

Министерство образования Российской Федерации

Пензенский государственный университет

И.Г. Кревский, М.Н. Селиверстов, К.В. Григорьева

ФОРМАЛЬНЫЕ ЯЗЫКИ, ГРАММАТИКИ И ОСНОВЫ
ПОСТРОЕНИЯ ТРАНСЛЯТОРОВ

Учебное пособие

(под ред. д.т.н., профессора А.М. Бершадского)

Пенза 2003

УДК 681.3

Рецензенты:

Кафедра «Системы автоматизированного проектирования» Воронежского
государственного технического университета

Доктор технических наук, профессор,
К.Б. Скобельцын

Санкт-Петербургский государственный
технический университет

И.Г. Кревский, М.Н. Селиверстов, К.В. Григорьева

Формальные языки, грамматики и основы построения трансляторов: Учебное
пособие / Под ред. А.М. Бершадского – Пенза: Изд-во Пенз. гос. ун-та, 2002.
– 124 с.: 15 ил., 6 табл., библиогр. 12 назв.

Представлен материал для изучения разделов, посвященных формальным языкам, грамматикам и разработке трансляторов в курсах «Лингвистическое и программное обеспечение САПР и «Теория вычислительных процессов и структур». Подробно рассмотрены основные вопросы - теория грамматик и автоматов, лексический анализ, нисходящий и восходящий синтаксический анализ, построение программы синтаксического анализа для заданного синтаксиса, применение синтаксических диаграмм для построения анализаторов, таблично-управляемые и программно-управляемые анализаторы, формирование постфиксной записи арифметических выражений и операторов языка, генерация объектного кода программы.

Приведены методические указания к лабораторным работам с вариантами заданий, а также требования к курсовому проекту по построению компиляторов.

Учебное пособие разработано на кафедре «Системы автоматизированного проектирования» и предназначено для студентов специальностей 22.03.00 «Системы автоматизированного проектирования» и 35.15.00 «Математическое обеспечение и администрирование информационных систем», также может быть использовано для подготовки дипломированных специалистов по другим специальностям направления 654600 «Информатика и вычислительная техника», бакалавров по направлению 552800 «Информатика и вычислительная техника».

СОДЕРЖАНИЕ:

ВВЕДЕНИЕ	4
1. СТРУКТУРА КОМПИЛЯТОРА. ТИПЫ ТРАНСЛИРУЮЩИХ ПРОГРАММ	7
2. ОПРЕДЕЛЕНИЕ ЯЗЫКА. СИНТАКСИС И СЕМАНТИКА	15
3. КЛАССИФИКАЦИЯ ГРАММАТИК. ИЕРАРХИЯ ХОМСКОГО.....	20
4. ПРОБЛЕМА РАЗБОРА.....	26
5. ЛЕКСИЧЕСКИЙ АНАЛИЗ	31
6. КОНЕЧНЫЕ АВТОМАТЫ.....	35
7. КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ	40
8. LL(1) - ГРАММАТИКИ	45
9. ПРЕОБРАЗОВАНИЕ ГРАММАТИК В LL(1) ФОРМУ	50
10. ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО ГРАФА	56
11. ПОСТРОЕНИЕ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА ДЛЯ ЗАДАННОГО СИНТАКСИСА	61
12. ПОСТРОЕНИЕ ТАБЛИЧНО-УПРАВЛЯЕМОЙ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА	66
13. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ	71
14. РАБОТА С ТАБЛИЦЕЙ СИМВОЛОВ	84
15. ВОССТАНОВЛЕНИЕ ПРИ СИНТАКСИЧЕСКИХ ОШИБКАХ	88
16. ПОСТФИКСНАЯ ЗАПИСЬ.....	91
17. ВНУТРЕННИЕ ФОРМЫ	103
18. МЕТОДЫ ГЕНЕРИРОВАНИЯ КОДА	105
19. ЛИТЕРАТУРА	107
20. ЛАБОРАТОРНЫЕ РАБОТЫ.....	108
ЛАБОРАТОРНАЯ РАБОТА №1. РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА	109
ЛАБОРАТОРНАЯ РАБОТА №2. РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА	110
ЛАБОРАТОРНАЯ РАБОТА №3. ФОРМИРОВАНИЕ ПОСТФИКСНОЙ ЗАПИСИ.....	112
ЛАБОРАТОРНАЯ РАБОТА №4. РАЗРАБОТКА ГЕНЕРАТОРА КОДА..	113
ПРИЛОЖЕНИЕ А. ВАРИАНТЫ ЗАДАНИЙ К ЛАБОРАТОРНЫМ РАБОТАМ.....	117
ПРИЛОЖЕНИЕ Б. ТРЕБОВАНИЯ К КУРСОВОМУ ПРОЕКТУ	123

ВВЕДЕНИЕ

Большую часть программы курса «Лингвистическое и программное обеспечение САПР» занимают вопросы, связанные с изучением формальных языков, грамматик и основ построения трансляторов. По опыту преподавания этой дисциплины студентам специальности 22.03.00 «Системы автоматизированного проектирования» освоение данного материала требует от них, наряду с владением навыками программирования, серьезного изучения теоретических основ построения трансляторов. Именно поэтому значительная часть лабораторного практикума и курсовое проектирование по дисциплине «Лингвистическое и программное обеспечение САПР» посвящены созданию транслятора. За рамками данного пособия сознательно оставлены вопросы, касающиеся общих вопросов организации программного обеспечения САПР, технологий структурного и объектно-ориентированного программирования, тем более что большая часть этого материала в ПГУ вынесена в отдельные дисциплины регионального компонента. Также не рассмотрены языки проектирования САПР, поскольку их рассмотрению посвящено отдельное пособие, находящееся сейчас в стадии написания.

Можно сказать, что изучение теоретических основ и приобретение практических навыков написания транслятора имеют большое воспитательное значение. Не секрет, что многие студенты информационных специальностей, а часто и выпускники вузов, прекрасно умея разрабатывать программный код, абсолютно не умеют правильно поставить задачу, искренне не понимают, что любая программа будет эффективно и корректно работать лишь в том случае, когда она предварительно правильно спроектирована. Разработка транслятора - это прекрасный способ понять на собственном опыте, что, не выполнив предварительных этапов (например, не разработав правильного формального описания языка и допускающей детерминированный анализ грамматики), невозможно сделать хорошо работающую программу.

Несмотря на то, что основы формальных языков, грамматик и построения трансляторов можно считать достаточно устоявшейся областью знания, литературы по этим вопросам явно недостаточно. Значительная часть книг издана уже давно и ныне сохранилась в вузовских библиотеках в единичных экземплярах, а порой просто утеряна. Отличительной особенностью предлагаемого пособия является наличие в нем, наряду с рассмотрением теоретических вопросов, описания лабораторных работ, заданий на лабораторные работы и на курсовое проектирование. Эта часть материалов является расширением ранее опубликованного в [1].

Так как в процессе преподавании данной дисциплины параллельно происходит освоение теории и практическое написание транслятора, на деле приходится идти на компромисс между краткостью и полнотой изложения теории. В ряде книг [2,3] представлены описания трансляторов почти без рассмотрения теоретических вопросов. Наиболее удачный пример рассмотрения разработки транслятора с минимальным изложением теории представляет посвященная данной теме глава [4]. Материалы этой книги были во многом использованы в главах 2 и 10-12 настоящего пособия. Несколько более формальное описание разработки транслятора представлено в [5]. В наибольшей степени компромисс между необходимой строгостью изложения формальных языков и грамматик, с одной стороны, и относительной простотой и краткостью, с другой стороны, выдерживается в [6]. Эта книга может быть особенно полезна при углубленном изучении материала глав 3-9 данного пособия.

Отдельные вопросы формальных языков и грамматик представлены в [7], но соответствующий раздел книги рассматривает лишь небольшую часть материала. При написании главы 16 настоящего пособия были частично использованы материалы [8]. Более детальное и подробное рассмотрение вопросов формальных языков и грамматик можно найти в [9-11]. Как видно из списка литературы, большая часть ее была издана в 70-80 годы. Отрадно, что в последние годы возобновилось издание серьезной литературы по

компьютерным технологиям. Материалы недавно изданной [12] были использованы при написании главы 13.

Данное пособие может использоваться также для изучения большей части дисциплины «Теория вычислительных процессов и структур» специальности 35.15.00 «Математическое обеспечение и администрирование информационных систем», а также сходных курсов, изучаемых студентами других специальностей направления подготовки дипломированных специалистов 65.46.00 и подготовки бакалавров 55.28.00 «Информатика и вычислительная техника»

1. СТРУКТУРА КОМПИЛЯТОРА. ТИПЫ ТРАНСЛИРУЮЩИХ ПРОГРАММ

Каждый компьютер способен непосредственно выполнять ограниченный набор относительно простых команд. Любые более сложные действия представляются последовательностью таких команд. Для выполнения программы, написанной на языке высокого уровня, ее обычно переводят в последовательность команд машинного кода.

Исходная программа (написанная на каком-либо языке программирования) представляет собой последовательность символов, которая вводится в компьютер и преобразуется в форму, пригодную для непосредственного выполнения.

Компилятор является программой, которая способна воспринимать строку символов определенного вида (т.е. текст программы на исходном языке) и выдавать другую строку символов (программу на машинном языке). Компиляторам присущ ряд общих черт, что упрощает процесс создания компилирующих программ. В состав любого компилятора входят три основных компонента:

- лексический анализатор (блок сканирования);
- синтаксический анализатор;
- генератор кода машинных команд.

Принцип действия анализаторов можно описать с помощью формальных моделей, в то время как для генератора кода пока не существует общепринятых четких формальных представлений. На фазе **лексического анализа** исходный текст программы в виде цепочки несвязанных друг с другом символов разбивается на единицы, называемые **лексемами**. Такими текстовыми единицами являются ключевые слова, используемые в языке (например, IF, DO и др.), имена переменных, константы и знаки операций (например, * или +). Далее эти слова рассматриваются как неделимые образования, а не как группы отдельных символов. После разбиения программы на лексемы следует фаза **синтаксического анализа**, называемая

грамматическим разбором, на которой проверяется правильность следования операторов. Например, для предложения IF, имеющего вид IF выражение THEN предложение; грамматический разбор состоит в том, чтобы убедиться, что вслед за лексемой IF следует правильное выражение, за этим выражением следует лексема THEN, за которой в свою очередь следует правильное предложение, оканчивающееся знаком ";". Последним выполняется процесс **генерации кода**, который использует результаты синтаксического анализа и создает программу на машинном языке, пригодную к выполнению. Хотя в состав любого компилятора входят все три описанных выше компонента, их взаимодействие может осуществляться разнообразными способами. Рассмотрим наиболее распространенные варианты взаимосвязи между этими компонентами.

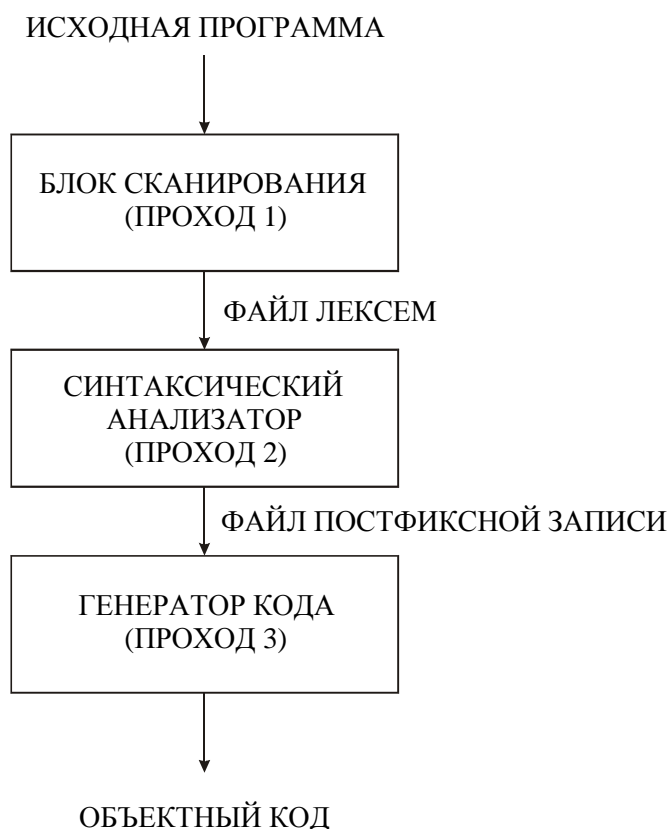


Рис.1.1. Трехпроходный транслятор.

Блок сканирования считывает исходную программу и представляет ее в форме файла лексем. Синтаксический анализатор читает этот файл и выдает

новое представление программы, например, в постфиксной форме. Наконец, этот файл считывается генератором кода, который создает объектный код программы.

Компилятор такого вида называется трехпроходным (рис.1.1), так как программа считывается трижды (исходный текст программы, файл лексем и файл в постфиксной форме).

Недостаток:

Невысокая скорость выполнения, так как в большинстве вычислительных систем операции, связанные с обращением к файлам, осуществляются сравнительно медленно.

Преимущества:

Относительная независимость каждой фазы компилирования. Так как связь между обрабатываемыми блоками осуществляется только через файлы данных, любой проход может быть реализован независимо от остальных. Это обеспечивает:

1. Возможность автономной разработки различных блоков компилятора разными разработчиками, необходимо только согласовать форматы промежуточных файлов.
2. Гибкость компилятора. Например, для реализации одного и того же языка для различных типов компьютеров, возможно использовать одни и те же блоки сканирования и синтаксического анализа, но написать специальные генераторы кода для каждого типа компьютера. При реализации семейства компиляторов с различных языков для одного типа компьютеров, очевидно, потребуются различные блоки сканирования и синтаксического анализа, но возможно использование общего генератора кода.
3. Минимальные требования к объему используемой оперативной памяти (модули различных фаз компиляции можно загружать по очереди, выгружая при этом предыдущий).

Для достижения высокой скорости компиляции применяется компилятор с однократной структурой (рис.1.2). На рисунке связи по управлению показаны сплошными линиями, передача данных – пунктиром.

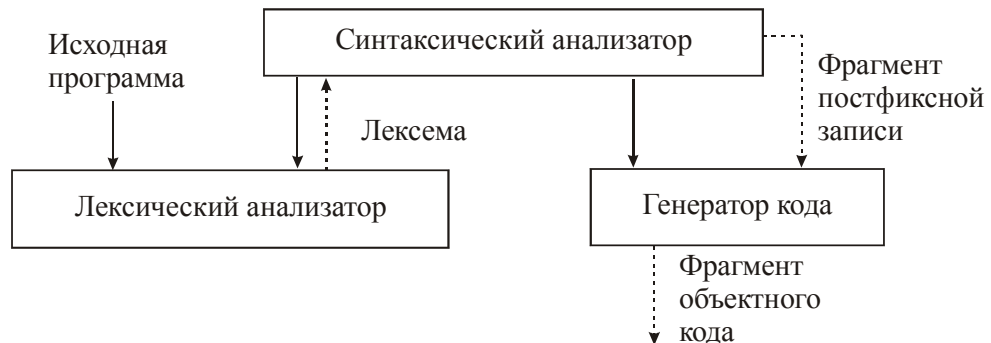


Рис.1.2. Однократный транслятор.

В этом случае синтаксический анализатор выступает в роли основной управляющей программы, вызывая блок сканирования и генератор кода, организованные в виде подпрограмм. Синтаксический анализатор постоянно обращается к блоку сканирования, получая от него лексему за лексемой из просматриваемой программы, до тех пор, пока не построит новый элемент постфиксной записи, после чего он обращается к генератору кода, который создает объектный код для этого фрагмента программы.

Преимущество:

Максимальная эффективность и скорость выполнения, так как программа просматривается лишь однажды, количество операций обращения к файлам минимально (только чтение из исходного и запись в объектный файлы).

Недостатки:

1. Проблемы при организации переходов вперед. Например, во время обработки предложения

GOTO метка ;

могут встретиться трудности, так как "метка" еще не встречалась в тексте программы.

2. Неоптимальность создаваемой объектной программы. Например, если встречается текст:

$$A = (B + C);$$

$$P = (B + C) + (E + M);$$

компилятор мог бы построить более эффективный объектный код, трансформировав программу следующим образом:

$$A = (B + C);$$

$$P = A + (E + M);$$

Однако однопроходный компилятор может утратить часть нужной информации к тому времени, когда в тексте встретится формула $(E + M)$.

3. Поскольку однопроходный компилятор должен полностью размещаться в памяти, его реализация сопровождается повышенными требованиями к ресурсу памяти, которые не всегда можно удовлетворить, имея систему с ограниченным объемом памяти.

Для повышения эффективности выполнения объектной программы в процесс компилирования может включаться фаза оптимизации. Блок оптимизации легко встраивается в трехпроходный компилятор, где размещается, обычно, между синтаксическим анализатором и генератором кода. На этой фазе постфиксный файл используется в качестве входных данных и создается новый файл, содержащий постфиксную запись эквивалентной программы с улучшенными характеристиками. Поскольку блок оптимизации записывает свои выходные данные в формате постфиксного файла, генератор кода не нуждается в изменении. На практике возможность оптимизации предусматривается по желанию пользователя: если необходимо, чтобы время компилирования было небольшим, блок оптимизации игнорируется; если же требуется получить программу с высокой скоростью выполнения, то после работы синтаксического анализатора вызывается блок оптимизации.

Возможны и другие способы структурной организации компилятора. На рис.1.3 показана структура двухпроходного компилятора, занимающая

промежуточное положение между двумя описанными выше вариантами организации. В этом случае синтаксический анализатор, вызывая блок сканирования, получает лексему за лексемой и строит файл постфиксной записи программы. Генератор кода считывает этот файл и создает объектный код программы. Подобной структуре свойственно относительно небольшое время выполнения, так как программа считывается лишь дважды (исходный текст и постфиксная запись). В этом случае легко разрешается проблема с оператором перехода вперед на метку, так как эта метка считывается на фазе первого прохода, перед вызовом генератора кода. В такой компилятор при необходимости легко включить блок оптимизации.

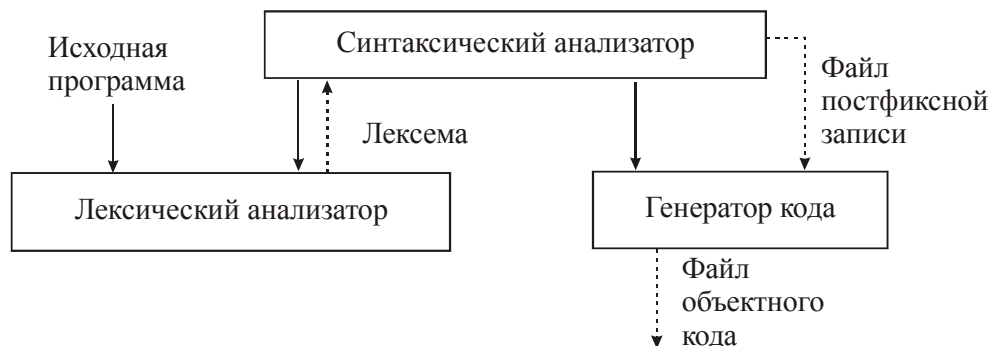


Рис.1.3. Двухпроходный транслятор.

Возможны различные модификации рассмотренных схем. Ясно, что рассмотренные типы компиляторов проявляют свои достоинства в определенных условиях работы и оказываются неэффективными в других случаях.

Интерпретаторы реализуют принцип, альтернативный компилированию. Компиляторы и интерпретаторы имеют много общего. Интерпретатор, тоже вначале просматривает исходную программу и выделяет в ней лексемы. Для этого используются блоки сканирования и анализаторы, аналогичные тем, которые входят в состав компилирующих программ. Однако интерпретатор вместо построения объектного кода,

способного выполнять те или иные операции, сам производит соответствующие действия.

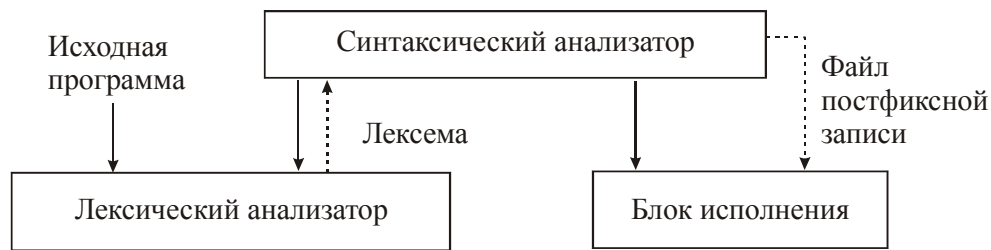


Рис. 1.4. Структура интерпретатора.

Достоинства интерпретатора:

- относительная простота реализации;
- удобство отладки программ.

Достоинства компилятора:

- скорость выполнения ;
- независимость выполняемого кода от системы программирования;
- возможность передавать программы заказчикам без исходных текстов.

Большинство современных интерпретаторов выполняют не исходный код, а преобразуют его в промежуточный, который затем интерпретируется. Это позволяет несколько повысить скорость исполнения и избежать передачи заказчикам исходных текстов. Для трансляции классических языков программирования (С, С++, Паскаль, Delphi и др.) обычно используются компиляторы. Как интерпретаторы выполнены большинство реализаций Бейсика и языков управления СУБД. Язык сетевого программирования Java реализован как интерпретируемый специальной виртуальной Java-машиной. Это обеспечивает возможность исполнения промежуточного кода Java на любом типе компьютера, где имеется такая виртуальная Java-машина.

Поскольку компилятор преобразует исходную программу в совокупность битов, полученную строку битов можно использовать и на

другой машине. Так как подготовка программы для одного типа ЭВМ осуществляется с помощью ЭВМ другого типа, соответствующие компиляторы получили название "**крассовых**" (т.е. перекрестных).

Конвертор – это транслятор с одного языка на другой язык того же уровня. Примером конвертора может быть программа, преобразующая код на языке Паскаль в код на С, или данные об объекте проектирования во внутреннем формате одной САПР в формат другой САПР.

Уточним термины. Под **транслятором** понимают любую программу, которая преобразует строку символов (т.е. исходную программу) в другую строку символов (объектную программу). Результатом этого процесса может быть как программа на машинном языке для той или иной машины, так и исходный текст программы на каком-либо другом языке. Термин "**компилятор**" будем использовать, понимая под ним программу, которая осуществляет классическое преобразование исходной программы в программу на машинном языке. Если же будут подразумеваться все разновидности процесса трансляции, будем использовать термин "**транслятор**".

