<Умн><Слож>◀	Удалить символ
)+<Ид>◀	**<Умн><Слож>**)<Умн><Слож>◀	5а (е)
)+<Ид>◀	**<Слож>**)<Умн><Слож>◀	3а (е)
)+<Ид>◀	)<Умнож><Слож>◀	Удалить символ
+<Ид>◀	<**Умн**><Слож>◀	5а (е)
+<Ид>◀	<**Слож**>◀	3б
+<Ид>◀	**+**<Терм><Слож>◀	Удалить символ
<Ид>◀	**<Терм>**<Слож>◀	4
<Ид>	**<Множ>**<Умн><Слож>◀	6а
<Ид>	<Ид><Умн><Слож>◀	Удалить символ
◀	**<Умн**><Слож>◀	5а (е)
◀	**<Слож>**◀	3а (е)
◀	◀	Конец

Оглавление

Метод рекурсивного спуска. Метод рекурсивного спуска основывается на синтаксических диаграммах языка. Согласно этому методу для каждого нетерминала разрабатывают рекурсивную процедуру. Основная программа вызывает процедуру аксиомы, которая вызывает процедуры нетерминалов, упомянутые в правой части аксиомы и т. д. В эти же процедуры встраивают семантическую обработку распознанных конструкций.

Пример. Определим синтаксис языка описания выражений синтаксическими диаграммами (см. рисунок 10).

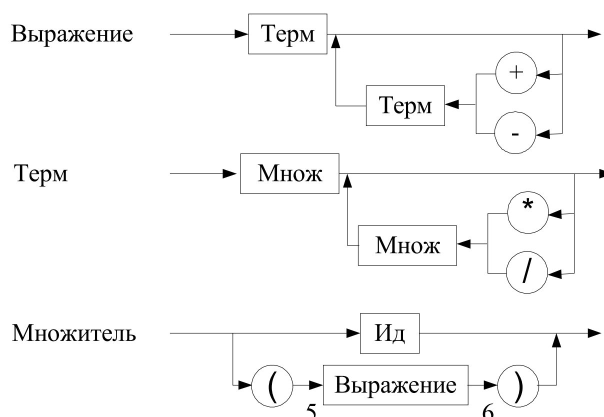


Рисунок 10 – Синтаксические диаграммы грамматики описания выражений

По каждой диаграмме пишем рекурсивную процедуру и добавляем основную программу, вызывающую процедуру аксиомы:

```
Функция Выражение:Boolean:
    R:=Терм()
    Цикл-пока R=true и (NextSymbol =’+’ или NextSymbol =’-’)
                R:=Терм()
    Все-цикл
    Выражение:= R
Все
Функция Терм:boolean:
    Множ()
    Цикл-пока R=true и (NextSymbol =’*’ или NextSymbol =’/’)
                R:=Множ()
    Все-цикл
    Терм:= R
Все
Функция Множ:Boolean:
    Если NextSymbol =’(’
      то  R:=Выражение()
```

Если NextSymbol ≠ ‘)’то Ошибка Все-если

иначе R:= Ид()

Все-если

Все

Основная программа:

R:=Выражение()

Если NextSymbol ≠ ‘◄’то Ошибка ()Все-если

Конец

Ниже приведен текст программы – распознавателя выражений. Распознаватель идентификаторов построен по синтаксической диаграмме на рисунке 11:

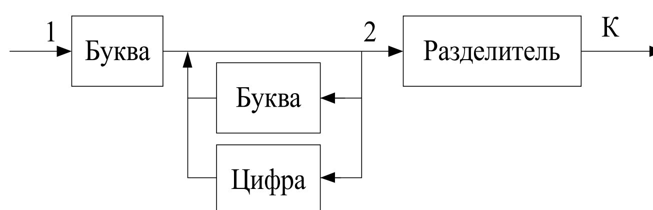


Рисунок 11 – Синтаксическая диаграмма нетерминала <Идентификатор>

```
program Compiler;

{$APPTYPE CONSOLE}

uses
  SysUtils;

Type SetofChar=set of AnsiChar;
Const Bukv:setofChar=['A'..'Z','a'..'z'];
Const Cyfr:setofChar=['0'..'9'];
Const Razd:setofChar=[' ','+','-','*','/',')'];
Const TableId:array[1..2,1..4] of Byte=((2,0,0,0),(2,2,10,0));
Function Culc(Var St:shortstring;Razd:setofChar):boolean;for-
ward;
Var St:shortstring; R:boolean;

Procedure Error(St:shortstring); {вывод сообщений об ошибках}
  Begin      WriteLn('Error *** ', st, ' ***');  End;
```

```
Procedure Probel(Var St:shortstring); {удаление пробелов}
  Begin   While (St<>'') and (St[1]=' ') do Delete(St,1,1);
End;

Function Id(Var St:shortstring;Razd:setofChar):boolean;
{распознаватель идентиф.}
  Var S:shortString;
  Begin
    Probel(St);
    S:='';
    if St[1] in Bukv then
       Begin
         S:=S+St[1]; Delete(St,1,1);
         While (St<>'')and (St[1] in Bukv) or (St[1] in Cyfr) do
            Begin
              S:=S+St[1]; Delete(St,1,1);
            End;
         if (St='') or (St[1] in Razd) then
            Begin
              Id:=true;  WriteLn('Identify=',S);
            End
         else
            Begin
              Id:=false; WriteLn('Wrong symbol *',St[1],'*');
            End;
       End
    else
       Begin
          Id:=false; WriteLn('Identifier waits...', St);
       End;
  End;

Function Mult(Var St:shortstring;Razd:setofChar):boolean;
  Var R:boolean;
  Begin
```

```
    Probel(St);
    if St[1]='(' then
      begin
        Delete(St,1,1); Probel(St);
        R:=Culc(St,Razd);
        Probel(St);
        if R and (St[1]=')') then Delete(St,1,1) else Error(St);
      end
    else R:=Id(St,Razd);
    Mult:=R;
  End;

Function Term(Var St:shortstring;Razd:setofChar):boolean;
  Var S:shortstring; R:boolean;
  Begin
     R:=Mult(St,Razd);
     if R then
       begin
         Probel(St);
         While ((St[1]='*') or (St[1]='/')) and R do
           begin
             Delete(St,1,1);
             R:=Mult(St,Razd);
           end;
       end;
     Term:=R;
  End;

Function Culc(Var St:shortstring;Razd:setofChar):boolean;
  Var S:shortstring; R:boolean;
  Begin
     R:=Term(St,Razd);
     if R then
       begin
         Probel(St);
```

```
          While ((St[1]='+') or (St[1]='-')) and R do
            begin
              Delete(St,1,1);
              R:=Term(St,Razd);
            end;
        end;
      Culc:=R;
   End;

Begin
   Writeln('Input Strings:'); Readln(St);
   R:=true;
   While (St<>'end') and R do
     Begin
       R:=Culc(St,Razd);
       if R and (length(st)=0) then Writeln('Yes') else
Writeln('No');
       Writeln('Input Strings:'); Readln(St);
     End;
     writeln('Input any key');
     readln;
End.
```

4.3 Синтаксические анализаторы LR(k)-грамматик. Грамматики предшествования

LR(k)-грамматики – это класс КС-грамматик, гарантирующих существование детерминированных *восходящих* распознавателей (*L* – левосторонний просмотр, *R* – правосторонний разбор, *k* – количество символов, просматриваемых для однозначного определения следующего правила). В грамматиках этого класса отсутствуют правила с правосторонней рекурсией, и обеспечивается однозначное выделение основы. К этому классу, например, относятся грамматики с операторным предшествованием, простым предшествованием и расширенным предшествованием, обеспечивающие еще более простые алгоритмы распознавания.

При восходящем разборе стек используют для накопления основы. Автомат при этом выполняет две основные операции: свертку и перенос. Свертка выполняется, когда в стеке накоплена вся основа, и заключается в ее замене на левую часть соответствующего правила. Перенос выполняется в процессе накопления основы и заключается в сохранении в стеке очередного распознаваемого символа сентенциальной формы. Основная проблема метода заключается в нахождении способа выделения очередной основы. Проще всего основу выделить для грамматик, получивших название «грамматики предшествования». Рассмотрим эти грамматики.

Если два символа $\alpha, \beta \in V$ расположены рядом в сентенциальной форме, то между ними возможны следующие отношения, названные отношениями предшествования:

- α принадлежит основе, а β – нет, т. е. α – конец основы: $\alpha \cdot> \beta$;
- β принадлежит основе, а α – нет, т. е. β – начало основы: $\alpha <\cdot \beta$;
- α и β принадлежит одной основе, т. е. $\alpha =\cdot \beta$;
- α и β не могут находиться рядом в сентенциальной форме (ошибка).

Грамматикой с предшествованием называется грамматика, в которой существует однозначное отношение предшествования между соседними символами. Это отношение позволяет просто определить очередную основу, т. е. момент выполнения каждой свертки.

Различают:

1) грамматики с простым предшествованием, для которых $\alpha, \beta \in V$;

2) грамматики с операторным предшествованием, для которых $\alpha, \beta \in VT$; т. е. отношение предшествования определено для терминальных символов и не зависит от нетерминальных символов, расположенных между ними;

Оглавление

3) грамматики со слабым предшествованием, для которых отношение предшествования не однозначно – оно требует выполнения специальных проверок.

Пример. Грамматика описания арифметических выражений, представленная ниже, относится к классу грамматик с операторным предшествованием:

<Выражение> ::= <Терм> | < Терм > + <Выражение> | <Терм> - <Выражение>

<Терм> ::= <Множитель> | <Множитель>* <Терм> | <Множитель> / <Терм>

<Множитель> ::= (<Выражение>) | <Идентификатор>

Отношения предшествования терминалов (знаков операций), полученные с учетом приоритетов операций, сведены в таблицу 9.

Таблица 9 – Таблица предшествования

	+	*	(	)	◀
▶	<·	<·	<·	?	Выход
+	·>	<·	<·	·>	·>
*	·>	·>	<·	·>	·>
(	<·	<·	<·	=	?
)	·>	·>	?	·>	·>

Обозначения:
? – ошибка;
< - начало основы;
> **-** конец основы;
= - принадлежат одной основе;
▶- начало выражения;
◀ - конец выражения.

В соответствие с этой таблицей при разборе выражения:

▶d+c*(a+b) ◀

содержимое стека будет выглядеть следующим образом:

Содержимое стека	Анализируемые символы	Отношение	Операция	Тройка	Результат свертки
▶	d+	<·	Перенос		
▶d+	c*	<·	Перенос		
▶d+ c*	(	<·	Перенос		
▶d+ c*(	a+	<·	Перенос		
▶d+ c*(a+	b)	·>	Свертка	R_1:= a + b	<Выражение>
▶d+ c*(	R_1)	=·	Свертка	R_1:= (R_1)	<Множитель>
▶d+ c*	R_1◀	·>	Свертка	R_2:= c* R_1	<Терм>
▶d+	R_2◀	·>	Свертка	R_3:= d+ R_2	<Выражение>
▶	R_3◀	Конец			

4.4 Польская запись. Алгоритм Бауэра-Замельзона

Результат синтаксического анализа – дерево грамматического разбора – представляют в виде таблицы или обратной польской записи.

Польская запись (в честь польского математика Лукасевича, предложившего этот вид записи выражений) представляет собой последовательность команд двух типов:

1) K_I, где I – идентификатор операнда – выбрать число по имени I и заслать его в стек операндов;

2) K_ξ, где ξ – операция – выбрать два верхних числа из стека операндов, произвести над ними операцию ξ и занести результат в стек операндов.