Контрольные вопросы

1. Каковы преимущества и недостатки одно-, двух- и трехпроходных компиляторов?
2. Чем отличаются интерпретатор и компилятор?
3. В каких случаях применяются конверторы?
4. Для чего нужны кросс-компиляторы?

2. ОПРЕДЕЛЕНИЕ ЯЗЫКА. СИНТАКСИС И СЕМАНТИКА

Прежде чем приступить к созданию компилятора, необходимо иметь четкое и однозначное определение исходного языка.

Можно представить язык состоящим из ряда строк (последовательностей символов). В описании языка определяется, какие строки принадлежат этому языку (синтаксис языка), и значение этих строк (семантика языка). **Синтаксис** – множество правил, которые задают множество формально правильных предложений. Синтаксис для конечного языка (состоящего из конечного числа строк) можно специфицировать, задав список строк.

Строки, принадлежащие языку, обычно называется предложениями языка. В реальных языках – бесконечное число предложений, так что их синтаксис нельзя определить путем перечисления этих предложений. Синтаксис очень простого языка можно описать на естественном языке, например – "все строки, состоящие только из 1 и 0" тогда 1111 и 1000110 – принадлежат языку, а 1020 – нет. Однако естественным языкам свойственны неоднозначности, поэтому для спецификации синтаксиса более сложных языков применяется более формальный метод определения синтаксиса.

Рассмотрим, например, предложение "Кошки спят". Слово "кошки" – подлежащее, а "спят" – сказуемое. Это предложение принадлежит языку, который можно описать, например, при помощи следующих синтаксических правил:

`<предложение> ::= <подлежащее><сказуемое>`

`<подлежащее> ::= кошки | собаки`

`<сказуемое> ::= едят | спят`

Смысл этих трех строк таков: предложение состоит из подлежащего, за которым следует сказуемое. Подлежащее состоит либо из одного слова "кошки", либо из одного слова "собаки". Сказуемое состоит либо из слова "спят", либо из слова "едят".

Идея заключается в том, что любое предложение можно получить из начального символа $\langle \text{предложение} \rangle$ последовательным применением правил подстановки.

Формализм, или нотация, использованный при написании этих правил, называется **Бэкуса–Наура формой (БНФ)**. синтаксические единицы $\langle \text{предложение} \rangle$, $\langle \text{подлежащее} \rangle$ и $\langle \text{сказуемое} \rangle$ называются нетерминальными символами, слова "кошки", "собаки", "спят", "едят" – терминальными символами, а правила – порождающими правилами. Символы " $::=$ ", " $|$ ", а также скобки нетерминальных символов " $\langle \rangle$ " – это метасимволы языка БНФ (метасимволы не принадлежат описываемому языку, а относятся к языку описания. **Семантика** задает значение всем предложениям языка. Изменим пример про животных:

$\langle \text{предложение} \rangle ::= \langle \text{подлежащее} \rangle \langle \text{сказуемое} \rangle$

$\langle \text{подлежащее} \rangle ::= \text{люди} \mid \text{собаки}$

$\langle \text{сказуемое} \rangle ::= \text{говорят} \mid \text{спят}$

Методы формального описания семантики разработаны недостаточно хорошо и в этих целях часто используется естественный язык.

Для рассмотрения задачи спецификации синтаксиса дадим несколько определений.

Алфавит – множество символов. Например: Русские буквы, Латинские буквы, цифры.

Если A – алфавит, A^* (замыкание A) обозначает множество всех строк (включая пустую строку), составленных из символов, входящих в A . A^+ обозначает множество всех строк (исключая пустую строку), состоящих из символов, входящих в A . Пустая строка обычно обозначается с помощью ε (эпсилон).

Синтаксис языка можно определить, пользуясь системой изображения множеств, например $L = \{ 0^n 1^n \mid n \geq 0 \}$. Данный язык включает строки, состоящие из одного или более нулей и того же числа последующих единиц, а также пустую строку.

Более сложный синтаксис языка лучше определять с помощью грамматики. В грамматику входит набор правил для получения предложений языка. Возьмем синтаксис L и воспользуемся следующими правилами:

$$1. S \rightarrow 0S1$$

$$2. S \rightarrow \varepsilon$$

Чтобы вывести предложение этого языка, поступим следующим образом. Начнем с символа S и заменим его на $0S1$ или ε . Если S опять появился в полученной строке, его опять можно заменить с помощью одного из этих правил, и т.д. Полученная таким образом любая строка, не содержащая S , является предложением этого языка. Например,

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$$

Последовательность таких шагов называется выводом строки 000111 , а символ \Rightarrow служит для разделения шагов вывода. Все предложения данного языка можно вывести, руководствуясь двумя правилами, любая строка, которую нельзя вывести с их помощью, не является предложением этого языка. Грамматику часто называют системой перезаписи.

Грамматика определяется как четверка (V_t, V_n, P, S) , где V_t – алфавит, символы которого называются **терминальными** (терминалами), из них строятся цепочки порождаемые грамматикой. V_n – алфавит, символы которого называется **нетерминальными** (нетерминалами), используются при построении цепочек; они могут входить в промежуточные цепочки, но не должны входить в результат порождения. V_t и V_n не имеют общих символов, т.е. $V_t \cap V_n = \emptyset$, полный алфавит (словарь) грамматики V определяется как $V_t \cup V_n$. P – набор **порождающих правил**, каждый элемент которых состоит из пары (α, β) , где α находится в V^+ , а β в V^* . α – левая часть правила, а β – правая, т.е. цепочки, построенные из символов алфавита V . Правило записывается $\alpha \rightarrow \beta$. S принадлежит V_n и называется **начальным символом** (аксиомой). Этот символ – отправная точка в получении любого предложения языка.

Грамматикой, генерирующей язык $L = \{ 0^n 1^n \mid n \geq 0 \}$ является $G_0 = (\{0,1\}, \{S\}, P, S)$, где $P = \{ S \rightarrow 0S1, S \rightarrow \varepsilon \}$.

Грамматикой, генерирующей язык $L = \{ a^n b^m \mid n, m \geq 0 \}$ является $G_0 = (\{ a, b \}, \{ S, A, B \}, P, S)$, где $P = \{ S \rightarrow AB, A \rightarrow aA, A \rightarrow \varepsilon, B \rightarrow bB, B \rightarrow \varepsilon \}$

Начав с символа S и применяя последовательно по одному из правил замены нетерминала в выводимой строке можно, например, генерировать строку $aaabb$:

$$S \Rightarrow AB \Rightarrow aAB \Rightarrow aaAB \Rightarrow aaaAB \Rightarrow aaaB \Rightarrow aaabB \Rightarrow aaabbB \Rightarrow aaabb$$

Каждая строка, которую можно вывести из начального символа, называется *сентенциальной формой*. Предложение – сентенциальная форма, содержащая только терминалы. Терминалы будем обозначать строчными (маленькими) буквами, а нетерминалы – прописными (большими).

Как правило, предложения языка можно генерировать с помощью более чем одной грамматики. Две грамматики, генерирующие один и тот же язык, называют **эквивалентными**. С точки зрения разработчика транслятора одна грамматика может быть гораздо удобней другой.

Контрольные вопросы

1. Что определяет синтаксис языка?
2. В чем отличие синтаксиса языка от семантики языка?
3. Что такое грамматика и как она задается?
4. Чем различаются терминальные и нетерминальные символы языка?
5. Чем определяется возможность появления того или иного символа в предложении языка?
6. Что такое «начальный символ» и чем он отличается от других символов языка?
7. Задана грамматика с порождающими правилами:

$$S \rightarrow (S) \qquad S \rightarrow \varepsilon ,$$

$$S \rightarrow SS \qquad \text{где } S - \text{начальный символ, } \varepsilon - \text{пустая строка}$$

Принадлежат ли языку, генерируемому данной грамматикой следующие строки (аргументировать ответ, в случае принадлежности, доказать с помощью вывода):

a) строка $((1))$,

b) строка $((()))$

3. КЛАССИФИКАЦИЯ ГРАММАТИК. ИЕРАРХИЯ ХОМСКОГО.

Ограничение типов правил, которые могут появляться в грамматике позволяет определить ряд специальных классов грамматик. Одна из стандартных классификаций известна как иерархия Хомского. Ее описывают следующим образом:

1. Любая грамматика определенного ранее вида – **грамматика типа 0**.
2. Если для всех правил вида $\alpha \rightarrow \beta$, $|\alpha| \leq |\beta|$, где $|\alpha|$ и $|\beta|$ – длина, т.е. число символов соответственно α и β , то грамматика называется **грамматикой типа 1, или контекстно-зависимой (КЗ)**.
3. Если все левые части правил грамматики состоят из одного нетерминального символа, то это **грамматика типа 2, или контекстно-свободная (КС)**.
4. Грамматика называется **грамматикой типа 3, или регулярной**, если каждое правило грамматики имеют одну из следующих форм.

В случае грамматики, выровненной вправо, в правой части грамматики имеется не более одного нетерминала, который может быть только самым правым символом:

$$A \rightarrow a$$

$$A \rightarrow aB$$

В случае грамматики, выровненной влево, в правой части грамматики имеется не более одного нетерминала, который может быть только самым левым символом:

$$A \rightarrow a$$

$$A \rightarrow Ba$$

Иерархия – включающая, т.е. все грамматики типа 3 являются грамматиками типа 2 и т.д. Иерархия грамматик соответствует иерархии языков. Например, если язык можно генерировать с помощью грамматики типа 2, то его называют языком типа 2. Необходимо помнить, что если для

генерации языка можно использовать несколько грамматик, то тип языка соответствует грамматике с наибольшим типом.

Иерархия Хомского важна с точки зрения построения трансляторов с различных языков. Чем меньше ограничений в грамматике, тем сложнее ограничения, которые можно наложить на генерируемый язык. Чем более универсален класс используемой грамматики, тем больше свойств языка мы можем описать. Однако чем более универсальна грамматика, тем сложнее должна быть программа, распознающая строки соответствующего языка.

Грамматики типа 3 можно использовать для описания некоторых свойств языков программирования или высокоуровневых языков описания аппаратуры. Например, для генерирования идентификаторов по определению многих языков программирования можно воспользоваться следующими правилами:

$$\begin{array}{ll} I \rightarrow l & R \rightarrow d \\ I \rightarrow l R & R \rightarrow l R \\ R \rightarrow l & R \rightarrow d R \end{array}$$

где буква (l) и цифра (d) обозначают терминалы (для краткости будем считать так, потому что перечисление всех возможных букв и цифр потребовало бы написания слишком большого числа правил). Иногда удобно объединять правые части правил, имеющих одинаковые левые части. Вышеприведенную грамматику можно также записать в виде:

$$\begin{array}{l} I \rightarrow l / l R \\ R \rightarrow l / d / l R / d R \end{array}$$

Вертикальную черту здесь надо понимать как "или".

Многие "локальные" средства языков программирования, например константы, ключевые слова языка и строки, представляются с помощью грамматик типа 3. Некоторые очень простые языки описания аппаратуры также можно описать с помощью регулярной грамматики. Однако грамматики типа 3 генерируют только строго ограниченные типы языков – регулярные выражения.

В алфавите A к регулярным выражениям относятся следующие:

1. Элемент A (или пустая строка).

Если P и Q – регулярные выражения, то регулярными будут также и выражения

2. PQ (Q следует за P)

3. P / Q (P или Q)

4. P^* (нуль или более экземпляров P)

В алфавите $\{a, b, c\}$

ab^* / ca^* – регулярное выражение, которое описывает язык, включающий следующие строки (помимо прочих):

$abb \ c \ caaa \ ab \ ca$

Пример регулярного выражения. Регулярное выражение, описывающее идентификатор, имеет вид:

$L (L / D)^*$, где L обозначает букву, D – цифру.

У регулярных выражений есть существенные ограничения. Например, регулярное выражение не может задавать шаблоны скобок произвольной длины, и, следовательно, их нельзя генерировать с помощью грамматики типа 3.

Пример нерегулярного выражения. Рассмотрим язык, состоящий из строк открывающих и закрывающих скобок (плюс пустая строка), обладающих следующими свойствами:

- 1.) При чтении слева направо число встреченных закрывающих скобок никогда не превышает число встреченных открывающих скобок.
- 2.) В каждой строке содержится одинаковое число открывающих и закрывающих скобок.

Например, следующие строки принадлежат языку:

$() (()) (())$

$((() (((()))) (())$

а приводимые ниже – нет:

$((() ())$ – не соответствует правилу 2.

$(())(())$ – не соответствует правилу 1.

Не существует способа представления этого языка с помощью регулярного выражения или его генерирования с помощью грамматики типа 3. Однако этот язык генерируется следующей контекстно-свободной грамматикой:

$$S \rightarrow (S)$$

$$S \rightarrow SS$$

$$S \rightarrow \varepsilon$$

В большинстве языков программирования и языков описания аппаратуры имеются пары скобок, которые необходимо согласовывать, например:

$()$, $[]$, `begin end`

каждой открывающей скобке должна соответствовать закрывающая.

Так

`begin () end` – правильно

`begin (end)` – неправильно

Контекстно-свободная грамматика позволяет специфицировать подобные ограничения. Как правило, большая часть синтаксиса языков программирования и специализированных языков САПР описывается с помощью КС-грамматики. Однако у большинства языков есть некоторые свойства, которые нельзя выразить с помощью КС-грамматики. Например, присваивание $X := Y$ может быть допустимым, если объявлено, что X и Y имеют соответствующие типы, или недопустимым при несоответствии типов. Условие такого вида не может быть специфицировано КС-грамматикой, и компиляторы обычно выполняют проверку типа не на фазе формального синтаксического анализа. Однако идею КС-грамматики можно расширить, включив некоторые контекстно-зависимые свойства языков.

Двухуровневые грамматики (W-грамматики, названной так в честь их изобретателя – А. ван Вейнгаардена). Идея применения двухуровневой грамматики состоит в том, что если правила обычной грамматики

обеспечивают конечный способ описания языка, состоящего из бесконечного числа строк, то здесь вторая грамматика применяется для генерирования бесконечного числа правил, которые в свою очередь генерируют предложения языка. Применение второй грамматики позволяет избежать рутинной работы, связанной с написанием бесконечного числа порождающих правил. С помощью двухуровневой грамматики можно генерировать любой язык типа 0. Данная концепция является даже слишком мощной (КЗ свойства большинства машинных языков относительно просты).

Атрибутивные грамматики. Атрибуты применяются для описания КЗ (точнее К-несвободных) аспектов языка. Рассмотрим пример. Пусть в некотором языке идентификаторы могут иметь тип `int` или `char` и являются терминалами грамматики. Описание с помощью КС порождающих правил.

$$D \rightarrow \text{int } I$$

$$D \rightarrow \text{char } I$$

Описав идентификатор, мы хотим запомнить его тип. Этот тип будет свойством описания, видоизменим грамматику, чтобы это указать:

$$D (.ID, \text{MODE}) \rightarrow \text{int } (.MODE1) I(.ID1)$$

$$ID = ID1$$

$$\text{MODE} = \text{MODE1}$$

аналогично для `char`. `MODE`, `MODE1`, `ID`, `ID1` пишутся в скобках после терминала или нетерминала грамматики и представляют собой его признаки (атрибуты). Преимущество атрибутивных грамматик в том, что они выглядят как КС, но могут специфицировать КЗ конструкции языка. Фактически любой язык типа 0 можно описать с помощью атрибутивной грамматики. Так как языки программирования представляется как КС, к которым добавляются контекстные ограничения, атрибутивные грамматики хорошо подходят для их описания.

Контрольные вопросы

1. Сколько типов грамматик насчитывает иерархия Хомского? Назовите эти типы.

2. Каковы преимущества и недостатки регулярных грамматик?
3. Кто впервые описал двухуровневые грамматики и для чего они применяются?
4. Каково назначение атрибутивных грамматик?
5. Построить регулярную грамматику для идентификаторов. Идентификатор состоит из букв, цифр и символов "_" и начинается обязательно с буквы.
6. Найти регулярную грамматику, генерирующую тот же язык, что и грамматика со следующими порождающими правилами (S - начальный символ):

$$S \rightarrow A B$$

$$Y \rightarrow y / y Y$$

$$A \rightarrow X / Y$$

$$B \rightarrow b / b B$$

$$X \rightarrow x / x X$$

7. Построить регулярную грамматику, генерирующую выражение.
 $(101)^* (010)^*$

4. ПРОБЛЕМА РАЗБОРА

Компилятор должен решить проблему проверки строк символов, чтобы определить, принадлежат ли они данному языку, и если да, то распознать структуру строк в терминах порождающих правил грамматики. Эта проблема известна как проблема разбора. Исследуем грамматику с порождающими правилами (E - начальный символ).

$$1. E \rightarrow E + T$$

$$2. E \rightarrow T$$

$$3. T \rightarrow T * F$$

$$4. T \rightarrow F$$

$$5. F \rightarrow (E)$$

$$6. F \rightarrow x$$

$$7. F \rightarrow y$$

Ясно, что строка $(x + y) * x$ принадлежит данному языку. В частности, это можно вывести следующим образом (для каждого шага вывода указан номер применяемого правила):

$$2) E \Rightarrow T$$

$$3) \Rightarrow T * F$$

$$4) \Rightarrow F * F$$

$$5) \Rightarrow (E) * F$$

$$1) \Rightarrow (E + T) * F$$

$$2) \Rightarrow (T + T) * F$$

$$4) \Rightarrow (F + T) * F$$

$$6) \Rightarrow (x + T) * F$$

$$4) \Rightarrow (x + F) * F$$

$$7) \Rightarrow (x + y) * F$$

$$6) \Rightarrow (x + y) * x$$

Или же это можно вывести так:

$$2) E \Rightarrow T$$

$$3) \Rightarrow T * F$$

$$6) \Rightarrow T * x$$

$$4) \Rightarrow F * x$$

$$5) \Rightarrow (E) * x$$

$$1) \Rightarrow (E + T) * x$$

$$4) \Rightarrow (E + F) * x$$

$$7) \Rightarrow (E + y) * x$$

$$2) \Rightarrow (T + y) * x$$

$$4) \Rightarrow (F + y) * x$$

$$6) \Rightarrow (x + y) * x$$

На каждом этапе первого вывода самый левый нетерминал в сентенциальной форме замещался с помощью одного из порождающих правил грамматики. Поэтому данный вывод называется **левосторонним**. Второй вывод, на каждом этапе которого замещается самый правый нетерминал, называется **правосторонним**. Существуют также другие выводы, не являющиеся ни лево-, ни правосторонними, однако при реализации трансляторов они практически не используются. **Левосторонний разбор** предложения определяется как последовательность порождающих правил, применяемая для генерирования предложения посредством левостороннего вывода. В данном случае левосторонний разбор можно записать как

2,3,4,5,1,2,4,6,4,7,6.

Правосторонний разбор предложения является обратной последовательностью порождающих правил, используемых для генерирования предложения посредством правостороннего вывода; например, в вышеприведенном случае правосторонний разбор запишется в виде

6,4,2,7,4,1,5,4,6,3,2.

Обратный порядок последовательности порождающих правил связан с тем, что правосторонний разбор обычно ассоциируется с приведением предложения к начальному символу, а не с генерированием предложения из начального символа (см. ниже разбор снизу вверх). Заметим, что каждое порождающее правило используется в обоих выводах (или разборах) одинаковое число раз.

Дерево разбора. Вывод может быть описан также в терминах построения дерева, известного как синтаксическое дерево (или дерево разбора). В случае со строкой

$(x + y) * x$

синтаксическое дерево будет таким, как показано на рис. 4.1.

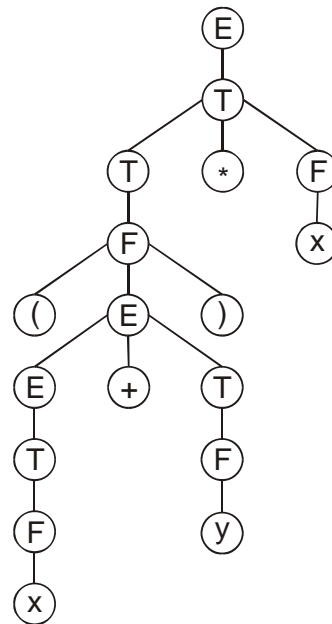


Рис. 4.1. Дерево разбора.

Проблему разбора можно свести к

- 1) нахождению левостороннего разбора;
- 2) нахождению правостороннего разбора;
- 3) построению синтаксического дерева.

Неоднозначные грамматики. В большинстве случаев левосторонний и правосторонний разборы и синтаксическое дерево являются уникальными. Однако для грамматики с порождающими правилами:

$$S \rightarrow S+S / x$$

предложение $x + x + x$ имеет два синтаксических дерева (рис. 4.2) и два левосторонних (и правосторонних) разбора:

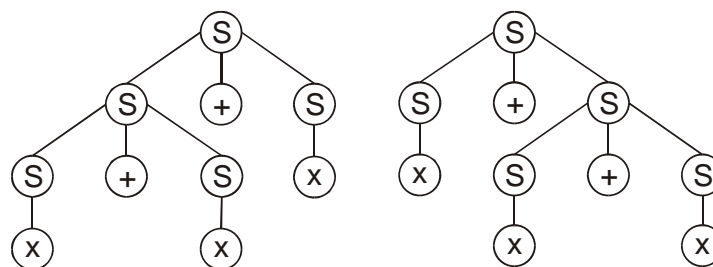


Рис. 4.2. Варианты разбора.

$S \Rightarrow S + S$	$S \Rightarrow S + S$
$\Rightarrow S + S + S$	$\Rightarrow x + S$
$\Rightarrow x + S + S$	$\Rightarrow x + S + S$
$\Rightarrow x + x + S$	$\Rightarrow x + x + S$
$\Rightarrow x + x + x$	$\Rightarrow x + x + x$

Если какое-либо предложение, генерированное грамматикой, имеет более одного дерева разбора, о такой грамматике говорят, что она неоднозначна. Эквивалентное условие заключается в том, что предложение должно иметь более одного левостороннего или правостороннего разбора. Задача установления неоднозначности грамматики является, в общем случае, неразрешимой, т.е. не существует универсального алгоритма, который принимал бы на входе любую грамматику и определял бы, однозначна она или нет. Некоторые неоднозначные грамматики можно преобразовать в однозначные, генерирующие тот же язык. Например, грамматика с порождающими правилами

$$S \rightarrow x / S + x$$

является однозначной и генерирует тот же язык, что и рассмотренная ранее неоднозначная грамматика.

Методы разбора обычно бывают нисходящими, т.е. начинают с начального символа и идут к предложению, или восходящими, т.е. начинают с предложения и идут к начальному символу.

Отказ от решения в разборе называют возвратом. Методы разбора могут быть недетерминированными и детерминированными в зависимости от того, возможен возврат или нет. Недетерминированные методы разбора весьма дорогие с точки зрения памяти и времени и крайне затрудняют включение в синтаксический анализатор действий, выполняемых во время компиляции, результаты которых позднее должны быть аннулированы (например, построение таблицы символов и т.п.). В дальнейшем мы будем рассматривать лишь детерминированные методы разбора. Основное

внимание в настоящем учебном пособии уделено методам нисходящего разбора, они же используются в лабораторных работах и при курсовом проектировании. Методы восходящего разбора рассмотрены в отдельной главе.

Контрольные вопросы

1. В чем состоит проблема разбора?
2. Что такое левосторонний и правосторонний разбор?
3. Почему основное внимание уделяется лево и правостороннему разбору при наличии большого числа разборов, не являющихся ни лево ни правосторонними?
4. К чему можно свести проблему разбора при построении синтаксического дерева?
5. Дайте определение неоднозначной грамматике.
6. Назовите отличия детерминированных методов разбора от недетерминированных.
7. Задана грамматика с порождающими правилами:

$$S \rightarrow S + T$$

$$F \rightarrow (S)$$

$$S \rightarrow T$$

$$F \rightarrow a$$

$$T \rightarrow T * F$$

$$F \rightarrow b$$

$$T \rightarrow F$$

- a) Построить синтаксическое дерево для выражения $(a + b) * a + a$.
 - b) Построить левосторонний разбор для выражения $(a + b) * a + a$.
 - c) Построить правосторонний разбор для выражения $(a + b) * (a + b)$.
8. Показать, что грамматика со следующими порождающими правилами является неоднозначной:

$$S \rightarrow \text{if } c \text{ then } S \text{ else } S$$

$$S \rightarrow x$$

$$S \rightarrow \text{if } c \text{ then } S$$

5. ЛЕКСИЧЕСКИЙ АНАЛИЗ

Первой фазой процесса компиляции является лексический анализ, то есть группирование строк литер, обозначающих идентификаторы, константы или слова языка и т.д., в единые символы (лексемы). Этот процесс может идти параллельно с другими фазами компиляции (например, в однопроходных компиляторах). Однако, в любом случае, при описании конструкции компилятора и его построения удобно представлять лексический анализ как самостоятельную фазу.

Блок сканирования (сканер) должен выдавать каждую лексему, устанавливая ее принадлежность тому или иному классу. Выбор классов зависит от особенностей транслируемого языка. Часто выделяют классы имен переменных, констант, ключевых слов, арифметических и логических операций ("+", "-", "*", "/" и т.д.), специальных символов ("=", ";", и т. д.)

Характер распознаваемых строк может намного упростить процесс лексического анализа. Например:

идентификаторы:

a1 one

числа:

100 10.54

ключевые слова языка:

begin if end

Все эти строки можно генерировать с помощью регулярных выражений. Например, вещественные числа можно генерировать посредством регулярного выражения $(+ | -) \mathbf{d} \mathbf{d}^* . \mathbf{d}^*$, где **d** обозначает любую цифру. Из выражения видно, что вещественное число состоит из следующих компонентов, расположенных именно в таком порядке:

1. возможного знака;
2. последовательности из одной или более цифр;
3. десятичной точки;
4. последовательности из нуля или более цифр.

Регулярные выражения эквивалентны грамматикам типа 3. Например, грамматика типа 3, соответствующая регулярному выражению для вещественного числа, имеет порождающие правила:

$$R \rightarrow +S / -S / dQ$$

$$S \rightarrow dQ$$

$$Q \rightarrow dQ / .F / .$$

$$F \rightarrow d / dF$$

где R - начальный символ, d – цифра.

Существует полное соответствие между регулярными выражениями (а потому и грамматиками типа 3) и конечными автоматами, более подробно рассмотренными в следующей главе.

Некоторые лексемы (например, $*$) могут быть распознаны по одному считанному символу, другие (такие, как $:=$) – по двум символам, а для идентификации третьих необходимо прочитать неизвестное заранее число символов (например, код константы). В последнем случае, чтобы найти конец лексемы, приходится считывать один лишний символ, не входящий в состав данной лексемы. Этот символ необходимо запоминать, чтобы при разборе следующей лексемы он не был утерян.

Лексический анализатор, наряду с разбиением исходного потока символов на лексемы, может включать в исходный текст дополнительную информацию или исключать из него строки символов. Примером дополнительно включаемой информации являются номера строк. Если их не включить, то информация о том, в какой строке исходного текста находилась та или иная лексема, будет, в случае выполняющего сканирование в отдельном проходе компилятора, утеряна после лексического анализа. Однако для диагностики на фазе синтаксического анализа необходимо иметь возможность ссылаться на ошибки в программе с указанием номеров строк оригинального исходного текста. С другой стороны, комментарии включаются в текст программы или описания объекта проектирования только для предоставления человеку дополнительных пояснений. Они никак

не влияют на генерируемый в дальнейшем код, и лексический анализатор обычно их просто исключает.

Пример структуры программы сканирования

Пусть реализуемый язык состоит только из оператора присваивания.
БНФ языка:

$\langle \text{Присваивание} \rangle ::= \langle \text{Идент} \rangle = \langle \text{Выражение} \rangle$

Правило показывает, что в левой части присваивания – идентификатор, далее следует символ присваивания (=), справа – выражение;

$\langle \text{Выражение} \rangle ::= \langle \text{Операнд} \rangle \mid \langle \text{Операнд} \rangle \langle \text{Бин.оп} \rangle \langle \text{Выражение} \rangle$

Выражение – это операнд, или операнд, за которым следует бинарная операция и выражение;

$\langle \text{Бин.оп} \rangle ::= "-" \mid "+" \mid "*" \mid "/"$

Бинарная операция – символ арифметической операции "-", "+", "*" или "/";

$\langle \text{Операнд} \rangle ::= \langle \text{Идент} \rangle \mid \langle \text{Const} \rangle$

Операнд – это идентификатор или константа;

$\langle \text{Идент} \rangle ::= \langle \text{Буква} \rangle$

Идентификатор состоит из одной буквы;

$\langle \text{Const} \rangle ::= \langle \text{Цифра} \rangle \langle \text{Const} \rangle \mid \langle \text{Цифра} \rangle$

Константа – последовательность цифр, состоящая хотя бы из одной цифры.

Лексический анализатор преобразует исходную программу в последовательность символов. Для удобства дальнейшей обработки лексем их разбивают на классы. В данном случае можно выделить следующие классы лексем:

- 1 – идентификатор;
- 2 – константа;
- 3 – символ присваивания;

4 – символы операций умножения и деления;

5 – символы операций сложения и вычитания.

Заметим, что необходимость разделения бинарных операций на две группы диктуется их различным приоритетом, играющим важную роль при генерации постфиксной записи.

Фрагмент программы лексического анализа:

```
/* prog - файл с транслируемой программой, lex -
выходной файл лексем */
while(!feof (prog))
{
    ch = readsym();
    /* чтение очередного символа ch с пропуском пробелов
    */

    if(isAlpha(ch))
        fprintf(lex, "%c %d", ch, 1);
    else
        if(isDigit(ch))
            digit();
            /* Процедура чтения числа и вывода его в файл */
        else
            if(ch == '=')
                fprintf(lex, "%c %d", ch, 3);
            else
                if(ch == '*' || ch == '/')
                    fprintf(lex, "%c %d", ch, 4);
                else
                    if(ch == '+' || ch == '-')
                        fprintf(lex, "%c %d", ch, 5);
                    else
                        printf("Недопустимый символ языка - %c \n", ch);
}
```

Контрольные вопросы

1. Дайте определение «лексемы».
2. В чем основная задача лексического анализатора?
3. Какие из перечисленных далее видов информации лексический анализатор должен включать в выходной файл лексем: символы языка, номер строки для каждой лексем, комментарии к программе, символы форматирования программы (пробелы, табуляции, переходы на новую строку).

6. КОНЕЧНЫЕ АВТОМАТЫ

Существует полное соответствие между регулярными выражениями (а поэтому и грамматиками типа 3) и конечными автоматами, которые определяются следующим образом:

Конечный автомат – это устройство для распознавания строк какого-либо языка. У него есть конечное множество состояний, отдельные из которых называются последними. По мере считывания каждой литеры строки контроль передается от состояния к состоянию в соответствии с заданным множеством переходов. Если после считывания последней литеры строки автомат будет находиться в одном из последних состояний, о строке говорят, что она принадлежит языку, принимаемому автоматом. В ином случае строка не принадлежит языку, принимаемому автоматом.

Конечный автомат формально определяется пятью характеристиками:

- конечным множеством состояний (K)
- конечным входным алфавитом (Σ)
- множеством переходов (δ)
- начальным состоянием ($S_0 \in K$)
- множеством последних состояний ($f \in K$)

$$M = (K, \Sigma, \delta, S_0, f).$$

Пример: Состояниями автомата являются A и B , входным алфавитом – $\{0,1\}$, начальным состоянием – A , множеством последних состояний – $\{A\}$, а переходами

$$\begin{array}{ll} \delta(A, 0) = A, & \delta(B, 0) = B, \\ \delta(A, 1) = B, & \delta(B, 1) = A. \end{array}$$

Эти переходы означают, что при чтении 0 в состоянии A управление передается в состояние A и т.д. При чтении строки

0 1 0 0 1 0 1 1

управление последовательно передается в следующем порядке через состояния:

$A, A, B, B, B, A, A, B, A.$

Так как A есть последнее состояние, строка принимается конечным автоматом, однако при чтении строки

0 0 1 1 1

автомат проходит через состояния

A, A, A, B, A, B

поскольку B не является последним состоянием, эта строка не принимается, т.е. она не принадлежит языку, принимаемому этим автоматом. В связи с тем, что нули не влияют на состояние автомата, а каждая единица изменяет его состояние с A на B и с B на A , и начальное состояние является тем же, что и последнее состояние, язык, принимаемый автоматом, состоит из тех строк, которые содержат четное число единиц.

Переходы можно представить с помощью таблицы (таблица 6.1) и схематически (рис.6.1).

Таблица 6.1

Состояния			
Вход		A	B
	0	A	B
	1	B	A

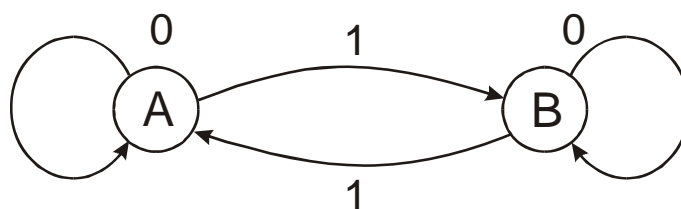


Рис.6.1. Переходы детерминированного конечного автомата.

Такой автомат называют детерминированным, потому что в каждом элементе таблицы переходов содержится одно состояние. В недетерминированном конечном автомате это положение не выдерживается.

Рассмотрим конечный автомат, определяемый следующим образом:

$$M_1 = (K_1, \Sigma_1, \delta_1, S_1, f_1),$$

Где $K_1 = \{A, B\}$, $\Sigma_1 = \{0, 1\}$, $S_1 = \{A\}$, $f_1 = \{B\}$,

а переходы представлены в таблице 6.2 и на рис.6.2:

Таблица 6.2

Состояния			
Вход		A	B
	0	\emptyset	$\{B\}$
	1	$\{A, B\}$	$\{B\}$

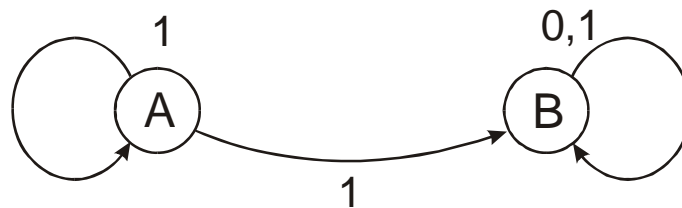


Рис.6.2. Переходы недетерминированного конечного автомата M_1 .

Первая строка будет принята, так как имеется переход (последовательность переходов), ведущий к последнему состоянию при чтении строки. Имеется также переход к непоследнему состоянию, но это не влияет на приемлемость строки. Поэтому прежде чем прийти к выводу о том, что строка не может быть принята недетерминированным конечным автоматом, необходимо перепробовать все возможные последовательности переходов.

Существует детерминированный конечный автомат M_2 , соответствующий автомату M_1 , который принимает тот же язык. Переходы автомата M_2 приведены в таблице 6.3 и на рис.6.3.

$M_2 = (K_2, \Sigma_2, \delta_2, S_2, f_2)$, где $K_2 = \{A, B, C\}$, $\Sigma_2 = \{0, 1\}$, $S_2 = \{A\}$, $f_2 = \{B\}$

Таблица 6.3

Состояния				
Вход		A	B	C
	0	C	B	C
	1	B	B	C

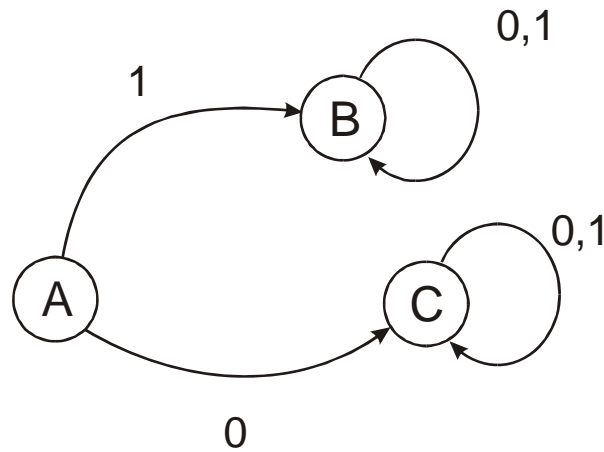


Рис.6.3. Переходы детерминированного конечного автомата M_2 .

Как и M_1 , M_2 принимает строки из нулей и единиц тогда и только тогда, когда строка начинается с единицы. Однако при распознавании строки с помощью M_2 возврат никогда не требуется, т.к. в процессе чтения определенного входного символа из любого состояния произойдет точно один переход к другому состоянию. Это значит, что при использовании M_2 время распознавания строки будет пропорционально ее длине.

Соответствие лексическому анализу заключается в том, что каждому языку типа 3 соответствует детерминированный конечный автомат, который распознает строки этого языка. Например, строки, генерируемые грамматикой G_1 с порождающими правилами:

$$A \rightarrow 1A \mid 1B \mid 1$$

$$B \rightarrow 0B \mid 1B \mid 0 \mid 1$$

где A – начальный символ, распознаются с помощью M_1 или M_2 . Грамматику получают из недетерминированного конечного автомата M_1 следующим образом:

1. Начальное состояние автомата становится начальным символом предложения грамматики.

2. Переходам

$$\delta(A, 1) = A,$$

$$\delta(B, 0) = B,$$

$$\delta(A, 1) = B,$$

$$\delta(B, 1) = B.$$

соответствуют порождающие правила

$$A \rightarrow 1A \mid 1B$$

$$B \rightarrow 0B \mid 1B$$

тому, что в состоянии A есть переход при чтении 1 к последнему состоянию B соответствует

$$A \rightarrow 1$$

и аналогично

$$B \rightarrow 0 \mid 1$$

Можно также, наоборот, из грамматики вывести автомат M_1 .

Контрольные вопросы

1. Какому типу грамматик по иерархии Хомского соответствуют конечные автоматы?
2. Дайте определение конечного автомата.
3. Чем отличается детерминированный конечный автомат от недетерминированного?
4. Можно ли однозначно преобразовать регулярное выражение в детерминированный конечный автомат?

7. КОНТЕКСТНО-СВОБОДНЫЕ ГРАММАТИКИ

Грамматики типа 3 (регулярные) удобны для генерирования символов, которые создаются во время лексического анализа, но они не очень подходят для описания способов объединения этих символов в предложения типичных языков высокого уровня. Например, как уже отмечалось выше, согласование скобок нельзя специфицировать с помощью грамматики типа 3. КС-грамматики (типа 2), хотя и не могут специфицировать все свойства типичных языков программирования, являются более универсальными и поэтому более пригодными в качестве основы для фазы синтаксического анализа (разбора) компиляции.

КС-грамматики традиционно служат основой для фазы синтаксического анализа компиляции. Если какой-либо язык программирования нельзя генерировать с помощью КС-грамматики, всегда можно найти такую КС-грамматику, которая генерирует надмножество языка, т.е. язык, включающий в себя заданный язык программирования.

Применение такой грамматики для синтаксического анализа означает, что ряд ограничений в реальном языке (например, обязательное объявление всех идентификаторов в программе) не будет проверяться анализатором. Однако в компиляторе нетрудно предусмотреть другие действия по выполнению необходимых проверок в исходной программе (например, за счет применения таблицы символов).

Из определения КС-грамматики ясно, что класс КС-грамматик более мощный (т.е. может генерировать больше языков), чем класс регулярных грамматик, но менее мощный, чем класс контекстно-зависимых (КЗ-грамматик). Язык

$$\{ a^n b^n \mid n \geq 0 \}$$

является контекстно-свободным, но не регулярным. Он генерируется грамматикой с порождающими правилами

$$S \rightarrow aSb \mid \varepsilon$$

С другой стороны, язык

$$\{ a^n b^n c^n / n \geq 0 \}$$

является контекстно-зависимым, а не контекстно-свободным.

Контекстно-свободные грамматики имеют ряд характеристик, для которых справедливы следующие положения.

1) Каноническая форма

а) Каждая КС-грамматика эквивалентна (т.е. генерирует тот же язык) грамматике в **нормальной форме Хомского**, т.е. со всеми порождающими правилами вида

$$A \rightarrow BC / a$$

при обычных условиях, касающихся терминалов и нетерминалов.

б) Каждая КС-грамматика эквивалентна грамматике в **нормальной форме Грейбаха**, т.е. со всеми порождающими правилами вида

$$A \rightarrow b\alpha$$

где α – строка нетерминалов (возможно, пустая).

2) Самовложение

Если в грамматике G есть нетерминал A , для которого

$$A \Rightarrow \alpha_1 A \alpha_2$$

(здесь α_1 и α_2 являются непустыми строками терминалов и/или нетерминалов), то о такой грамматике говорят, что она содержит самовложение. Например, две приведенные ниже грамматики содержат самовложение:

1) $G_1 = (\{S\}, \{a, b\}, P, S)$, где элементы P :

$$S \rightarrow aSb$$

$$S \rightarrow \varepsilon$$

2) $G_2 = (\{S, A\}, \{\mathbf{begin}, \mathbf{end}, [,]\}, P, S)$, где элементы P :

$$S \rightarrow \mathbf{begin} A \mathbf{end}$$

$$S \rightarrow \varepsilon$$

$$A \rightarrow [S]$$

В последнем случае об A и S говорят, что они проявляют свойство самовложения. Теоретически любая КС-грамматика, не содержащая

самовложения, эквивалентна регулярной грамматике и генерирует регулярный язык. С другой стороны, регулярная грамматика не может содержать самовложения. Именно самовложение позволяет эффективно различать КС (нерегулярные) и регулярные языки. Как видно из второго примера, согласование скобок и т.п. требует самовложения, поэтому его нельзя специфицировать посредством регулярной грамматики.

С точки зрения разбора важно, что КС язык в состоянии распознавать **автомат магазинного типа**, эквивалентный конечному автомату, к которому добавлена память магазинного типа (стек). В функции автомата магазинного типа входит:

а) чтение входного символа, замещение верхнего символа стека строкой символов (возможно, пустой) и изменение состояния или

б) все то же самое, но без чтения входного символа.

Автомат магазинного типа можно представить кратным

$$(K, S, \Gamma, \delta, S_0, Z_0),$$

где K – конечное множество состояний, S – входной алфавит, Γ – алфавит магазинный, δ – множество переходов, S_0 – начальное состояние, Z_0 – символ магазина, который первоначально находится в стеке.

Рассмотрим, например, автомат магазинного типа M , определенный следующим образом:

$$K = \{A\}, \quad S_0 = \{A\},$$

$$S = \{ '(', ') ' \}, \quad Z_0 = I.$$

$$\Gamma = \{ O, I \},$$

$$\delta \text{ задается как } \delta(A, I, '(') = (A, IO)$$

(что означает: в состоянии A с I в вершине стека при чтении '(' перейти к состоянию A и заменить I на IO).

$$\delta(A, O, '(') = (A, OO), \quad \delta(A, O, ')') = (A, \varepsilon), \quad \delta(A, I, \varepsilon) = (A, \varepsilon).$$

Автомат M распознает согласуемые пары скобок. Открывающие скобки (представляемые как O) помещаются в стек и удаляются оттуда, когда встречается соответствующая закрывающая скобка. Строка скобок

принимается, если после считывания всей строки стек остается пустым. Это – обычный способ принятия строк автоматами магазинного типа, хотя можно также определить автоматы магазинного типа, которые принимают строки по конечному состоянию. Эти два типа эквивалентны.

Описанный выше автомат магазинного типа является детерминированным, т.е. для каждого допустимого входного символа имеется однозначный переход. Что же касается конечных автоматов, то мы можем также определять недетерминированные автоматы магазинного типа, содержащие множество переходов для заданного входа, состояния и содержания стека.

При разборе происходит эффективное моделирование соответствующего автомата магазинного типа. Некоторые КС языки не могут анализироваться детерминированным образом (т.е. без возврата). Язык, допускающий детерминированный анализ, называется **детерминированным**; большинство языков программирования являются детерминированными или, по крайней мере, почти таковыми.

Между КС-грамматиками и автоматами магазинного типа существует полное соответствие, и детерминированность автомата может зависеть от того, какая грамматика используется для генерирования языка. Метод разбора является детерминированным (для конкретной грамматики), если при разборе данной грамматики не требуется делать возврат. Некоторые языки можно разбирать детерминировано с помощью только одного из методов грамматического разбора. В частности, ряд языков можно разбирать детерминировано снизу вверх, но не сверху вниз. Далее мы будем рассматривать исключительно детерминированные методы разбора. Недетерминированные методы могут применяться к таким строчно-ориентированным языкам, как Фортран. Для языков же сильно рекурсивных (C++, Паскаль), где компилятор может быть вынужден возвращаться назад не только в текущей строке, но и в большой части программы, издержки, возникающие в случае возврата, просто неприемлемы. Другой недостаток возврата заключается в том, что он может

вызвать отмену действий компилятора, которые осуществляются параллельно с синтаксическим анализом.

Контрольные вопросы

1. Дайте определение контекстно-свободной грамматики.
2. Что такое нормальная форма Хомского?
3. Что такое нормальная форма Грейбаха?
4. Приведите пример самовложения грамматик.
5. Какой язык называется детерминированным?
6. Можно ли реализовать КС-грамматику в виде детерминированного конечного автомата?
7. Можно ли реализовать КС-грамматику в виде автомата магазинного типа?