Рассмотрим алгоритм Бауэра-Замельзона, по которому осуществляется представление выражений в виде польской записи.

Алгоритм Бауэра-Замельзона. Грамматика, описывающая правила записи арифметических выражений, относится к классу грамматик операторного предшествования, т. е. порядок следования терминальных символов (знаков операций), однозначно определяет порядок выделения троек, причем нетерминальные символы (имена операндов) на этот порядок не влияют.

Синтаксический распознаватель выражений в процессе разбора должен формировать запись, по которой затем выполняется генерация кода. В качестве такой записи часто используют обратную польскую запись.

Согласно алгоритму Бауэра-Замельзона разбор выражения и формирование польской записи выполняется в два этапа:

1) разбор выражения и построение эквивалентной польской записи;
2) выполнение (или трансляция) польской записи.

При этом используется два стека: стек операций – на первом этапе и стек операндов – на втором этапе.

Построение польской записи выполняется следующим образом: транслятор читает выражение слева направо и вырабатывает последовательность команд по следующему правилу:

а) если символ – операнд, то вырабатывается команда $\mathbf{K_I}$,

б) если символ – операция, то выполняются действия согласно таблице 10:

Таблица 10 – Таблица генератора

η\ξ	+	*	(	)	←
→	I	I	I	?	Вых
+	II	I	I	IV	IV
*	IV	II	I	IV	IV
(	I	I	I	III	?

Обозначения:

? - ошибка;

η - верхний символ в стеке операций;

ξ - текущий символ.

Операции:

I – заслать ξ в стек операций и читать следующий символ;

II – генерировать $\mathbf{K}_\eta$, заслать ξ в стек операций и читать следующий символ;

III – удалить верхний символ из стека операций и читать следующий символ;

IV – генерировать $\mathbf{K}_\eta$ и повторить с тем же входным символом.

На этапе выполнения польская запись читается слева направо и выполняется.

Польская запись может использоваться как промежуточная форма не только для выражений, но и для других операторов. Соответственно при этом для получения записи должен использоваться модифицированный алгоритм Бауэра-Замельзона.

Пример. Построить тройки для выражения **(*a*+*b***c*)/*d*.**

1. Построение польской записи:

Стек операций	Символ	Действие	Команда
▶	(	I	
▶ (	a		$\mathbf{K}_a$
▶ (	+	I	
▶ (+	b		$\mathbf{K}_b$
▶ (+	*	I	
▶ (+ *	c		$\mathbf{K}_c$
▶ (+ *	)	IV	$\mathbf{K}_*$
▶ (+	)	IV	$\mathbf{K}_+$
▶ (	)	III	
▶	/	I	
▶ /	d		$\mathbf{K}_d$
▶ /	←	IV	$\mathbf{K}_/$
▶	←	Конец	

В результате получаем:

$\mathbf{K}_a\ \mathbf{K}_b\ \mathbf{K}_c\ \mathbf{K}_*\ \mathbf{K}_+\ \mathbf{K}_d\ \mathbf{K}_/$ или **abc*+d/**

Оглавление

Теперь необходимо выполнить польскую запись в соответствии с ее определением.

2. Выполнение операций:

Стек операндов	**Команда**	**Тройка**
$\varnothing$	K_a	
a	K_b	
a b	K_c	
a b c	K_*	$T_1 = b*c$
a T_1	K_+	$T_2 = a+T_1$
T_2	K_d	
T_2 **d**	$K_/$	$T_3 = T_2/d$
T_3		

Контрольные вопросы

1. Определите, автомат с магазинной памятью. В чем особенность его использования при грамматическом разборе?
   Ответ.
2. Как построить автомат с магазинной памятью по синтаксическим диаграммам? В чем заключается особенность полученной программы?
   Ответ.
3. Определите LL(k) грамматики. Какими особенностями они обладают? В чем заключается метод рекурсивного спуска?
   Ответ.
4. Определите LR(k) грамматики. Какими особенностями они обладают? В чем заключается стековый метод?
   Ответ.
5. Какой вид имеет польская запись? Для чего она используется?
   Ответ.
6. В чем заключается алгоритм Бауэра-Замельзона?
   Ответ.