8. LL(1) - ГРАММАТИКИ

LL(1)-грамматика – это грамматика такого типа, на основании которой можно получить детерминированный синтаксический анализатор, работающий по принципу сверху вниз. Прежде чем более точно определить LL(1)-грамматику, введем понятие s-грамматики.

S-грамматика представляет собой грамматику, в которой:

- 1) правая часть каждого порождающего правила начинается с терминала;
- 2) в тех случаях, когда в левой части более чем одного порождающего правила появляется один и тот же нетерминал, соответствующие правые части начинаются с разных терминалов.

Первое условие аналогично утверждению, что грамматика находится в нормальной форме Грейбаха, только за терминалом в начале каждой правой части правила могут следовать нетерминалы и/или терминалы.

Второе условие позволяет написать детерминированный нисходящий анализатор, так как при выводе предложения языка всегда можно сделать выбор между альтернативными порождающими правилами для самого левого нетерминала в сентенциальной форме, предварительно исследовав один следующий символ.

Грамматика с порождающими правилами

$$\begin{array}{ll}
 S \rightarrow pX & X \rightarrow x \\
 S \rightarrow qY & Y \rightarrow y \\
 X \rightarrow aXb & Y \rightarrow aYd
 \end{array}$$

представляет собой s-грамматику, тогда как следующая грамматика, которая генерирует тот же язык, не является ею:

$$\begin{array}{ll}
 S \rightarrow R & X \rightarrow aXb \\
 S \rightarrow T & X \rightarrow x \\
 R \rightarrow pX & Y \rightarrow aYd \\
 T \rightarrow qY & Y \rightarrow y
 \end{array}$$

поскольку правые части двух правил не начинаются с терминалов.

Определить, используется ли в качестве заданной грамматики s-грамматика, очень легко, и в некоторых случаях грамматику, которая не является s-грамматикой, можно преобразовать в нее, не затрагивая при этом генерируемый язык.

Рассмотрим проблему разбора строки *raaaxbbb* с помощью приведенной выше s-грамматики. Начав с символа *S*, попытаемся генерировать строку. Применим левосторонний вывод. Результат приводится ниже.

Исходная строка: *raaaxbbb*

Вывод: $S \rightarrow pX \rightarrow paXb \rightarrow paaXbb \rightarrow raaxXbbb \rightarrow raaaxbbb$

При выводе начальные терминалы в сентенциальной форме сверяются с символами в исходной строке. Там, где допускается замена самых левых терминалов в сентенциальной форме с помощью более чем одного порождающего правила, всегда можно выбрать соответствующее порождающее правило, исследовав следующий символ во входной строке. Это связано с тем, что, поскольку мы имеем дело с s-грамматикой, правые части альтернативных порождающих правил будут начинаться с различных терминалов. Таким образом, всегда можно написать детерминированный анализатор, осуществляющий разбор сверху вниз, для языка, генерируемого s-грамматикой.

LL(1)-грамматика является обобщением s-грамматики, и принцип ее обобщения все еще позволяет строить нисходящие детерминированные анализаторы. Две буквы L в LL(1) означают, что строки разбираются слева направо (Left) и используются самые левые выводы (Left), а цифра 1 – что варианты порождающих правил выбираются с помощью одного предварительно просмотренного символа. Кстати, если речь идет, например, о LL(2)-грамматике, это значит, что строки разбираются слева направо и используются самые левые выводы, а варианты порождающих правил выбираются с помощью двух предварительно просмотренных символов. Аналогично, термин LR(1)-грамматика подразумевает, что строки

разбираются слева направо (Left), используются самые правые выводы (Right), а варианты порождающих правил выбираются с помощью одного предварительно просмотренного символа. Более подробно LR-грамматики будут рассмотрены в параграфе, посвященном методам восходящего разбора.

Даже если правая часть порождающего правила и не начинается с терминала, каждый вариант какого-либо нетерминального символа может дать начало только тем строкам, которые начинаются с одного из конкретного множества терминалов. Например, на основании порождающих правил

$$S \rightarrow RY$$

$$R \rightarrow paXb$$

$$S \rightarrow TZ$$

$$T \rightarrow qaYd$$

можно вывести, что если для R и T нет других порождающих правил, порождающее правило $S \rightarrow RY$ желательно применять в разборе сверху вниз (нисходящий разбор – от начального символа – к строке) только когда предварительно просматриваемым символом является p . Аналогично порождающее правило $S \rightarrow TZ$ рекомендуется в тех случаях, когда таким символом окажется q .

Это приводит к идее множеств символов-предшественников, определяемых как

$$a \in S(A) \Leftrightarrow A \Rightarrow a\alpha,$$

где A – нетерминальный символ, α – строка терминалов и/или нетерминалов, возможно пустая, а $S(A)$ обозначает множество символов-предшественников A . В грамматике с порождающими правилами

$$P \rightarrow Ac$$

$$A \rightarrow aA$$

$$P \rightarrow Bd$$

$$B \rightarrow b$$

$$A \rightarrow a$$

$$B \rightarrow bB$$

a и b – символы-предшественники для P . Определим также множество символов-предшественников для строки терминалов и/или нетерминалов:

$$a \in S(\alpha) \Leftrightarrow \alpha \Rightarrow a\beta,$$

где α и β – строки терминалов и/или нетерминалов (β может быть пустой строкой).

Необходимым условием того, чтобы грамматика обладала признаком LL(1), является непересекаемость множеств символов-предшественников для альтернативных правых сторон порождающих правил.

Следует проявлять осторожность в тех случаях, когда нетерминал в начале правой части может генерировать пустую строку. Например, в грамматике

$$\begin{array}{ll} P \rightarrow AB & A \rightarrow \varepsilon \\ P \rightarrow BG & B \rightarrow bB \\ A \rightarrow aA & B \rightarrow c \end{array}$$

имеем $S(AB) = \{a, b, c\}$, причем b и c входят в множество, т.к. A может генерировать пустую строку; $S(BG) = \{b, c\}$ – множества пересекаются, следовательно грамматика не будет LL(1).

Рассмотрим также грамматику с порождающими правилами

$$\begin{array}{ll} T \rightarrow AB & Q \rightarrow \varepsilon \\ A \rightarrow PQ & B \rightarrow bB \\ A \rightarrow BC & B \rightarrow e \\ P \rightarrow pP & C \rightarrow cC \\ P \rightarrow \varepsilon & C \rightarrow f \\ Q \rightarrow qQ & \end{array}$$

для которой $S(PQ) = \{p, q\}$ и $S(BC) = \{b, e\}$

Однако так как PQ может генерировать пустую строку, следующим просматриваемым символом при применении порождающего правила $A \rightarrow PQ$ может быть b или e (вероятные символы, следующие за A), и одного следующего просматриваемого символа недостаточно, чтобы различить две альтернативные правые части для A (b и e являются также символами предшественниками для BC).

Поэтому вводится понятие направляющих символов, которые определяются так:

если A – нетерминал, то его направляющими символами будут

$S(A)$ + (все символы, следующие за A , если A может генерировать пустую строку).

Иными словами, это множество всех символов-предшественников + все символы, следующие за A , если A может генерировать пустую строку. В общем случае для заданного варианта α нетерминала P ($P \rightarrow \alpha$) имеем:

$$DS(P, \alpha) = \{a \mid a \in S(\alpha) \text{ или } (\alpha \Rightarrow \varepsilon \text{ и } a \in F(P))\}$$

где $F(P)$ есть множество символов, которые могут следовать за P . Так, в приведенном выше примере направляющие символы – это символы

$$DS(A, PQ) = \{p, q, b, e\}$$

$$DS(A, BC) = \{b, e\}$$

Поскольку указанные множества пересекаются, данная грамматика не может служить основой для детерминированного нисходящего анализатора, использующего один предварительно просматриваемый символ для различения альтернативных правых частей.

Уточним определение LL(1)-грамматики. Грамматику называют **LL(1)-грамматикой**, если для каждого нетерминала, появляющегося в левой части более одного порождающего правила, множества направляющих символов, соответствующих правым частям альтернативных порождающих правил – непересекающиеся. Все LL(1)-грамматики можно разбирать детерминировано сверху вниз.

Контрольные вопросы

1. Дайте определение LL(1)-грамматики.
2. Какой тип разбора (восходящий или нисходящий) подразумевает LL(1)-грамматика.
3. Что такое направляющий символ?
4. Является ли приведенная ниже грамматика (S - начальный символ) LL(1)-грамматикой? Обосновать ответ.

$$S \rightarrow - P / P$$

$$P \rightarrow (S) / o / P B P$$

$$B \rightarrow + / - / * / /$$

9. ПРЕОБРАЗОВАНИЕ ГРАММАТИК В LL(1) ФОРМУ

"Очевидной" грамматикой для большинства языков программирования является не LL(1)-грамматика. Однако обычно очень большое число контекстно-свободных средств языка программирования можно представить с помощью LL(1)-грамматики. Проблема заключается в том, чтобы, имея грамматику, которая не обладает признаком LL(1), найти эквивалентную ей LL(1)-грамматику. Не существует универсального алгоритма преобразования любой КС-грамматики в LL(1) форму (а также определения самой возможности такого преобразования). Однако существует ряд приемов, позволяющих выполнить такое преобразование во многих частных случаях.

Устранение левой рекурсии

Грамматика, содержащая левую рекурсию, не является LL(1)-грамматикой. Рассмотрим правила

$$A \rightarrow Aa \text{ (левая рекурсия в } A\text{)}$$

$$A \rightarrow a$$

Здесь a символ-предшественник для обоих вариантов нетерминала A . Аналогично грамматика, содержащая левый рекурсивный цикл, не может быть LL(1)-грамматикой, например

$$A \rightarrow BC$$

$$B \rightarrow CD$$

$$C \rightarrow AE$$

Грамматику, содержащую левый рекурсивный цикл, можно преобразовать в грамматику, содержащую только прямую левую рекурсию, и далее, за счет введения дополнительных нетерминалов, левую рекурсию можно исключить полностью (в действительности она заменяется правой рекурсией, которая не представляет проблемы в отношении LL(1)-свойства). В качестве примера рассмотрим грамматику с порождающими правилами

$$S \rightarrow Aa$$

$$C \rightarrow Dd$$

$$A \rightarrow Bb$$

$$C \rightarrow e$$

$$B \rightarrow Cc$$

$$D \rightarrow Az$$

которая имеет левый рекурсивный цикл, вовлекающий A, B, C, D . Чтобы заменить этот цикл прямой левой рекурсией, упорядочим нетерминалы следующим образом: S, A, B, C, D .

Рассмотрим все порождающие правила вида

$$X_i \rightarrow X_j \gamma,$$

где X_i и X_j – нетерминалы, а γ – строка терминальных и нетерминальных символов. В отношении правил, для которых $j \geq i$, никакие действия не производятся. Однако это неравенство не может выдерживаться для всех правил, если есть левый рекурсивный цикл. При выбранном нами порядке мы имеем дело с единственным правилом:

$$D \rightarrow Az$$

так как A предшествует D в этом упорядочении. Теперь начнем замещать A , пользуясь всеми правилами, имеющими A в левой части. В результате получаем

$$D \rightarrow Bbz$$

Поскольку B предшествует D в упорядочении, процесс повторяется, что дает правило:

$$D \rightarrow Ccbz$$

Затем он повторяется еще раз и дает два правила:

$$D \rightarrow ecbz$$

$$D \rightarrow Ddcbz$$

Теперь преобразованная грамматика выглядит следующим образом:

$$S \rightarrow Aa$$

$$C \rightarrow e$$

$$A \rightarrow Bb$$

$$D \rightarrow Ddcbz$$

$$B \rightarrow Cc$$

$$D \rightarrow ecbz$$

$$C \rightarrow Dd$$

Все эти порождающие правила имеют требуемый вид, а левый рекурсивный цикл заменен прямой левой рекурсией. Чтобы исключить прямую левую рекурсию, введем новый нетерминальный символ Z и заменим правила

$$D \rightarrow ecbz$$

$$D \rightarrow Ddcbz$$

на

$$D \rightarrow ecbz$$

$$Z \rightarrow dcbz$$

$$D \rightarrow ecbzZ$$

$$Z \rightarrow dcbzZ$$

Заметим, что до и после преобразования D генерирует регулярное выражение

$$(ecbz)(dcbz)^*$$

Обобщая, можно показать, что если нетерминал A появляется в левых частях $r + s$ порождающих правил, r из которых используют прямую левую рекурсию, а s – нет, т.е.

$$A \rightarrow A\alpha_1, A \rightarrow A\alpha_2, \dots, A \rightarrow A\alpha_r$$

$$A \rightarrow \beta_1, A \rightarrow \beta_2, \dots, A \rightarrow \beta_s$$

то эти правила можно заменить на следующие:

$$\left. \begin{array}{l} A \rightarrow \beta_i \\ A \rightarrow \beta_i Z \end{array} \right\} 1 \leq i \leq s \quad \left. \begin{array}{l} Z \rightarrow \alpha_i \\ Z \rightarrow \alpha_i Z \end{array} \right\} 1 \leq i \leq r$$

Неформальное доказательство заключается в том, что до и после преобразования A генерирует регулярное выражение

$$(\beta_1 | \beta_2 | \dots | \beta_s)(\alpha_1 | \alpha_2 | \dots | \alpha_r)^*$$

Следует обратить внимание, что устранив левую рекурсию (или левый рекурсивный цикл), мы еще не получаем LL(1)-грамматику, т.к. для некоторых нетерминалов в левой части правил полученных грамматик существуют альтернативные правые части, начинающиеся с одних и тех же символов. Поэтому после устранения левой рекурсии следует продолжить преобразование грамматики к LL(1) виду.

Факторизация

Во многих ситуациях грамматики, не обладающие признаком LL(1), можно преобразовать в LL(1)-грамматики с помощью процесса факторизации. Рассмотрим пример такой ситуации.

$$\begin{array}{ll}
 P \rightarrow \text{begin } D ; C \text{ end} & C \rightarrow s ; C \\
 D \rightarrow d , D & C \rightarrow s \\
 D \rightarrow d &
 \end{array}$$

В процессе факторизации мы заменяем несколько правил для одного нетерминала в левой части, правая часть которых начинается с одного и того же символа (цепочки символов) на одно правило, где в правой части за общим началом следует дополнительно вводимый нетерминал. Также грамматика дополняется правилами для дополнительного нетерминала, согласно которым из него выводятся различные «остатки» первоначальной правой части правила. Для приведенной выше грамматики это даст следующую LL(1)-грамматику:

$$\begin{array}{ll}
 P \rightarrow \text{begin } D ; C \text{ end} & \\
 D \rightarrow d X & \text{(вводим дополнительный нетерминал } X) \\
 X \rightarrow , D & \text{(по 1-му правилу для } D \text{ исходной грамматики за } d \text{ следует } , D) \\
 X \rightarrow \varepsilon & \text{(по 2-му правилу для } D \text{ исходной грамматики за } d \text{ ничего нет} \\
 & \text{(пустая строка))} \\
 C \rightarrow s Y & \text{(вводим дополнительный нетерминал } Y) \\
 Y \rightarrow ; C & \text{(по 1-му правилу для } C \text{ исходной грамматики за } s \text{ следует } ; C) \\
 Y \rightarrow \varepsilon & \text{(по 2-му правилу для } C \text{ исходной грамматики за } s \text{ ничего нет} \\
 & \text{(пустая строка))}
 \end{array}$$

Аналогичным образом, порождающие правила

$$\begin{array}{ll}
 S \rightarrow aSb & S \rightarrow \varepsilon \\
 S \rightarrow aSc &
 \end{array}$$

можно преобразовать путем факторизации в правила

$$S \rightarrow aSX$$

$$X \rightarrow b$$

$$S \rightarrow \varepsilon$$

$$X \rightarrow c$$

и полученной в результате грамматики будет LL(1).

Процесс факторизации, однако, нельзя автоматизировать, распространив его на общий случай. Следующий пример показывает, что может произойти. Рассмотрим правила

$$1. P \rightarrow Qx$$

$$4. Q \rightarrow q$$

$$2. P \rightarrow Ry$$

$$5. R \rightarrow sRn$$

$$3. Q \rightarrow sQm$$

$$6. R \rightarrow r$$

Оба множества направляющих символов для двух вариантов P содержат s , и, пытаясь "вынести s за скобки", мы замещаем Q и R в правых частях правил 1 и 2:

$$P \rightarrow sQmx$$

$$P \rightarrow qx$$

$$P \rightarrow sRny$$

$$P \rightarrow ry$$

Эти правила можно заменить следующими:

$$P \rightarrow qx$$

$$P_1 \rightarrow Qmx$$

$$P \rightarrow ry$$

$$P_1 \rightarrow Rny$$

$$P \rightarrow sP_1$$

Правила для P_1 аналогичны первоначальным правилам для P и имеют пересекающиеся множества направляющих символов. Мы можем преобразовать эти правила так же, как и правила для P :

$$P_1 \rightarrow sQmxx$$

$$P_1 \rightarrow sRnny$$

$$P_1 \rightarrow qmxx$$

$$P_1 \rightarrow rnny$$

Факторизуя, получаем

$$P_1 \rightarrow qmxx$$

$$P_2 \rightarrow Qmxx$$

$$P_1 \rightarrow rnny$$

$$P_2 \rightarrow Rnny$$

$$P_1 \rightarrow sP_2$$

Правила для P_2 аналогично правилам для P_1 и P , но длиннее их, и теперь уже очевидно, что этот процесс бесконечный. Таким образом, не

всегда факторизация позволяет осуществить необходимое преобразование, некоторые грамматики вообще невозможно преобразовать в LL(1)-форму.

Контрольные вопросы

1. Почему левая рекурсия является препятствием для LL(1)-разбора?
2. В чем заключается процесс факторизации правил грамматики?
3. Является ли приведенная ниже грамматика (S - начальный символ) LL(1)-грамматикой? Обосновать ответ. Если не является – преобразовать к LL(1) виду.

$$S \rightarrow D V$$

$$D \rightarrow i / i, D$$

$$V \rightarrow \{ O \}$$

$$O \rightarrow p / O, p$$

10. ПОСТРОЕНИЕ СИНТАКСИЧЕСКОГО ГРАФА

Для построения синтаксического анализатора можно использовать два различных метода. Один из них – это написать универсальную программу грамматического разбора, пригодную для всех возможных грамматик заданного класса. В этом случае конкретные грамматики задаются этой программе в виде данных некоторой структуры, которая управляет ее работой. Поэтому такая программа называется таблично-управляемой.

Другой метод – это разработка программы грамматического разбора специально для заданного конкретного языка; при этом его синтаксис по определенным правилам отображается в последовательность операторов, т.е. в программу. Такую реализацию разбора будем называть программно-управляемой.

Каждый из этих методов имеет свои преимущества и недостатки. При построении транслятора для конкретного языка программирования вряд ли потребуется высокая гибкость и параметризация, свойственные универсальной программе. Программа грамматического разбора, предназначенная специально для данного языка, обычно оказывается более эффективной и с ней легче работать, легче встроить выполнение параллельно с синтаксическим разбором различных дополнительных действий, например, проверки соответствия типов данных и пр., что может оказаться особенно важно при реализации специализированных языков САПР. Использование универсальной программы позволяет разработать синтаксический анализатор максимально быстро, с меньшими требованиями к квалификации разработчика (в идеале, он должен только представить в требуемой форме и ввести грамматику реализуемого языка). В любом случае полезно представлять заданный синтаксис в виде так называемого синтаксического графа (синтаксической диаграммы, графа распознавания). Такой граф отражает управление ходом работы при грамматическом анализе предложения.

Для нисходящего грамматического разбора характерно, что цель анализа известна с самого начала. Эта цель – распознать предложение, т.е. последовательность символов, которая может порождаться из начального символа. Применение порождающего правила, т.е. замена одного символа последовательностью символов, соответствует расщеплению одной цели на некоторое число подцелей, которые должны следовать в определенном порядке. Поэтому нисходящий метод можно называть также **целеориентированным грамматическим разбором**. При построении программы грамматического разбора можно воспользоваться этим очевидным соответствием между нетерминальными символами и целями: для каждого нетерминального символа строится своя процедура грамматического разбора. Цель каждой такой процедуры – распознавание части предложения, которая может порождаться из соответствующего нетерминального символа. Поскольку мы хотим построить граф, представляющий всю программу грамматического разбора, то каждый нетерминальный символ будет отражаться в подграф.

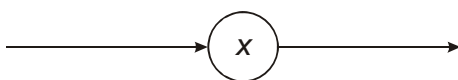
Правила построения синтаксического графа:

A1. Каждый нетерминальный символ A с соответствующим множеством порождающих правил

$$A ::= \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

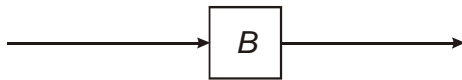
отображается в синтаксический граф A , структура которого определяется правой частью порождающего правила в соответствии с **A2-A6**.

A2. Каждое появление **терминального** символа x в β_i соответствует оператору распознавания этого символа во входном предложении. На графе это изображается ребром, помеченным символом x , заключенным в кружок или овал:

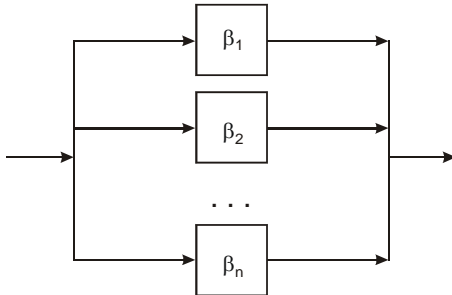



A3. Каждому появлению **нетерминального** символа B в β_i соответствует обращение к процедуре распознавания B . На графе это

изображается ребром, помеченным символом B , заключенным в прямоугольник:

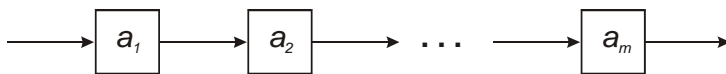



А4. Порождающее правило, имеющее вид $A ::= \beta_1 / \beta_2 / \dots / \beta_n$



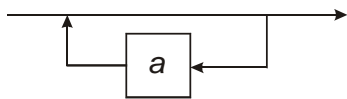
отображается в граф, где каждое  получено применением правил **А2-А6** к β_i .

А5. Строка β , имеющая вид $\beta = \alpha_1 \alpha_2 \dots \alpha_m$ отображается в граф



где каждое  получено применением правил **А2-А6** к α_i .

А6. Строка β , имеющая вид $\beta = \{\alpha\}^*$ отображается в граф



где  получено применением правил **А2-А6** к α .

Пример.

$A ::= x / (B),$

$B ::= AC$

$C ::= \{+A\}^*$

Здесь "+", "x", "(" и ")" – терминальные символы, а "{" и "}" являются метасимволами. Язык, порождаемый из A , состоит из выражений с операндами x , знаком операции "+" и скобками.

Примеры предложений:

x (x) $(x+x)$ $((x))$

Графы, полученные с помощью применения шести правил построения графов, показаны на рис.10.1. Заметим, что эту систему графов можно свести в один граф, подставив соответственно C в B и B в A (см. рис.10.2).

Синтаксический граф является эквивалентным представлением грамматики языка; его можно использовать вместо множества порождающих правил БНФ. Это очень удобная форма, и во многих (если не в большинстве) случаев она предпочтительнее БНФ.

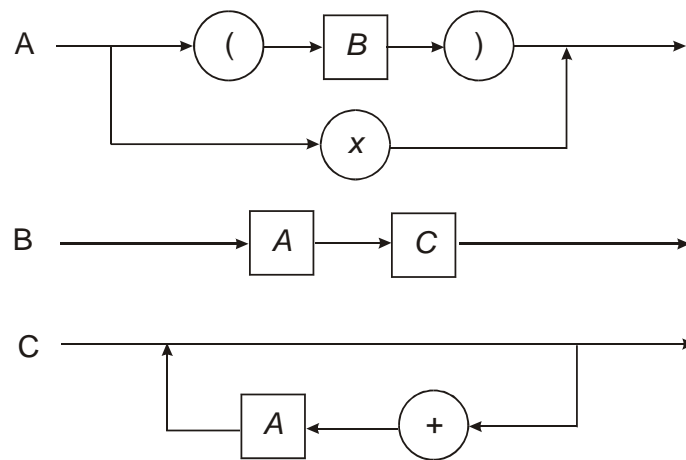


Рис. 10.1. Синтаксические графы.

Граф является подходящим представлением, которое может служить отправной точкой для разработчика языка.

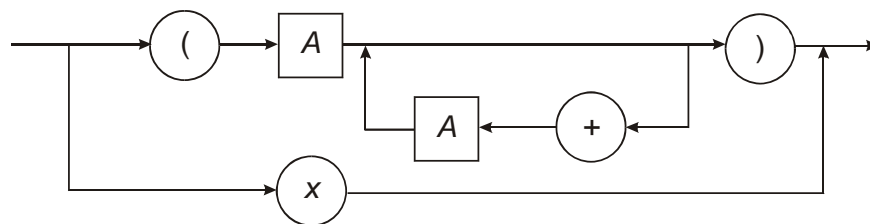


Рис. 10.2. Сводный синтаксический граф.

Граф дает ясное и точное представление о структуре языка, а также позволяет лучше представить себе процесс грамматического разбора.

Для того чтобы обеспечить детерминированный грамматический разбор с просмотром вперед на один символ, были установлены LL(1)-

ограничения, при графическом представлении синтаксиса они проявляются следующим образом.

1. При каждом разветвлении можно выбрать ветвь, по которой будет идти дальнейший разбор по очередному символу на этой ветви. Это означает, что никакие две ветви не должны начинаться с одного и того же символа.

2. Если какой-либо граф A можно пройти, не читая вообще никаких входных символов, то такая "нулевая ветвь" должна помечаться всеми символами, которые могут следовать за A . (Это влияет на решение о переходе на эту ветвь).

Контрольные вопросы

1. Назовите преимущества и недостатки таблично-управляемых и программно-управляемых синтаксических анализаторов.
2. Какой метод разбора можно также назвать целеориентированным?

11. ПОСТРОЕНИЕ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА ДЛЯ ЗАДАННОГО СИНТАКСИСА

Программу, которая распознает какой-либо язык, легко построить на основе его детерминированного синтаксического графа (если таковой существует). Этот граф фактически представляет собой блок-схему программы, при ее разработке рекомендуется строго следовать правилам преобразования, подобным тем, с помощью которых можно предварительно получить из БНФ графическое представление синтаксиса. Применение данных правил предполагает наличие основной программы, содержащей процедуры, которые соответствуют различным подцелям, а также процедуру перехода к очередному символу.

Для простоты мы будем считать, что предложение, которое нужно анализировать, представлено входным файлом `input` и что терминальные символы – отдельные значения типа `char`. Пусть символьная переменная `char ch` всегда содержит очередной читаемый символ. Тогда переход к следующему символу выражается оператором:

```
ch = fgetc(input);
```

Следует отметить, что функция `char fgetc(FILE *fp)` является стандартной функцией языка Си для чтения символа из файла и для ее использования необходимо в начале программы подключить соответствующий заголовочный файл директивой `#include <stdio.h>`

Основная программа будет состоять из оператора чтения первого символа, за которым следует оператор активации основной цели грамматического разбора. Отдельные процедуры, соответствующие целям грамматического разбора или графам, получаются по следующим правилам. Пусть оператор, полученный с помощью преобразования графа S , обозначается через $T(S)$.

Правила преобразования графа в программу:

В1. Свести систему графов к как можно меньшему числу отдельных графов с помощью соответствующих подстановок.

В2. Преобразовать каждый граф в описание процедуры в соответствии с приведенными ниже правилами **В3-В7**.

В3. Последовательность элементов

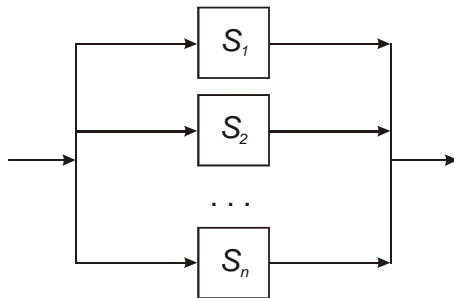


переводится в составной оператор

```

{
  T(S1 );
  T(S2 );
  ... ;
  T(Sn )
}
  
```

В4. Выбор элементов

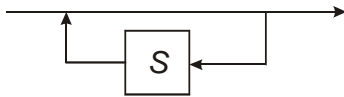


переводится в условный оператор

```

if(belongsto(ch, L1)) T(S1);
  else if(belongsto(ch, L2)) T(S2);
    else ...
      if(belongsto(ch, Ln)) T(Sn);
        else error();
  
```

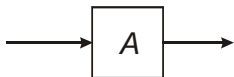
где L_i означает множество начальных символов конструкции S_i ($L_i = first(S_i)$), а функция $belongsto(ch, L_i)$ возвращает истинное значение, если символ ch принадлежит соответствующему множеству начальных символов L_i и ложное значение в противном случае.

В5. Цикл вида

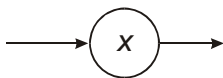
переводится в оператор

```
while(belongsTo(ch, L)) T(S);
```

где $T(S)$ есть отображение S в соответствии с правилами **В3-В7**, а L есть множество $L = \text{first}(S)$.

В6. Элемента графа, обозначающий другой граф A

переводится в оператор обращения к функции $A()$.

В7. Элемент графа, обозначающий терминальный символ

переводится в оператор

```
if(ch == 'x') ch = fgetc(input);
else error();
```

где `error()` – функция, к которой обращаются при появлении неправильной конструкции.

Теперь покажем применение этих правил на примере преобразования редуцированного графа (см. рис.10.2) в программу грамматического разбора.

```
char ch;
void A()
{
    if(ch == 'x') ch = fgetc(input);
    else
        if(ch == '(')
        {
            ch = fgetc(input);
            A();
            while(ch == '+')
            {
                ch = fgetc(input);
```

```

        A();
    }
    if(ch == ')') ch = fgetc(input);
    else error();
}
else error();
}

void main(int argc, char **argv)
{
    ch = fgetc(input);
    A();
}

```

При этом преобразовании свободно применялись некоторые очевидные правила программирования, позволяющие упростить программу. Например, при буквальном переводе четвертая строка имела бы вид:

```

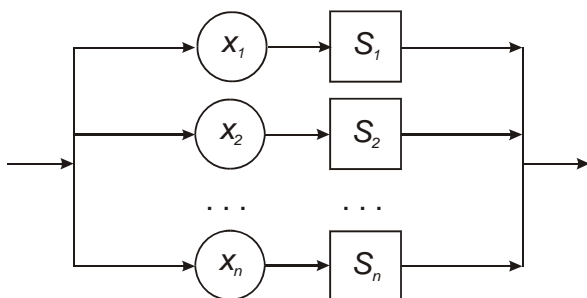
if(ch == 'x')
    if(ch == 'x') ch = fgetc(input); else error();
else ...

```

Ясно, что ее можно сократить, как это сделано в программе. Операторы чтения в восьмой и двенадцатой строках тоже получены с помощью такого же упрощения.

По-видимому, полезно определить, когда вообще возможны подобные упрощения, и показать это непосредственно в виде графов. Два основных случая покрываются следующими дополнительными правилами:

В4а.



```

if(ch == 'x1')
{
    ch = fgetc(input);
    T(S1);
}
else
if(ch == 'x2')

```

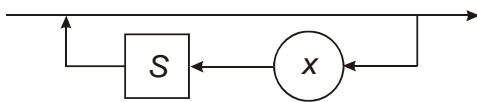


```

{
    ch = fgetc(input);
    T(S2);
}
else
. . .
if(ch = 'xn')
{
    ch = fgetc(input);
    T(Sn);
}
else error();

```

B5a.



```

while(ch == 'x')
{
    ch = fgetc(input); T(S);
}

```

Кроме того, часто встречающуюся конструкцию

```

ch = fgetc(input);
T(S);
while(W)
{
    ch = fgetc(input);
    T(S);
}

```

можно, разумеется, выразить короче:

```

do {
    ch = fgetc(input);
    T(S);
} while(W);

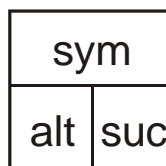
```

Мы намеренно не описываем пока функцию `error` ("ошибка"). Поскольку сейчас нас интересует лишь, как определить, правильно ли входное предложение, мы можем считать, что эта процедура заканчивает работу программы. На практике в случае появления неправильных конструкций нужно организовать корректную их диагностику, что будет рассмотрено далее.

12. ПОСТРОЕНИЕ ТАБЛИЧНО-УПРАВЛЯЕМОЙ ПРОГРАММЫ ГРАММАТИЧЕСКОГО РАЗБОРА

Конкретные грамматики задаются таблично-управляемой универсальной программе в виде исходных данных, предшествующих предложениям, которые нужно разобрать. Универсальная программа работает в строгом соответствии с методом простого нисходящего грамматического разбора; поэтому она довольно проста, если основана на детерминированном синтаксическом графе, т.е. если предложения можно анализировать с просмотром вперед на один символ без возврата.

В данном случае грамматика (предположим, что она представлена в виде детерминированного множества синтаксических графов) преобразуется в подходящую структуру данных, а не в структуру программы. Естественный способ представить граф – это ввести узел для каждого символа и связать эти узлы с помощью ссылок. Следовательно, "таблица" – это не просто массив. Узлы этой структуры представляют собой структуры (struct) с объединениями (union). Объединение позволяет идентифицировать тип узла. Первый идентифицируется терминальным символом, который он обозначает (tsym), второй – ссылкой на структуру данных, представляющую соответствующий нетерминальный символ (nsym). Оба варианта объединения содержат две ссылки: одна указывает на следующий символ, **последователь** (suc), а другая связана со списком возможных **альтернатив** (alt). Графически узел можно изобразить следующим образом



Также нужен элемент, представляющий пустую последовательность, символ "пусто". Обозначим его с помощью терминального элемента, называемого empty.

```
struct node;
typedef node *pointer;
struct node {
```

```

pointer suc;
pointer alt;
int isTerminal;
union {
    char tsym;
    hpointer nsym;
};
};

```

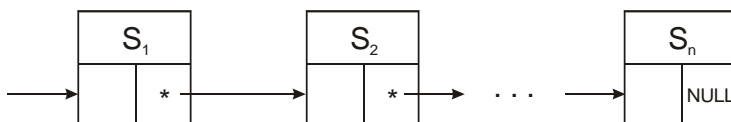
Правила преобразования графов в структуре данных аналогичны правилам **B1-B7**.

Правила преобразования графов в структурах данных:

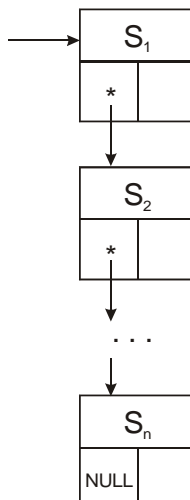
C1. Свести систему графов к как можно меньшему числу отдельных графов с помощью соответствующих подстановок.

C2. Преобразовать каждый граф в структуру данных согласно правилам **C3-C5**, приведенным ниже.

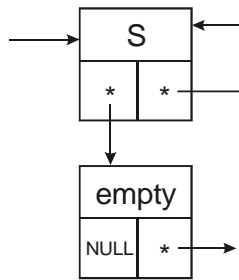
C3. Последовательность элементов (см. рис. к правилу **B3**) преобразуется в следующий список узлов:



C4. Список альтернатив (см. рис. к правилу **B4**) преобразуется в следующую структуру данных:



С5. Цикл (см. рис. к правилу **B5**) преобразуется в следующую структуру:



В качестве примера рассмотрим построение структуры данных для сводного синтаксического графа, приведенного выше на рис.10.2.

Структура данных идентифицируется **узлом-заголовком**, который содержит имя нетерминального символа (цели), к которому относится структура. Вообще говоря, заголовок не является необходимым, так как можно вместо поля цели указывать непосредственно на "вход" в соответствующую структуру. Однако заголовок можно использовать для хранения выводимого на печать имени структуры:

```
struct header;
typedef header *hpointer;
struct header {
    pointer entry;
    char sym;
};
```

Программа, производящая грамматический разбор предложения, представленного в виде последовательности символов входного файла, состоит из повторяющегося оператора, описывающего переход от одного узла к следующему узлу.

В поле `sym` для терминального символа помещается сам символ, для нетерминального – ссылка на соответствующую структуру данных. Она оформлена как процедура, задающая интерпретацию графа; если встречается узел, представляющий нетерминальный символ, то интерпретация графа, на который ссылается данный узел, предшествует завершению интерпретации текущего графа. Следовательно, процедура интерпретации вызывается

рекурсивно. Если текущий символ (*sym*) входного файла совпадает с символом в текущем узле структуры данных, то процедура переходит к узлу, на который указывает поле *suc*, иначе – к узлу, на который указывает поле *alt*. Полученная структура данных приведена на рис.12.1.

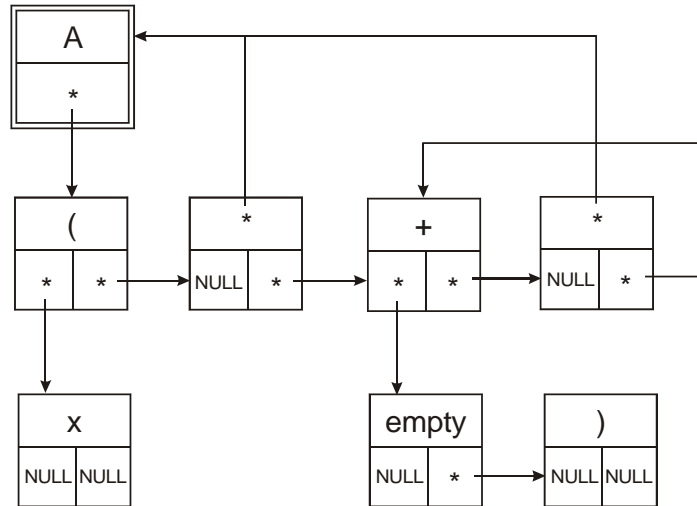


Рис. 12.1. Представление синтаксического графа в виде структуры данных.

```

void parse(hpointer goal, int &match)
{
    pointer s;
    s = goal->entry;
    do
    {
        if(s->isTerminal)
        {
            if(s->tsym == sym)
            {
                match = 1;
                sym = fgetc(input);
            }
            else
                match = (s->tsym == empty)? 1 : 0;
        }
        else
            parse(s->nsym, match);
        if(match)
            s = s->suc;
        else
            s = s->alt;
    } while(s != NULL);
}

```

Программа грамматического разбора, приведенная выше, "стремится" к новой подцели G , как только она появляется, не проверяя даже, содержится ли текущий символ входного файла во множестве начальных символов соответствующего графа $first(G)$. Это предполагает, что в синтаксическом графе не должно существовать выбора между несколькими альтернативными нетерминальными элементами. В частности, если какой-либо нетерминальный символ может порождать пустую последовательность, то ни одна из правых частей соответствующих ему порождающих правил не должна начинаться с нетерминального символа.

При такой организации грамматического разбора программа обычно считывает представленную в определенном формате грамматику языка, заполняя при этом соответствующие структуры данных, и только после этого читает текст на разбираемом языке и выполняет его синтаксический анализ.

На основе этой программы грамматического разбора можно построить более сложные таблично-управляемые программы грамматического разбора, которые могут работать с более широкими классами грамматик. Небольшая модификация позволяет также осуществлять и возвраты, но это будет сопровождаться значительной потерей эффективности.

13. ВОСХОДЯЩИЙ СИНТАКСИЧЕСКИЙ АНАЛИЗ

В этом разделе будет рассмотрен основной метод восходящего синтаксического анализа, известный как синтаксический анализ типа "перенос/свертка" (shift-reduce) и называемый далее сокращенно ПС-анализом.

ПС-анализ пытается строить дерево разбора для входной строки, начиная с листьев (снизу) и работая по направлению к корню дерева (вверх). Этот процесс можно рассматривать как свертку строки w к стартовому символу грамматики. На каждом шаге свертки (reduction step) некоторая подстрока, соответствующая правой части продукции, заменяется символом из левой части этой продукции, и если на каждом шаге подстроки выбираются корректно, то мы получаем обращенное правое порождение.

Пример:

Рассмотрим грамматику

$$S \rightarrow aABe$$

$$A \rightarrow Abc / b$$

$$B \rightarrow d$$

Предложение $abbcde$ сводится к S с помощью следующих шагов:

$abbcde$

$aAbcde$

$aA.de$

$aABe$

S

Мы сканируем строку $abbcde$ в поисках подстроки, соответствующей правой части какой-либо продукции. Такими подстроками являются b и d . Выберем крайнее слева b и заменим его нетерминалом A , который представляет собой левую часть продукции $A \rightarrow b$; таким образом, получим строку $aAbcde$. Теперь правым частям продукций соответствуют подстроки Abc , b и d . Хотя b и является крайней слева подстрокой, соответствующей правой части одной из продукций, выберем для замены подстроку Abc и

заменим ее нетерминалом A в соответствии с продукцией $A \rightarrow Abs$. В результате получим строку $aAde$. Заменяя d на B , левую часть продукции $B \rightarrow d$, получаем $aABe$, которая в соответствии с первой продукцией заменяется стартовым символом S . Итак, последовательность из четырех сверток позволяет привести строку $abbcd$ к стартовому символу S . Эти сокращения представляют собой обращенное (т.е. записанное в обратном порядке) правое приведение $S \xRightarrow{rm} aABe \xRightarrow{rm} aAde \xRightarrow{rm} aAbcde \xRightarrow{rm} abbcd$.

Основы

Неформально говоря, основа, или дескриптор (handle) строки – это подстрока, которая совпадает с правой частью продукции и свертка которой в левую часть продукции представляет собой один шаг обращенного правого порождения. Во многих случаях крайняя слева подстрока β соответствующая правой части некоторой продукции $A \rightarrow \beta$ не является основой, поскольку свертка в соответствии с продукцией $A \rightarrow \beta$ приводит к строке, которая не может быть свернута к стартовому символу. Если в предыдущем примере мы заменим во второй строке $aAbcde$ символ b нетерминалом A , то получим строку $aAAcde$, которая не может быть свернута в S . По этой причине нам следует дать более точное определение основы.

Говоря формально, **основа правосентенциальной формы** γ является продукцией $A \rightarrow \beta$ и позицией строки β в γ , такими, что β может быть заменена нетерминалом A для получения предыдущей правосентенциальной формы в правом порождении γ . Таким образом, если $S \xRightarrow{rm} aAw \xRightarrow{rm} a\beta w$, то $A \rightarrow \beta$ в позиции после a представляет собой основу строки $a\beta w$. Строка w справа от основы содержит только терминальные символы. Заметим, что грамматика может быть неоднозначной, с несколькими правыми порождениями $a\beta w$. Если грамматика однозначна, то каждая правосентенциальная форма грамматики имеет ровно одну основу.

В приведенном выше примере $abbcd e$ представляет собой правосенденциальную форму, основой которой является $A \rightarrow \beta$ в позиции 2. Аналогично $aAbcde$ представляет собой правосенденциальную форму, дескриптор которой – $A \rightarrow A\beta c$ в позиции 2. Иногда мы будем говорить "подстрока β представляет собой основу $\alpha\beta w$ ", если позиция β и продукция $A \rightarrow \beta$ определяются однозначно.

На рис.13.1 изображена основа $A \rightarrow \beta$ в дереве разбора правосенденциальной формы $\alpha\beta w$. Основа представляет крайнее слева завершенное поддерево, состоящее из узла и всех его потомков. На рис.13.1 узел A — нижний крайний слева внутренний узел, все потомки которого находятся в дереве. Свертку β к A в $\alpha\beta w$ можно представить как "обрезку основы", т.е. удаление из дерева разбора всех потомков A .

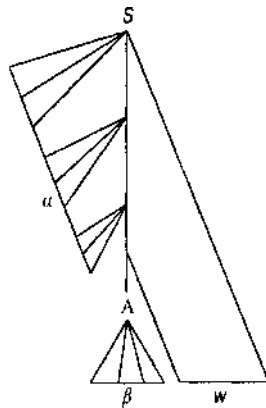


Рис.13.1. Основа $A \rightarrow \beta$ в дереве разбора $\alpha\beta w$

Пример 13.а

Рассмотрим следующую грамматику

$$(1) E \rightarrow E + E$$

$$(2) E \rightarrow E * E$$

$$(3) E \rightarrow (E)$$

$$(4) E \rightarrow \text{id}$$

(13.1)

и правое порождение

$$\begin{aligned}
& E \Rightarrow \underline{E + E} \\
& \Rightarrow \underline{E + E * E} \\
& \Rightarrow \underline{E + E * id_3} \\
& \Rightarrow \underline{E + id_2 * id_3} \\
& \Rightarrow \underline{id_1 + id_2 * id_3}
\end{aligned}$$

Для удобства мы пометили подстрочными индексами **id** и подчеркнули основу каждой правосентенциальной формы. Например, **id₁** представляет собой основу право-сентенциальной формы **id₁+id₂*id₃**, поскольку **id** является правой частью продукции $E \rightarrow id$, и замена **id₁** на E приведет к предыдущей правосентенциальной форме $E+id_2*id_3$. Обратите внимание на то, что строка справа от основы состоит только из терминальных символов.