5 Распределение памяти под программы и данные

После распознавания конструкций и определения таблиц переменных, именованных констант и временных переменных осуществляют распределение памяти под эти данные и программы. При этом могут использоваться разные типы распределения памяти.

Различают:

1) статическое;
2) автоматическое;
3) управляемое;
4) базированное.

Статическое распределение памяти выполняется:

а) во время компиляции – для подпрограмм и для инициализированных переменных;

б) во время компоновки – для неинициализированных переменных.

Автоматическое распределение памяти может выполняться для переменных подпрограмм, используемых только при активации подпрограммы. Эти переменные обычно размещаются в стеке.

Управляемое распределение памяти выполняется во время выполнения программы специальными подпрограммами, например, new и delete.

Базированное распределение памяти выполняется во время выполнения программы из выделенного буфера. Доступ к такой памяти осуществляется через вычисляемые адреса.

Кроме этого, различают:

1) локальную память;
2) глобальную память.

Локальная память предполагает ограниченный доступ (из подпрограммы, из файла и т. п.). В универсальных языках программирования локальная память чаще всего отводится в стеке, т.е. для нее используется автоматическое распределение. Однако статическая память C++ распределяется статически, но программно в нее ограничивается доступ из других подпрограмм.

Глобальная память предполагает неограниченный доступ из любого места программы. Как правило эта память распределяется статически.

Контрольные вопросы

1. Какие виды распределения памяти Вы знаете? Чем они различаются и для чего используются?
   Ответ.
2. Чем различаются локальная и глобальная виды памяти программы?
   Ответ.

6 Генерация и оптимизация кодов

Генерация машинного кода выполняется заменой распознанной конструкции соответствующими машинными командами, например, тройка сложения целых чисел может заменяться последовательностью команд:

Mov AX, <Op1>

Add AX, <Op2>

Mov AX, <Result>

В такой последовательности команд выполняется подстановка адресов операндов и результата.

Очевидно, что код, полученный таким образом, является не оптимальным. Поэтому при генерации кода выполняется оптимизация.

Все способы оптимизации можно разделить на две группы:

1) машинно-независимая оптимизация;
2) машинно-зависимая оптимизация.

Машинно-независимая оптимизация включает:

а) исключение повторных вычислений одних и тех же операндов;

б) выполнение операций над константами во время трансляции;

в) вынесение из циклов вычисления величин, не зависящих от параметров циклов;

г) упрощение сложных логических выражений и т. п.

Машинно-зависимая оптимизация включает:

а) исключение лишних передач управления типа «память-регистр»;

б) выбор более эффективных команд т. п.

Оба типа оптимизации предполагают разработку соответствующих семантических моделей для представления результатов распознавания конструкций. Обычно с этой целью используют разного типа таблицы.

Оглавление

Контрольные вопросы

1. Назовите машинно-независимые способы оптимизации кода. Почему они так названы?

   Ответ.

2. Назовите машинно-зависимые способы оптимизации кода. Почему они так названы?

   Ответ.

Литература

1. Д. Грис. Конструирование компиляторов для цифровых вычислительных машин – М.: Мир, 1975. – 544 с.
2. Ахо А., Сети Р., Ульман Д. Компиляторы: принципы, технологии и инструменты. Пер. с англ. – М.: «Вильямс», 2003.
3. В. Дж. Рейуорд–Смит. Теория формальных языков. Вводный курс. – М.: Радио и связь, 1988. – 128 с.