Поскольку грамматика (13.1) неоднозначна, имеется еще одно правое порождение той же строки:

$$\begin{aligned}
& E \Rightarrow \underline{E * E} \\
& \Rightarrow \underline{E * id_3} \\
& \Rightarrow \underline{E + E} * id_3 \\
& \Rightarrow \underline{E + id_2} * id_3 \\
& \Rightarrow \underline{id_1 + id_2} * id_3
\end{aligned}$$

Рассмотрим правосентенциальную форму $E+E*id_3$. В этом порождении $E+E$ – основа $E+E*id_3$, в то время как в ранее представленном порождении ее основой является id_3 .

Первое порождение дает оператору $*$ больший приоритет, чем оператору $+$, в то время как во втором порождении выше приоритет оператора $+$.

Обрезка основ

Обращенное правое порождение может быть получено посредством "**обрезки основ**". Мы начинаем процесс со строки терминалов w , которую хотим проанализировать. Если w – предложение рассматриваемой

грамматики, то $w = \gamma_n$, где γ_n – n -я правосенденциальная форма некоторого, еще неизвестного правого порождения

$$S \xRightarrow{rm} \gamma_0 \xRightarrow{rm} \gamma_1 \xRightarrow{rm} \gamma_2 \xRightarrow{rm} \dots \xRightarrow{rm} \gamma_{n-1} \xRightarrow{rm} \gamma_n = w$$

Для воссоздания этого порождения в обратном порядке мы находим основу β_n в γ_n и заменяем ее левой частью продукции $A_n \rightarrow \beta_n$, для получения $(n-1)$ -й сенденциальной формы γ_{n-1} . Заметим, что пока мы не знаем, каким образом искать основы, но вскоре познакомимся с соответствующими методами.

Затем мы повторяем описанный процесс, т.е. находим в γ_{n-1} основу β_{n-1} и свертываем ее для получения правосенденциальной формы γ_{n-2} . Если после очередного шага правосенденциальная форма содержит только стартовый символ S , мы прекращаем процесс и сообщаем об успешном завершении анализа. Обращенная последовательность продукций, использованных в свертках, представляет собой правое порождение входной строки.

Пример 13.b

Рассмотрим грамматику (13.1) из примера 13.a и входную строку **id₁+id₂*id₃**. Последовательность сверток, приводящая входную строку к стартовому символу E , показана в таблице 13.1. Следует отметить, что последовательность правосенденциальных форм в этом примере представляет собой обращение первой последовательности правых порождений из примера 13.a.

Таблица 13.1. Свертки, выполняемые ПС-анализатором

правосенденциальная форма	основа	сворачивающая продукция
id ₁ +id ₂ *id ₃	id ₁	$E \rightarrow id$
E +id ₂ *id ₃	id ₂	$E \rightarrow id$
E + E *id ₃	id ₃	$E \rightarrow id$
E + E * E	E * E	$E \rightarrow E * E$
E + E	E + E	$E \rightarrow E + E$
E		

Стековая реализация ПС-анализа

Существует две проблемы при синтаксическом анализе методом обрезки основ. Первая заключается в обнаружении подстроки для свертки в правосентенциальной форме, а вторая – в определении, какая именно продукция должна быть выбрана, если имеется несколько продукций с соответствующей подстрокой в правой части. Перед тем как ответить на эти вопросы, сначала рассмотрим структуры данных, используемые в ПС-анализаторе.

Достаточно удобный путь реализации ПС-анализатора состоит в использовании стека для хранения символов грамматики и входного буфера для хранения анализируемой строки. В качестве маркера дна стека мы используем \$, и этот же символ является маркером правого конца входной строки. Изначально стек пуст, а входной буфер содержит строку w :

Стек	Вход
\$	w \$

Синтаксический анализатор работает путем переноса нуля или нескольких символов в стек до тех пор, пока на вершине стека не окажется основа β . Затем он свертывает β к левой части соответствующей продукции. Синтаксический анализатор повторяет этот цикл, пока не будет обнаружена ошибка или он не придет в конфигурацию, когда в стеке находится только стартовый символ, а входной буфер пуст:

Стек	Вход
\$S	\$

Попав в эту конфигурацию, синтаксический анализатор прекращает работу и сообщает об успешном разборе входной строки.

Пример 13.с

Пройдем пошагово все действия, выполняемые синтаксическим анализатором при разборе входной строки $id_1 + id_2 * id_3$ грамматики (13.1), используя первое порождение из примера 13.а. Последовательность действий

показана в таблице 13.2. Заметим, что поскольку грамматика (13.1) имеет два правых порождения для данной входной строки, существует еще одна последовательность переносов и сверток, которые может выполнить анализатор.

Таблица 13.2. Конфигурации ПС-анализатора для входной строки $id_1 + id_2 * id_3$

Стек	Вход	Действие
(1) \$	$id_1 + id_2 * id_3 \$$	Перенос
(2) $\$id_1$	$+id_2 * id_3 \$$	Свертка по $E \rightarrow id$
(3) $\$E$	$+id_2 * id_3 \$$	Перенос
(4) $\$E +$	$id_2 * id_3 \$$	Перенос
(5) $\$E + id_2$	$*id_3 \$$	Свертка по $E \rightarrow id$
(6) $\$E + E$	$*id_3 \$$	Перенос
(7) $\$E + E *$	$id_3 \$$	Перенос
(8) $\$E + E * id_3$	$\$$	Свертка по $E \rightarrow id$
(9) $\$E + E * E$	$\$$	Свертка по $E \rightarrow E * E$
(10) $\$E + E$	$\$$	Свертка по $E \rightarrow E + E$
(11) $\$E$	$\$$	Допуск

Основными операциями синтаксического анализатора являются перенос и свертка, но на самом деле ПС-анализатор может выполнять четыре действия: (1) перенос, (2) свертка, (3) допуск, (4) ошибка.

1. При переносе очередной входной символ переносится на вершину стека.
2. При свертке синтаксический анализатор распознает правый конец основы на вершине стека, после чего он должен найти левый конец основы и принять решение о том, каким нетерминалом заменить основу.
3. При допуске синтаксический анализатор сообщает об успешном разборе входной строки.
4. При ошибке синтаксический анализатор обнаруживает ошибку во входном потоке и вызывает программу восстановления после ошибок.

Рассмотрим очень важный факт, который поясняет использование стека в ПС-анализаторе: основа всегда находится на вершине стека и никогда – внутри него. Это становится очевидным при рассмотрении возможных видов двух последовательных шагов в любом правом порождении. Эта два шага могут быть следующими:

$$(1) \underset{rm}{S} \overset{*}{\Rightarrow} \underset{rm}{\alpha A z} \overset{*}{\Rightarrow} \underset{rm}{\alpha \beta B y z} \overset{*}{\Rightarrow} \underset{rm}{\alpha \beta \gamma y z}$$

$$(2) \underset{rm}{S} \overset{*}{\Rightarrow} \underset{rm}{\alpha B x A z} \overset{*}{\Rightarrow} \underset{rm}{\alpha B x y z} \overset{*}{\Rightarrow} \underset{rm}{\alpha \gamma x y z}$$

В случае (1) A заменяется на $\beta B y$, а затем крайний справа нетерминал B в правой части – на γ . В случае (2) A также заменяется первым, но в этот раз правая часть представляет собой строку γ , состоящую только из терминалов. Следующий крайний справа нетерминал B находится слева от γ .

Рассмотрим случай (1) в обратном порядке, начиная с момента, когда ПС-анализатор достиг конфигурации

Стек	Вход
$\$ \alpha \beta \gamma$	$y z \$$

Теперь синтаксический анализатор свертывает основу γ в B и переходит в конфигурацию

Стек	Вход
$\$ \alpha \beta B$	$y z \$$

Поскольку B является крайним справа нетерминалом в $\alpha \beta B y z$, правый конец основы строки $\alpha \beta B y z$ не может находиться внутри стека. Таким образом, синтаксический анализатор может перенести строку y в стек для получения конфигурации

Стек	Вход
$\$ \alpha \beta B y$	$z \$$

в которой $\beta B y$ является основой и свертывается в A .

В случае (2) в конфигураций

Стек	Вход
$\$ \alpha \gamma$	$x y z \$$

основа γ находится на вершине стека. После свертки γ в B синтаксический анализатор может перенести строку $x\gamma$ для получения следующей основы γ на вершине стека:

Стек	Вход
$\$aBx\gamma$	$z\$$

Теперь синтаксический анализатор свертывает γ в A .

В обоих случаях после свертки синтаксический анализатор для получения очередной основы переносит нуль или несколько символов в стек. Синтаксический анализатор никогда не заглядывает внутрь стека в поисках правого края основы. Все это делает стек особенно удобным для использования в реализации ПС-анализатора. Впрочем, мы все еще не выяснили, каким образом осуществлять выбор очередного действия для корректной работы синтаксического анализатора.

Активные префиксы

Префиксы правосентенциальных форм, которые встречаются в стеке ПС-анализатора, называются **активными** (**viable prefixes**). Эквивалентное определение активного префикса заключается в том, что это – префикс правосентенциальной формы, который не выходит за правый конец крайней справа основы этой сентенциальной формы. Согласно этому определению, к концу активного префикса всегда можно добавить терминальные символы для получения правосентенциальной формы. Следовательно, просканированная часть входного потока не содержит ошибок только в том случае, когда она может быть свернута в активный префикс.

Конфликты в процессе ПС-анализа

Существуют контекстно-свободные грамматики, для которых ПС-анализ не применим. Любой ПС-анализатор для такой грамматики может

достичь конфигурации, в которой синтаксический анализатор по информации о содержимом стека и об очередном входном символе не в состоянии решить, должен ли использоваться перенос или свертка (**конфликт перенос/свертка, shift/reduce conflict**), либо какая из нескольких возможных сверток должна применяться (**конфликт свертка/свертка, reduce/reduce conflict**). Сейчас мы рассмотрим несколько примеров синтаксических конструкций, приводящих к построению таких грамматик. Технически эти грамматики не входят в класс LR(k)-грамматик, мы говорим о них как о не-LR-грамматиках. k в LR(k) означает количество символов входного потока, следующих за текущим, которые синтаксический анализатор может при необходимости просмотреть, не перенося в стек. Практически используемые грамматики в основном принадлежат классу LR(1).

Пример 13.d

Неоднозначная грамматика не может быть LR-грамматикой. Рассмотрим классический случай грамматики "кочующего else":

```
stmt →  if expr then stmt
      |  if expr then stmt else stmt
      |  other
```

Если ПС-анализатор находится в конфигурации

Стек	Вход
\$... if <i>expr</i> then <i>stmt</i>	else ... \$

то мы не можем сказать, является ли подстрока **if *expr* then *stmt*** основой, независимо от того, что находится в стеке под нею. Здесь возникает конфликт перенос/свертка – в зависимости от того, что следует за **else** во входном потоке, верным решением может оказаться свертка **if *expr* then *stmt*** в *stmt*, а может – перенос **else** и поиск еще одного *stmt* для завершения альтернативы **if *expr* then *stmt* else *stmt***. Таким образом, мы не можем сказать, что следует использовать в данном случае – перенос или свертку, а значит, грамматика не является LR(1)-грамматикой. Более того, никакая

неоднозначная грамматика не может быть LR(k)-грамматикой ни при каком k .

Следует отметить, однако, что ПС-анализ может быть легко адаптирован к разбору некоторых неоднозначных грамматик вроде приведенной выше. При построении такого синтаксического анализатора для грамматики, содержащей две приведенные выше продукции, мы получим конфликт переноса/свертки – переносе **else** или свертки $stmt \rightarrow \text{if } expr \text{ then } stmt$. Если мы разрешим конфликт в пользу переноса, синтаксический анализатор будет работать нормально.

Еще одна причина того, что грамматика не является LR, возникает, когда есть основа, но содержимого стека и очередного входного символа недостаточно для определения продукции, которая должна использоваться в свертке. Следующий пример иллюстрирует эту ситуацию.

Пример 13.e

Предположим, что есть лексический анализатор, который возвращает символ **id** для всех идентификаторов, независимо от их использования. Предположим также, что наш язык вызывает процедуры по именам, с параметрами взятыми в скобки; тот же синтаксис используется и для работы с массивами. Поскольку трансляции индексов массива и параметров процедуры существенно отличаются друг от друга, мы должны использовать различные продукции для порождения списка фактических параметров и индексов. Следовательно, наша грамматика может иметь (среди прочих) продукции типа

- | | | |
|-----|----------------|----------------------------|
| (1) | $stmt$ | id (parameter_list) |
| (2) | $stmt$ | $expr := expr$ |
| (3) | parameter_list | parameter_list , parameter |
| (4) | parameter_list | parameter |
| (5) | parameter | id |
| (6) | $expr$ | id (expr_list) |

- (7) `expr` `id`
- (8) `expr_list` `expr_list , expr`
- (9) `expr_list` `expr`

Инструкция, начинающаяся с `A(I, J)`, будет передана синтаксическому анализатору как поток символов **id(id, id)**. После переноса первых трех символов в стек ПС-анализатор окажется в конфигурации

Стек	Вход
... id (id	, id ...

Очевидно, что символ **id** на вершине стека должен быть свернут, но какой продукцией? Правильный выбор – продукция (5), если `A` – процедура, и (7), если `A` – массив. Содержимое стека не может подсказать, чем является `A`; для принятия решения мы должны использовать информацию из таблицы символов, которая была занесена туда при объявлении `A`.

Одно из решений состоит в замене символа **id** в продукции (1) на **procid** и использовании более интеллектуального лексического анализатора, который возвращает **procid** при распознавании идентификатора, представляющего собой имя процедуры. Такой способ требует от лексического анализатора обращения к таблице символов перед тем, как вернуть символ.

Если мы внесем эти изменения, то при обработке `A(I, J)` синтаксический анализатор может оказаться либо в конфигурации, приведенной ранее, либо в следующей:

Стек	Вход
... procid (id	, id ...

В первом случае выбираем свертку по продукции (7); в последнем – по продукции (5). Обратите внимание, что выбор определяется третьим от вершины символом в стеке, который даже не участвует в свертке. Для управления разбором ПС-анализ может использовать информацию "из глубин" стека.

Контрольные вопросы

1. Восходящий разбор является левосторонним или правосторонним?
2. Дайте понятие основы правосенденциальной формы.
3. В чем заключается "обрезка основ".
4. Какие четыре действия может выполнять ПС-анализатор?
5. Что такое активный префикс?
6. Приведите пример конфликта при работе ПС-анализатора.

14. РАБОТА С ТАБЛИЦЕЙ СИМВОЛОВ

Поскольку синтаксический анализатор обычно использует контекстно-свободную грамматику, необходимо найти метод определения контекстно-зависимых частей языка. Например, во многих языках идентификаторы не могут применяться, если они не описаны, также имеются ограничения в отношении способов употребления в программе значений различных типов. Для запоминания описанных идентификаторов и их типов большинство компиляторов пользуется таблицей символов.

Когда описывается идентификатор, например,

```
int a;
```

это называется **определяющей реализацией** `a`. Однако `a` может встречаться и в другом контексте:

```
a=4 или a+b или read(a)
```

здесь имеются **прикладные реализации** `a`.

В случае определяющей реализации идентификатора (специфицируемого пользователем типа данных, операции и т. п.), компилятор помещает объект в таблицу символов. В случае прикладной реализации в таблице символов осуществляется поиск элемента, соответствующего определяющей реализации объекта, чтобы узнать его тип и (возможно) другие признаки, требующиеся во время компиляции.

Во многих языках один и тот же идентификатор может использоваться для представления в разных частях программы различных объектов. В таких случаях структура программы помогает различать эти объекты, например

```
{
  int a;
  ...
}
...
{
  char a;
  ...
}
```

В данном случае в двух разных блоках *a* представляет два разных объекта. Таблица символов должна иметь ту же блочную структуру, что и программа, чтобы различать виды употребления одного и того же идентификатора.

Многие современные языки высокого уровня обладают следующими свойствами:

Определяющая реализация идентификатора появляется (текстуально) раньше любой прикладной реализации.

При наличии прикладной реализации идентификатора соответствующая определяющая реализация находится в наименьшем включающем блоке, в котором содержится описание этого идентификатора.

В одном и том же блоке идентификатор не может описываться более одного раза.

В таблице символов компилятора может содержаться и другая информация об идентификаторе, необходимая во время компиляции, например, его адрес в ходе прогона или в случае константы – ее значение.

Реализация таблиц символов внутри программ возможна либо в виде массива, либо в виде цепочной структуры, в которой каждый элемент содержит ссылку на последующий и, возможно, предыдущий элемент структуры.

Реализация в виде массива

Достоинства такой организации:

Быстрое выделение памяти под таблицу символов (происходит один раз при объявлении массива)

Экономия места за счет отсутствия необходимости хранить информацию о размещении других элементов массива

Простота реализации методов ускоренного поиска, имитации стека и т.д.

Недостатки:

При трансляции небольших программ и малом количестве идентификаторов – неэффективное использование памяти, т.к. большая часть массива остается незанятой

При трансляции больших программ может возникнуть ситуация, когда физическая память для размещения информации о переменных имеется, а массив уже полностью заполнен. Соответственно, из-за переполнения трансляция оказывается невозможной.

Реализация в виде цепочной структуры (связанного списка)

Достоинством такой организации является наиболее полное использование ресурсов памяти.

Недостатки:

1. Выделение памяти осуществляется значительно медленнее, т.к. производится отдельно под каждый элемент.
2. Дополнительные затраты памяти, поскольку хранятся ссылки на последующий и, возможно, предыдущий элемент

На практике также встречается комбинация этих подходов.

В качестве структуры данных для таблицы символов очень удобен стек, каждым элементом которого служит элемент этой таблицы символов.

При встрече с описанием соответствующий элемент таблицы символов помещается в верхнюю часть стека, а при выходе из блока все элементы таблицы символов, соответствующие описаниям в этом блоке, удаляются из стека. Указатель же стека понижается до положения, которое он имеет при вхождении в блок. В результате в любой момент разбора элементы таблицы символов, соответствующие всем текущим идентификаторам, находятся в стеке, а связанные с ним прикладные и определяющие реализации идентификаторов требуют поиска в стеке в направлении сверху вниз. Применение стека вместо более сложной цепной структуры дает возможность сэкономить место, занимаемое указателями в этой цепной структуре.

Рассмотренный метод проиллюстрирован на рис.14.1.

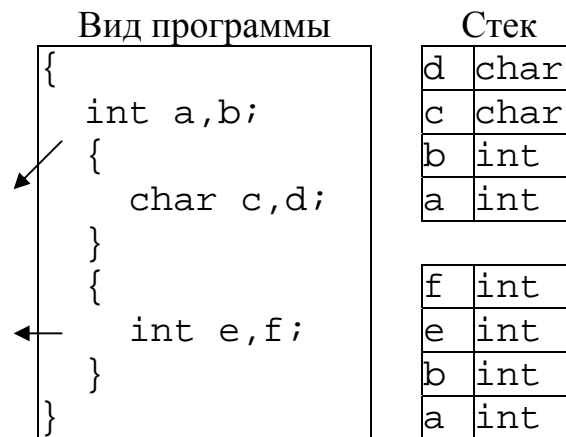


Рис.14.1. Пример реализации со стеком

Контрольные вопросы

1. В чем заключается необходимость использования таблицы символов при синтаксическом анализе?
2. В чем отличие определяющей реализации от прикладной?
3. Назовите преимущества и недостатки реализации таблицы символов в виде массива и связанного списка.

15. ВОССТАНОВЛЕНИЕ ПРИ СИНТАКСИЧЕСКИХ ОШИБКАХ

В процессе анализа текста программы, транслятор должен выдавать соответствующую диагностику об ошибках и продолжать процесс грамматического разбора, возможно находя дальнейшие ошибки.

Для продолжения работы, можно:

1. сделать какие-то предположения о том, что на самом деле имел в виду автор неправильной программы;
2. пропустить некоторую часть входной последовательности;
3. попытаться восстановить текст и при неудаче пропустить часть последовательности.

Сделать достаточно разумные предположения о действительных намерениях программиста сложно, поэтому этот подход применяется редко, хотя работы в этом направлении ведутся.

При 2-м подходе применимы следующие рекомендации.

Восстановление намного облегчается в случае языка с простой структурой. Так, если при обнаружении ошибки пропускается какая-то часть входной последовательности, то язык обязательно должен содержать **служебные (ключевые)** слова, неправильное употребление которых маловероятно и которые поэтому могут использоваться для возобновления грамматического разбора.

По построению программы грамматического разбора. При появлении неправильной конструкции процедура должна пропустить входной текст, пока не встретится символ, который по правилам может следовать за той конструкцией языка, которую она пыталась обнаружить. Это означает, что каждой процедуре грамматического разбора в момент ее текущей активации должно быть известно множество внешних символов (следующих за разбираемой конструкцией). В конце каждой процедуры вставляется явная проверка: действительно ли следующий символ входного текста содержится среди этих внешних символов.

После обнаружения ошибки процедура должна самостоятельно продолжать просмотр текста до того места, откуда можно возобновить анализ, а не прекращать работу и сообщать об ошибке вызвавшей ее процедуре. Таким образом, крайне желательно, чтобы из процедуры грамматического разбора не было другого выхода, кроме обычного завершения работы.

Игнорирование больших фрагментов текста до следующего внешнего символа может иметь нежелательные последствия. Поэтому к множеству символов, прекращающих пропуск текста, добавляют служебные слова, отмечающие начало конструкции, которую не следует пропускать. Таким образом, в качестве параметров процедуре разбора передаются **СИМВОЛЫ ВОЗОБНОВЛЕНИЯ**, а не просто внешние символы. Часто множество символов возобновления с самого начала содержат отдельные служебные слова и при проходе иерархии подцелей грамматического разбора постепенно дополняются внешними символами этих подцелей. Для выполнения описанной выше проверки введем общую функцию и назовем ее `test`. Данная функция имеет три параметра:

Множество `s1` допустимых следующих символов; если текущий символ к нему не принадлежит, то имеет место ошибка.

Множество `s2` дополнительных символов возобновления, появление которых определенно является ошибкой, но которые ни в коем случае нельзя пропускать.

Номер `n`, который присваивается ошибке, если функция ее обнаружит.

```
void test(char[] s1, char[] s2, int n)
{
    if(!belongsTo(sym, s1)
    {
        error(n);
        s1= unite(s1, s2);
        while(!belongsTo(sym, s1))
            sym = fgetc(input);
    }
}
```

Ясно, что ни одна схема, не может эффективно справляться со всеми возможными неправильными конструкциями. Любая схема восстановления, реализованная с разумными затратами, потерпит неудачу, т.е. не сможет адекватно обработать некоторые ошибочные конструкции. Однако хороший транслятор должен обладать такими свойствами:

- никакая входная последовательность не должна приводить к катастрофе;
- все конструкции, которые по определению языка являются незаконными, должны обнаруживаться и отмечаться;
- ошибки, программиста должны правильно диагностироваться и не вызывать каких-либо дальнейших отклонений в работе транслятора сообщений о так называемых наведенных ошибках.

Первые два требования должны выполняться безусловно, последнее свойство – пожелание, так как всегда и полностью избежать наведенных ошибок практически невозможно.

Контрольные вопросы

1. Перечислите подходы, которые могут применяться для восстановления синтаксического анализа после ошибки. Какой из них является наиболее эффективным?
2. Какие символы считаются символами возобновления?

16. ПОСТФИКСНАЯ ЗАПИСЬ

Обычные арифметические выражения, используемые в повседневной практике и содержащие скобки, называют **инфиксными** выражениями, поскольку знак операции располагается между операндами. Порядок выполнения действий в таких выражениях определяется старшинством операций и скобками. Вычисление и компиляция таких выражений подразумевает их предварительный анализ с целью выявления порядка выполнения операций. Существуют формы записи арифметических выражений без скобок, в которых порядок действий задается порядком знаков операций в выражении. Такие формы записи называются **польской** или **бесскобочной** записью. Польская запись может быть **префиксной**, в которой знак операции предшествует операндам, и **постфиксной**, в которой знак операции следует за операндом. Вычисление и компиляция бесскобочных выражений оказывается проще, чем выражений со скобками, поскольку операции должны выполняться в порядке описания и предварительный анализ не требуется.

Префиксная польская запись (ПрПЗ) определяется следующим образом.

Если инфиксное выражение E представляет собой один операнд a , то ПрПЗ выражения E – это просто a .

Если инфиксное выражение $E_1 * E_2$, где $*$ – знак операции, а E_1 и E_2 – инфиксные выражения для операндов, то ПрПЗ этого выражения – это $*E_1'E_2'$, где E_1', E_2' – ПрПЗ выражений E_1 и E_2 .

Если (E) есть инфиксное выражение, то ПрПЗ этого выражения есть ПрПЗ E .

Перечисленные правила определяют порядок построения ПрПЗ заданного инфиксного выражения. Например, для выражений $(a + b) * (c - d)$ построение ПрПЗ можно выполнить так. Обозначим операнды первой выполняемой операции:

$$E_1 = (a + b) \text{ и } E_2 = (c - d).$$

Согласно определению префиксная запись выражения $E_1 * E_2$ – это $*E_1'E_2'$, где E_1', E_2' – префиксные записи выражений E_1 и E_2 . Выполняя построение постфиксных записей для этих выражений,

$$E_1' = +ab, E_2' = -cd,$$

окончательно получаем результат в виде:

$$*+ab-cd$$

Постфиксная запись отличается тем, что знак операции ставится непосредственно за операндами. Так, например, инфиксной записи $(A+B)$ соответствует **постфиксная форма** $AB+$.

Постфиксная польская запись (ПоПЗ) определяется следующим образом.

Если инфиксное выражение E представляет собой один операнд **а**, то ПоПЗ выражения E – это **а**.

Если инфиксное выражение $E_1 * E_2$, где $*$ – знак операции, E_1, E_2 – инфиксные выражения для операндов, то ПоПЗ этого выражения это – $E_1'E_2'*$, где E_1', E_2' – постфиксные выражения E_1, E_2 .

Если (E) есть инфиксное выражение, то постфиксная запись этого выражения есть постфиксная запись E .

Аналогично предыдущему примеру построим ПоПЗ выражения

$$(a + b) * (c - d).$$

Обозначая операнды внешней операции

$$E_1 = (a + b) \text{ и } E_2 = (c - d)$$

найдем постфиксные записи операндов, которые имеют вид:

$$E_1' = ab+ \text{ и } E_2' = cd-$$

Подставляя полученные постфиксные записи в выражение

$$E_1'E_2'*$$

окончательно получаем:

$$ab+cd-*$$

Постфиксная запись выражений обладает двумя ценными свойствами, благодаря которым ее широко используют в компиляторах:

Для записи любого выражения не нужны скобки. Так как оператор непосредственно следует за операндами, участвующими в операции, неопределенность в указании операндов отсутствует. Например, выражение $(A+B)+C$ имеет постфиксную форму $AB+C+$, а выражение $A+(B+C)$ представляется в форме $ABC++$.

К моменту считывания очередного оператора соответствующие операнды уже были прочитаны. Поэтому оператор может быть выполнен без считывания каких-либо дополнительных данных.

Сказанное выше относится к бинарным операциям, однако это нетрудно распространить на унарные операции. Так, унарный оператор отрицания (эту операцию будем обозначать знаком \sim) просто ставят непосредственно за аргументом. Например, инфиксная запись $\sim A$ представляется в форме $A\sim$, а выражение $\sim(A+B)$ преобразуется в $AB+\sim$. (Заметим, что знак "-" может стоять в инфиксной записи, указывая как бинарную, так и унарную операцию, и его правильный смысл становится очевидным из контекста. В постфиксной записи это сделать труднее.

Благодаря описанным выше свойствам выражение в постфиксной форме может быть вычислено с помощью простого алгоритма:

```
while(lex != NULL) // пока в выражении еще есть лексемы
{
    lex = getNextLex(); // считать следующую лексему
    if(isOperand(lex)) // лексема есть операнд
        push(lex); // записать лексему в стек
    if(isOperator(lex)) // лексема есть оператор
        push(performOperation(lex, pop(), pop()));
    /* выполнить указанную лексемой операцию над
последними элементами, записанными в стек, и заменить
эти элементы результатом операции;
    */
}
```

В результате вычислений значение выражения оказывается единственным элементом стека.

Такой алгоритм, выполняющий операции над стеком, лежит практически в основе любого компилятора. Обычно блок синтаксического анализа преобразует программу в постфиксную форму, после чего генератор кода строит объектную программу, просматривая выражения вышеописанным методом.

Очевидно, что вычисление постфиксной записи выражения не представляет трудности, преобразование же инфиксной записи в постфиксную заметно сложнее. Предположим, что продукция языка имеет вид

$$A \rightarrow B \text{ и } C$$

где A , B и C являются нетерминалами, а "и" – терминал. Будем считать, что знаки операций являются терминалами. Тогда эта продукция означает, что нетерминал A является инфиксной записью, в которой участвуют операнды B и C вместе с оператором "и". Следовательно, постфиксная форма выражения образуется из операндов B , C , за которыми следует оператор "и". Таким образом, для данной продукции постфиксная запись имеет вид

Постфиксная запись для B

Постфиксная запись для C

и

Если терминальные символы выводятся в выходной файл в таком порядке, то все фразы языка будут представлены в постфиксной форме.

Преобразование выражения из инфиксной записи в постфиксную

Части выражения, заключенные в скобки на самом нижнем уровне скобочных вложений для данного выражения, должны быть сначала преобразованы в постфиксную форму, с тем, чтобы их можно было рассматривать как один операнд. При таком подходе преобразование всего

выражения приведет к полному исключению из него скобок. Последняя открываемая в группе скобок скобочная пара содержит первое преобразуемое выражение в этой группе. Принцип "последнее открываемое выражение вычисляется первым" предполагает использование стека.

Рассмотрим два выражения в инфиксной форме: $A+B*C$ и $(A+B)*C$ и соответствующие им постфиксные формы $ABC*+$ и $AB+C*$. В каждом случае порядок следования операндов в этих формах совпадает с порядком следования операндов в исходных выражениях. При просмотре первого выражения $(A+B*C)$ первый операнд A может быть сразу же помещен в постфиксное выражение. Очевидно, что символ "+" не может быть помещен в это выражение до тех пор, пока туда не будет помещен второй, еще не просмотренный операнд. Следовательно, он (т.е. символ "+") должен быть сохранен, а впоследствии извлечен и помещен в соответствующую позицию. После просмотра операнда B этот символ записывается вслед за операндом A . К этому моменту просмотренными уже оказываются два операнда. Однако извлекать и помещать символ "+" в постфиксную запись еще рано, т.к. за ним следует символ "*", имеющий более высокий приоритет. Во втором выражении наличие скобок обуславливает выполнение операции "+" в первую очередь.

Так как при преобразовании инфиксной формы в постфиксную правила приоритета играют существенную роль, для их учета введем функцию `precedence(oper1, oper2)`, где `oper1` и `oper2` – символы, обозначающие операции. Эта функция возвращает значение `TRUE`, если `oper1` имеет более высокий или равный приоритет по сравнению с `oper2`. В противном случае функция `precedence(oper1, oper2)` возвращает значение `FALSE`. Например, значения функций `precedence(" * ", " + ")` и `precedence(" + " " + ")` – "истина", а `precedence(" + ", " * ")` – "ложь". Равный приоритет подчеркнут, так как неправильное формирование постфиксной записи (как вручную, так разработанными студентами

программными генераторами) для операций с равным приоритетом является одной из типичных ошибок.

Рассмотрим теперь макет алгоритма для преобразования строки, представленной в инфиксной форме и не содержащей скобок, в постфиксную строку. Поскольку мы считаем, что во входной строке скобки отсутствуют, единственным признаком порядка выполнения операций является их приоритет.

```

1. (Установить в постфиксную строку " ");
2. (Очистить стек с именем opstk);
3. while (на входе еще имеются символы) {
4.     read(symb);
5.     if(isOperand(symb) == TRUE) { //символ есть операнд
6.         (добавить символ к постфиксной строке);
7.     } else { //символ есть операция
8.         while(empty(stack) == FALSE) &&
           (precedence(stacktop(opstk), symb) == TRUE)) {
9.             smbtp = pop(opstk);
/* smbtp имеет приоритет больший, чем symb, поэтому она
может быть добавлена к постфиксной строке. */
10.            (добавить smbtp к постфиксной строке);
11.        } // end while
/* в этой точке либо opstk пуст, либо symb имеет
приоритет над stacktop(opstk). Нельзя поместить symb в
постфиксную строку до тех пор, пока не считана
следующая операция, которая может иметь более высокий
приоритет. Следовательно, необходимо сохранить symb. */
12. push(opstk, symb);
13. } // end if
14. } // end while

```



```

/* к этому моменту строка оказывается просмотренной
целиком. Необходимо поместить оставшиеся в стеке
операции в постфиксную строку. */
15. while(empty(opstk) == FALSE) {
16.     smbtp=pop(opstk);
17.     (добавить smbtp к постфиксной строке);
18. } // end while

```

Для обеспечения возможности работы со скобками после считывания открывающейся скобки она записывается в стек. Это может быть выполнено путем установки правила `precedence(op, "(") = FALSE` для любого символа операции, отличного от символа правой (закрывающей) скобки. Также определим `precedence("(", op) = FALSE` для того, чтобы символ операции, появляющийся после левой скобки, записывался в стек.

После считывания закрывающей скобки все операции вплоть до первой открывающей скобки должны быть прочитаны из стека и помещены в постфиксную строку. Это может быть сделано путем установки `precedence(op, ")") = TRUE` для всех операций `op`, отличных от левой скобки. После считывания этих операций из стека и закрытия открывающей скобки необходимо выполнить следующую операцию. Открывающая скобка должна быть удалена из стека и отброшена вместе с закрывающей скобкой. Обе скобки не помещаются затем ни в постфиксную строку, ни в стек. Установим функцию `precedence("(", ")")` равной `FALSE`. Это гарантирует нам то, что при достижении закрывающей скобки цикл, начинающийся в строке 8, будет пропущен, а открывающая скобка не будет помещена в постфиксную строку.

Выполнение продолжится со строки 12. Однако, поскольку закрывающая скобка не должна помещаться в стек, строка 12 заменяется оператором

```

if((empty(opstk) == TRUE) || (symb <> ")"))
    push(opstk, symb);

```

```
else smbtp=pop(opstk);
```

С учетом приведенных соглашений для функции `precedence`, а также исправлений для строки 12 рассмотренный алгоритм может быть использован для преобразования любой строки, записанной в инфиксной форме, в постфиксную. Подытожим правила приоритетности скобок:

`precedence("(", op) == FALSE` для любой операции `op`

`precedence(op, "(") == FALSE` для любой операции `op`, отличной от `)`

`precedence(op, ")") == TRUE` для любой операции `op`, отличной от `(`

`precedence(")", op) = "неопределенно"` для любой операции `op`, (попытка сравнения двух указанных операций означает ошибку)

Проиллюстрируем этот алгоритм несколькими примерами:

Пример1:

$A+B*C$

Приводится содержимое `symb`, постфиксной строки и `opstk` после просмотра каждого символа. Вершина `opstk` находится справа.

Строка	symb	Постфиксная строка	opstk
1	A	A	
2	+	A	+
3	B	AB	+
4	*	AB	+*
5	C	ABC	+*
6		ABC*	+
7		ABC*+	

Строки 1, 3 и 5 соответствуют просмотру операнда таким образом, что символ (`symb`) немедленно помещается в постфиксную строку. В строке 2 была обнаружена операция, а стек оказался пустым, поэтому операция помещается в стек. В строке 4 приоритет нового символа (`*`) больше, чем

приоритет символа, расположенного в вершине стека (+), поэтому новый символ помещается в стек. На 6-м и 7-м шагах входная строка пуста, поэтому из стека считываются элементы, которые затем помещаются в постфиксную строку.

Пример2:

$$(A+B)*C$$

Строка	symp	Постфиксная строка	Opstk
1	((
2	A	A	(
3	+	A	(+
4	B	AB	(+
5)	AB+	
6	*	AB+	*
8		AB+C*	

В этом примере при обнаружении правой скобки из стека начинают извлекаться элементы до тех пор, пока не будет обнаружена левая скобка, после чего обе скобки отбрасываются. Использование скобок для изменения приоритетности выполнения операций приводит к последовательности их расположения в постфиксной строке, отличной от последовательности в примере 1.

Стек используется для запоминания в нем просмотренных ранее операций. Если текущая просматриваемая операция имеет больший приоритет, чем операция, расположенная в вершине стека, то эта новая операция записывается в стек. Это означает, что при окончательной выборке из стека всех элементов и записи их в одну строку в постфиксной форме эта новая операция будет предшествовать операции, ранее расположенной перед ним (что является правильным, поскольку она имеет более высокий приоритет). С другой стороны, если приоритет новой операции меньший или равный, чем у операции из стека, то операция, находящаяся на вершине

стека, должна быть выполнена первой. Следовательно, она извлекается из стека и помещается в выходную строку, а текущий рассматриваемый символ сравнивается со следующим элементом, занимающим теперь вершину стека, и т. д. Помещая во входную строку скобки, можно изменить последовательность вычислений. Так, при обнаружении левой скобки она записывается в стек. При обнаружении соответствующей ей правой скобки все операции между этими скобками помещаются в выходную строку, поскольку они выполняются прежде любых других операций, расположенных за этими скобками.

Постфиксная запись операторов (IF и др.)

Ясно, что постфиксная запись для выражений формируется достаточно просто. Для операторов этот процесс выполняется сложнее. Рассмотрим его на примере оператора IF, синтаксис которого задан следующей конструкцией:

```
<IF_STMT> ::= IF <ВЫРАЖЕНИЕ> THEN
<СПИСОК ПРЕДЛОЖЕНИЙ>
{ELSE <СПИСОК ПРЕДЛОЖЕНИЙ>}
END
```

Фигурные скобки здесь являются метасимволами, обозначающими необходимость конструкции. Чтобы выполнить это предложение, необходимо оценить выражение и, если его значение ложно, передать управление группе операторов, следующих за словом ELSE (если оно есть), или в конец предложения, отмеченный словом END. Однако ключевые слова ELSE или END еще не прочитаны на данном этапе анализа, и поэтому мы не располагаем точным указанием места, куда следует передать управление.

Эту трудность можно преодолеть, если определить точку передачи управления. Мы отметим эту точку так называемой системной меткой и на текущем этапе разбора поместим имя этой метки в постфиксную запись непосредственно за выражением. Когда в тексте программы встретится

ключевое слово ELSE (или END), поместим такую же системную метку в соответствующем месте постфиксной записи. Таким образом, системная метка присутствует в анализируемой программе в двух местах: после выражения в точке, где выполняется передача управления вперед, а также в точке, куда должно быть передано управление. С помощью этих данных генератор кода установит соответствующую адресную связь.

Можно построить команду перехода вперед, которая обходит список операторов, следующих за ключевым словом THEN, однако в связи с обработкой ключевого слова END могут возникнуть некоторые осложнения. Если в предложении IF слово ELSE опущено, то передача управления должна быть совершена в точку, соответствующую слову END. Однако если слово ELSE входит в предложение IF, то управление должно быть передано в точку, соответствующую этому слову, а от группы операторов, следующих за словом THEN, должен происходить безусловный переход к точке, соответствующей слову END. Эта мера позволит после выполнения операторов из группы, следующей за словом THEN, обойти выполнение операторов, написанных после слова ELSE. Заметим, однако, что в обоих случаях слово END представляет точку в предложении, куда передается управление одной командой перехода вперед. Благодаря этому обстоятельству удастся связать друг с другом части объектного кода, реализующего предложение IF.

Учитывая сказанное выше, постфиксную запись предложения IF можно представить в следующей форме:

<ВЫРАЖЕНИЕ>

Метка перехода вперед для слова IF

IF

<СПИСОК ПРЕДЛОЖЕНИЙ>

{Метка обхода /* Метка обхода списка операторов ELSE */

Метка перехода вперед

```

/* Точка, куда передается управление при обходе списка
операторов THEN */
ELSE
<СПИСОК ПРЕДЛОЖЕНИЙ> }
Системная метка для передачи управления из точки,
отмеченной меткой перехода вперед или меткой обхода
списка операторов ELSE
END_IF

```

где IF является бинарным оператором, который проверяет истинность выражения и, если его значение равно 0 (ЛОЖНО), обходит список операторов THEN, а ELSE – бинарный оператор, который строит команду перехода к списку операторов ELSE и определяет метку перехода вперед. Аналогичные действия выполняет и запись END_IF.

Контрольные вопросы

1. Назовите отличия префиксной, инфиксной и постфиксной записи выражений.
2. В чем преимущество постфиксной записи с точки зрения ее применения в компиляторах?
3. Существуют ли формальные способы преобразования инфиксной записи выражения в постфиксную?
4. Вычислить значение представленного в постфиксной записи выражения при $a = 2$; $b = 4$ (все переменные и числовые константы имеют длину 1 символ).

1 a + 1 a 3 - - * b +

5. Преобразовать инфиксную запись выражения в постфиксную.

$(a + (a - b)) * (b + 2 - a)$

17. ВНУТРЕННИЕ ФОРМЫ

Построение кода программы непосредственно из постфиксной записи возможно, но такую форму записи трудно оптимизировать. Большинство компиляторов используют внутренние формы, удобные для оптимизации, для построения объектного кода программы. Одной из наиболее распространенных форм внутреннего представления генерируемого кода являются четверки.

Четверка – это объект, состоящий из четырех элементов: операции, двух операндов и результата. Если в результате операции вычисляется значение некоторой переменной, то такую четверку нетрудно построить. Например, предложение

$$X := Y + Z$$

представляется четверкой

$$(PLUS_OP, S_y, S_z, S_x),$$

которая указывает, что надо сложить (*PLUS_OP*) переменную, определенную ячейкой *S_y* таблицы символов, с переменной, определенной ячейкой *S_z*, и сохранить результат в *S_x*. Теперь исходное предложение оказалось представленным самостоятельной единицей, которую генератор кода может обрабатывать независимо от места ее размещения. Таким образом, оптимизатор может изменять последовательность операций (четверок), не усложняя процесса генерирования кода.

Для унарных операторов можно просто игнорировать поле второго операнда четверки, а операции, которые требуют более двух операндов, можно представить в форме последовательности, состоящей из нескольких четверок.

Например, оператор $X := F(A, B, C, D)$ можно записать в виде группы из трех четверок

$$(F1, A, B, T1)$$

$$(F2, T1, C, T2)$$

$$(F, T2, D, X)$$

Функции $F1$ и $F2$ производят промежуточные вычисления, а ячейки $T1$ и $T2$ предназначены для хранения результатов этих действий. Во время фактического построения программы генератор кода сможет правильно представить в объектном коде операцию $F1$ (за которой должны следовать операции $F2$ и F).

Рассмотрим еще пример, связанный с использованием промежуточных ячеек. Пусть дано предложение

$$X := X + Y * Z,$$

которое представлено постфиксной записью

$$Sx \ Sx \ Sy \ Sz * + :=$$

В это предложение входят операции умножения Y на Z , сложения результата с X и присвоения переменной X значения полученной суммы. Во время генерирования четверки, осуществляющей умножение, потребуется ячейка, в которой будет храниться промежуточный результат этого действия. Будем предполагать, что в этом случае создается временная, или внутренняя переменная. Таким образом, рассматриваемое предложение будет представлено последовательностью

$$(MULT_OP, Sy, Sz, T1)$$

$$(ADD_OP, Sx, T1, Sx),$$

в которой первая четверка указывает на то, что надо умножить Y на Z и записать результат в $T1$, а вторая четверка определяет необходимость сложения переменной X с переменной $T1$, содержащей результат операции $Y * Z$, и записи суммы в X .

Контрольные вопросы

1. Из каких частей состоит "четверка" во внутренней форме представления генерируемого кода?
2. В чем состоит преимущество применения четверок при генерации кода?

18. МЕТОДЫ ГЕНЕРИРОВАНИЯ КОДА

Обычно одинаковые фрагменты внутренней записи кода (операция в постфиксной записи, четверка и пр.) формируют одни и те же команды машинного языка. Например, четверка *PLUS_OP* в случае, если все операции в процессоре, для которого генерируется код, выполняются над регистром-аккумулятором, всегда генерирует следующий код:

```
LOAD регистр, операнд 1
ADD регистр, операнд 2
STORE регистр, результат
```

Эта последовательность машинных команд является всегда корректной, но не обязательно оптимальной. Например, предложение $X := X + Y * Z$ реализуется шестью командами:

```
LOAD регистр, Y      (четверка (MULT_OP, Sy, Sz, T1))
MUL регистр, Z
STORE регистр, T1
LOAD регистр, X      (четверка (ADD_OP, Sx, T1, Sx))
ADD регистр, T1
STORE регистр, X
```

в то время как можно построить более короткую программу, дающую тот же результат:

```
LOAD регистр, Y
MUL регистр, Z
ADD регистр, X
STORE регистр, X
```

Отметим, что код, который генерируется непосредственным образом, является всегда правильным, но не всегда оптимальным. Поэтому желательно иметь средства, позволяющие изменить код, не влияя на корректность вычислений.

Так как каждой четверке (или другому элементу промежуточной записи) в большинстве случаев соответствует единственная

последовательность машинных команд, генератор кода обычно представлен набором подпрограмм, по одной на каждую четверку.

19. ЛИТЕРАТУРА

1. Разработка трансляторов с языков САПР. Методические указания к выполнению лабораторных работ /Сост.: Кревский И.Г.; ПГТУ. - Пенза, 1993.
2. Берри Р., Микинз Б. Язык Си: введение для программистов: Пер. с англ. - М.: Финансы и статистика, 1988.
3. Язык «Си» для профессионалов / По материалам книги Г.Шилдта. – М.: ИВК-Софт, 1992.
4. Вирт Н. Алгоритмы + структуры данных = программы. - М.:Мир, 1985.
5. Зелковиц М., Шоу А., Гэннон Дж. Принципы разработки программного обеспечения. Пер. с англ. - М.: Мир, 1982.
6. Хантер Р. Проектирование и конструирование компиляторов: Пер. с англ. - М.: Финансы и статистика, 1984.
7. Разработка САПР. В 10 кн. Кн.3 Проектирование программного обеспечения САПР: Практ.пособие /Б.С.Федоров, Н.Б.Гуляев. – М.: Высш.шк., 1990.
8. Лэнгсам Й., Огенстайн М., Тененбаум А. Структуры данных для персональных ЭВМ: Пер. с англ. - М.: Мир, 1989.
9. Льюис Ф., Розенкранц Д., Стирнз Р. Теоретические основы проектирования компиляторов: Пер. с англ. - М.: Мир, 1979.
- 10.Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. В 2-х т.: Пер. с англ. - М.: Мир, 1978.
- 11.Грис Д. Конструирование компиляторов для цифровых вычислительных машин. Пер. с англ. - М.: Мир, 1975.
12. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. : Пер с англ. – М. Издательский дом «Вильямс», 2001. – 768с.

20. ЛАБОРАТОРНЫЕ РАБОТЫ

Основной целью лабораторных работ по разработке трансляторов является получение практических навыков, позволяющих разрабатывать трансляторы как языков программирования, так специализированных языков САПР. Отличительной особенностью данных лабораторных работ является наличие одного сквозного задания (формального описания реализуемого языка) для всех работ. Заданные для реализации языки подобны очень простым языкам программирования. В каждой работе студенты разрабатывают программу, выполняющую соответствующий этап трансляции с реализуемого языка. Для написания трансляторов рекомендуется использовать язык программирования Си.

Перед тем, как приступить к выполнению лабораторных работ, необходимо изучить первые 4 раздела настоящего пособия.

В лабораторных работах предлагается трехпроходная организация компилятора. Блок сканирования считывает исходную программу и представляет ее в форме файла лексем (Лабораторная работа №1). Синтаксический анализатор читает этот файл, разбирает (Лабораторная работа №2) и выдает новое представление программы в постфиксной форме (Лабораторная работа №3). Наконец, этот файл считывается генератором кода, который создает объектный код программы (Лабораторная работа №4).

При написании учебного транслятора важно то, что реализующие каждый проход отдельные программы легче отлаживать, чем одну большую. Во 2-й и 3-й лабораторных работах пишется одна и та же программа. В результате выполнения лабораторной работы №2 должна быть написана программа, выполняющая синтаксический анализ (включая выявление и диагностику всех синтаксических ошибок). В лабораторной работе №3 в программу добавляются функции формирования постфиксной записи.

ЛАБОРАТОРНАЯ РАБОТА №1.

РАЗРАБОТКА ЛЕКСИЧЕСКОГО АНАЛИЗАТОРА

1. Порядок выполнения работы.

1.1. Ознакомиться с разделом 5 настоящего пособия.

1.2. По варианту задания определить, какие классы лексем будут в вашем языке.

1.3. Составить контрольные примеры на реализуемом языке. Хотя бы один пример должен проверять поведение вашей программы при наличии недопустимых символов в транслируемом файле.

1.4. Запрограммировать и отладить модуль сканирования. Выполнить тестирование на контрольных примерах. Результатом работы должна быть таблица, содержащая лексемы и признаки их классов. Необходимо включить в результирующий файл информацию о номерах строк исходного текста транслируемой программы.

1.5. Оформить отчет.

2. Содержание отчета.

2.1. Название работы и ее исполнители.

2.2. Цель работы.

2.3. БНФ реализуемого языка.

2.4. Список классов лексем реализуемого языка.

2.5. Краткое (по 2-3 предложения) описание процедур (функций), из которых состоит программа лексического анализа. Наилучший вариант – включение описаний в текст программы в виде комментариев.

2.6. Листинг программы.

2.7. Распечатки контрольных примеров и результатов их выполнения.

2.8. Выводы по проделанной работе.

ЛАБОРАТОРНАЯ РАБОТА №2.

РАЗРАБОТКА СИНТАКСИЧЕСКОГО АНАЛИЗАТОРА

1. Порядок выполнения работы.

1.1. Ознакомиться разделами 7-11, 14, 15 настоящего пособия.

1.2. По варианту задания построить синтаксический граф для реализуемого языка. При этом преобразовать описание языка таким образом, чтобы оно учитывало приоритет операций в выражениях.

1.3. Составить контрольные примеры на реализуемом языке. Хотя бы один пример должен проверять поведение вашей программы при наличии синтаксических ошибок в контрольном примере.

1.4. Запрограммировать и отладить программу, производящую синтаксический анализ реализуемого языка. Выполнить тестирование на контрольных примерах. При этом пример пропускается через программу лексического анализа, а файл с лексемами является входным для программы синтаксического анализа. При необходимости доработать модуль сканирования. Лабораторная работа считается выполненной, если программа выдает правильные и понятные сообщения о синтаксических ошибках с указанием строк, где эта ошибка имеет место.

1.5. Оформить отчет.

2. Содержание отчета.

2.1. Название работы и ее исполнители.

2.2. Цель работы.

2.3. Синтаксические диаграммы реализуемого языка.

2.4. Краткое (по 2-3 предложения) описание процедур (функций), из которых состоит программа синтаксического анализа. Наилучший вариант – включение описаний в текст программы в виде комментариев.

2.5. Листинг программы.

2.6. В случае необходимости – информация о доработке программы лексического анализа.

2.7. Распечатки контрольных примеров и результатов их выполнения.

2.8. Выводы по проделанной работе.

ЛАБОРАТОРНАЯ РАБОТА №3.

ФОРМИРОВАНИЕ ПОСТФИКСНОЙ ЗАПИСИ

1. Порядок выполнения работы.

- 1.1. Ознакомиться с разделом 16 настоящего пособия.
- 1.2. Вручную сформировать постфиксную запись для контрольных примеров.
- 1.3. Запрограммировать и включить в программу синтаксического анализа функции, осуществляющие формирование постфиксной записи. Выполнить тестирование на контрольных примерах. В результате работы программы постфиксная запись должна записываться в файл.
- 1.4. Оформить отчет.

2. Содержание отчета.

- 2.1. Название работы и ее исполнители.
- 2.2. Цель работы.
- 2.3. Краткое (по 2-3 предложения) описание процедур (функций) добавленных в программу синтаксического анализа для формирования постфиксной записи, а также информация об их вызове.
- 2.4. Листинг программы.
- 2.5. Распечатки контрольных примеров и результатов их выполнения
- 2.6. Выводы по проделанной работе.

ЛАБОРАТОРНАЯ РАБОТА №4.

РАЗРАБОТКА ГЕНЕРАТОРА КОДА

1. Порядок выполнения работы.

1.1. Ознакомиться с разделом 18 настоящего пособия.

1.2. Ознакомиться с теоретической частью данной лабораторной работы и структурой виртуальной машины, для которой будет производиться генерация мнемкокода.

1.3. Для контрольных примеров на реализуемом языке вручную составить соответствующие им программы на мнемкокоде гипотетического процессора, рассмотренного в данной работе.

1.4. Запрограммировать и отладить программу, производящую генерацию мнемкокода гипотетического процессора. Выполнить тестирование на контрольных примерах. При этом пример пропускается через программы лексического и синтаксического анализа, а файл с постфиксной записью является входным для программы генерации кода. При необходимости доработать программы лексического и синтаксического анализа.

1.5. Оформить отчет.

2. Теоретическая часть

Последней фазой компиляции является генерация кода. Результатом выполнения этой фазы обычно является программа в выполняемых кодах той ЭВМ, на которой она должна выполняться. Однако в ряде случаев в качестве выходного языка транслятора используют ассемблер. В данной работе мы будем генерировать программу на языке ассемблера. Чтобы облегчить написание генератора кода и освободить его от посторонних соображений, связанных с конкретными особенностями какой-либо ЭВМ, будем использовать гипотетический процессор (виртуальную машину). Этот процессор не существует на самом деле (в аппаратном виде). При выборе его архитектуры требовалась максимальная простота и, в то же время, возможность легко выполнять на нем программы на языках, реализуемых в

процессе выполнения лабораторных работ и курсового проектирования. Особенностью архитектуры является то, что все действия выполняются только над элементами в вершине стека, результаты операций также помещаются в вершину стека (рис.20.1). Поэтому в арифметических и логических операциях нет необходимости в указании адреса операндов. Если операция имеет 2 операнда, то ее выполнения подразумевает перенос элемента из вершины стека в регистр-аккумулятор и «понижение» на один элемент вниз указателя стека. Второй операнд, оказавшийся в вершине стека, подается непосредственно в АЛУ. Результат операции помещается в вершину стека вместо него.

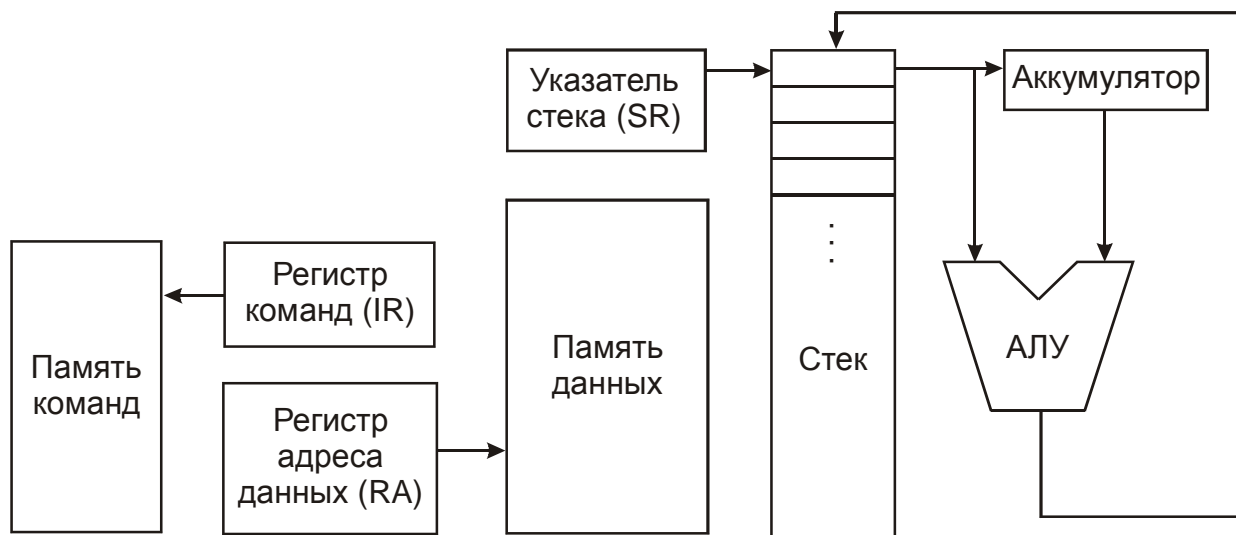


Рис. 20.1. Структура виртуальной машины

Команды:

LIT const – поместить константу в вершину стека.

LOAD n – поместить переменную, размещенную по адресу n в вершину стека.

STO n – запись значения из вершины стека по адресу n (присваивание).

JMP k – безусловный переход к команде, расположенной по адресу k.

JEQ k – переход к команде, расположенной по адресу k в случае равенства двух верхних элементов стека.

JLT k – переход к команде, расположенной по адресу k, если число в вершине стека меньше следующего за ним числа стека.

JLE k – переход к команде, расположенной по адресу k, если число в вершине стека меньше или равно следующему за ним числу стека.

JGT k – переход к команде, расположенной по адресу k, если число в вершине стека больше следующего за ним числа стека.

JGE k – переход к команде, расположенной по адресу k, если число в вершине стека больше или равно следующему за ним числу стека.

JNE k – переход к команде, расположенной по адресу k в случае неравенства двух верхних элементов стека.

ADR – содержимое регистра адреса данных помещается в вершину стека.

STAD – содержимое вершины стека помещается в регистр адреса данных.

ADD – сложение двух верхних элементов стека, результат помещается в вершину стека.

MUL – умножение двух верхних элементов стека, результат помещается в вершину стека.

SUB – вычитание элемента в вершине стека из следующего за ним элемента стека, результат помещается в вершину стека.

DIV – деление на элемент в вершине стека следующего за ним элемента стека, результат помещается в вершину стека.

AND – логическое "И" (логическое умножение) двух верхних элементов стека, результат помещается в вершину стека.

OR – логическое "ИЛИ" (логическое сложение) двух верхних элементов стека, результат помещается в вершину стека.

XOR – сложение по модулю 2 двух верхних элементов стека, результат помещается в вершину стека.

NOT – знаковая инверсия элемента в вершине стека (например, -5 заменяется на 5).

NOL – поразрядная логическая инверсия элемента в вершине стека.

NOP – пустая операция .

Пример генерации кода

Исходная программа	Постфиксная запись	Мнемокод
var a,b,c	a b c	LIT 5
a=5;	a 5 =	STO 1
b=7+a;	b 7 a + =	LIT 7
c=a*b;	c a b * =	LOAD 1
		ADD
		STO 2
		LOAD 1
		LOAD 2
		MUL
		STO 3

3. Содержание отчета

- 3.1. Название работы и ее исполнители.
- 3.2. Цель работы.
- 3.3. Краткое (по 2-3 предложения) описание процедур (функций) программы генерации кода.
- 3.4. Листинг программы.
- 3.5. В случае необходимости, информация о доработке программ лексического и синтаксического анализа.
- 3.6. Распечатки контрольных примеров и результатов их выполнения.
- 3.7. Выводы по проделанной работе.

ПРИЛОЖЕНИЕ А. ВАРИАНТЫ ЗАДАНИЙ К ЛАБОРАТОРНЫМ РАБОТАМ

Во всех вариантах все переменные должны быть объявлены до начала вычислений.

<Буква> – буква латинского алфавита (a...z).

<Цифра> – цифра от 0 до 9.

Вариант 1

$$\langle \text{Программа} \rangle ::= \langle \text{Объявление переменных} \rangle \langle \text{Описание вычислений} \rangle .$$
$$\langle \text{Описание вычислений} \rangle ::= \langle \text{Список присваиваний} \rangle$$
$$\langle \text{Объявление переменных} \rangle ::= \text{Var } \langle \text{Список переменных} \rangle$$
$$\langle \text{Список переменных} \rangle ::= \langle \text{Идент} \rangle \mid \langle \text{Идент} \rangle, \langle \text{Список переменных} \rangle$$
$$\langle \text{Список присваиваний} \rangle ::= \langle \text{Присваивание} \rangle \mid \langle \text{Присваивание} \rangle \langle \text{Список присваиваний} \rangle$$
$$\langle \text{Присваивание} \rangle ::= \langle \text{Идент} \rangle = \langle \text{Выражение} \rangle$$
$$\langle \text{Выражение} \rangle ::= \langle \text{Ун.оп.} \rangle \langle \text{Подвыражение} \rangle \mid \langle \text{Подвыражение} \rangle$$
$$\langle \text{Подвыражение} \rangle ::= (\langle \text{Выражение} \rangle) \mid \langle \text{Операнд} \rangle \mid \langle \text{Подвыражение} \rangle \langle \text{Бин.оп.} \rangle \langle \text{Подвыражение} \rangle$$
 $\langle Y_{H.OP.} \rangle ::= \text{"-"}$
$$\langle \text{Бин.оп.} \rangle ::= "-" \mid "+" \mid "*" \mid "/"$$
$$\langle \text{Операнд} \rangle ::= \langle \text{Идент} \rangle \mid \langle \text{Const} \rangle$$
$$\langle \text{Идент} \rangle ::= \langle \text{Буква} \rangle \langle \text{Идент} \rangle \mid \langle \text{Буква} \rangle$$
$$\langle \text{Const} \rangle ::= \langle \text{Цифра} \rangle \langle \text{Const} \rangle \mid \langle \text{Цифра} \rangle$$

На одной строке может быть только объявление переменных или один оператор присваивания

Вариант 2

$$\langle \text{Программа} \rangle ::= \langle \text{Объявление переменных} \rangle \langle \text{Описание вычислений} \rangle$$
$$\langle \text{Описание вычислений} \rangle ::= \text{Begin } \langle \text{Список присваиваний} \rangle \text{ End}$$
$$\langle \text{Объявление переменных} \rangle ::= \text{Var } \langle \text{Список переменных} \rangle ;$$
$$\langle \text{Список переменных} \rangle ::= \langle \text{Идент} \rangle \mid \langle \text{Идент} \rangle, \langle \text{Список переменных} \rangle$$
$$\langle \text{Список присваиваний} \rangle ::= \langle \text{Присваивание} \rangle \mid \langle \text{Присваивание} \rangle \langle \text{Список присваиваний} \rangle$$
$$\langle \text{Присваивание} \rangle ::= \langle \text{Идент} \rangle := \langle \text{Выражение} \rangle ;$$
$$\langle \text{Выражение} \rangle ::= \langle \text{Ун.оп.} \rangle \langle \text{Подвыражение} \rangle \mid \langle \text{Подвыражение} \rangle$$
$$\langle \text{Подвыражение} \rangle ::= (\langle \text{Выражение} \rangle) \mid \langle \text{Операнд} \rangle \mid \langle \text{Подвыражение} \rangle \langle \text{Бин.оп.} \rangle \langle \text{Подвыражение} \rangle$$
 $\langle Y_{H.OP.} \rangle ::= \text{"-"}$
$$\langle \text{Бин.оп.} \rangle ::= "-" \mid "+" \mid "*" \mid "/"$$
$$\langle \text{Операнд} \rangle ::= \langle \text{Идент} \rangle \mid \langle \text{Const} \rangle$$
$$\langle \text{Идент} \rangle ::= \langle \text{Буква} \rangle \langle \text{Идент} \rangle \mid \langle \text{Буква} \rangle$$
$$\langle \text{Const} \rangle ::= \langle \text{Цифра} \rangle \langle \text{Const} \rangle \mid \langle \text{Цифра} \rangle$$

Вариант 3

```

<Программа> ::= <Объявление переменных> <Описание вычислений>
<Описание вычислений> ::= [ <Список присваиваний> ]
<Объявление переменных> ::= Var <Список переменных> ;
<Список переменных> ::= <Идент> | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                                <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> = <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                                <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= "-"
<Бин.оп.> ::= "-" | "+" | "*" | "/"
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= <Цифра> <Const> | <Цифра>

```

Вариант 4

```

<Программа> ::= <Объявление переменных> <Описание вычислений>
<Описание вычислений> ::= <Список присваиваний>
<Объявление переменных> ::= Var <Список переменных> ;
<Список переменных> ::= <Идент> | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                                <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> := <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                                <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= "-"
<Бин.оп.> ::= "-" | "+" | "*" | "/"
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= <Цифра> <Const> | <Цифра>

```


Вариант 7

```

<Программа> ::= <Объявление переменных> <Описание вычислений> .
<Описание вычислений> ::= Begin <Список присваиваний> End
<Объявление переменных> ::= Var <Список переменных> : Integer ;
<Список переменных> ::= <Идент> | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                           <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> = <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                  <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= "-"
<Бин.оп.> ::= "-" | "+" | "*" | "/"
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= <Цифра> <Const> | <Цифра>

```

Вариант 8

```

<Программа> ::= <Объявление переменных> <Описание вычислений> .
<Описание вычислений> ::= Begin <Список присваиваний> End
<Объявление переменных> ::= Var <Список переменных> : Logical ;
<Список переменных> ::= <Идент> | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                                <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> := <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                                <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= " ! "
<Бин.оп.> ::= " & " | " | " | " ^ "
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= 0 | 1

```


Вариант 9

```

<Программа> ::= <Объявление переменных> <Описание вычислений> .
<Описание вычислений> ::= Begin <Список присваиваний> End
<Объявление переменных> ::= Var <Список переменных> : Boolean ;
<Список переменных> ::= <Идент> | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                           <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> = <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                   <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= " .NOT . "
<Бин.оп.> ::= " .AND . " | " .OR . " | " .XOR . "
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= 0 | 1

```

Вариант 10

```

<Программа> ::= <Объявление переменных> <Описание вычислений>
<Описание вычислений> ::= Begin <Список присваиваний> End
<Объявление переменных> ::= Integer <Список переменных>
<Список переменных> ::= <Идент> ; | <Идент> , <Список переменных>
<Список присваиваний> ::= <Присваивание> |
                                <Присваивание> <Список присваиваний>
<Присваивание> ::= <Идент> := <Выражение> ;
<Выражение> ::= <Ун.оп.> <Подвыражение> | <Подвыражение>
<Подвыражение> ::= ( <Выражение> ) | <Операнд> |
                                <Подвыражение> <Бин.оп.> <Подвыражение>
<Ун.оп.> ::= "-"
<Бин.оп.> ::= "-" | "+" | "*" | "/"
<Операнд> ::= <Идент> | <Const>
<Идент> ::= <Буква> <Идент> | <Буква>
<Const> ::= <Цифра> <Const> | <Цифра>

```


ПРИЛОЖЕНИЕ Б. ТРЕБОВАНИЯ К КУРСОВОМУ ПРОЕКТУ

Основой заданий на курсовое проектирование являются задания, используемые в ходе лабораторных работ. С одной стороны это облегчает усвоение нового материала за счет выполнения студентами курсового проектирования параллельно с прохождением лабораторных работ. С другой стороны задания на курсовое проектирование содержат более сложные языковые конструкции, приближенные к конструкциям реальных языков программирования и проектирования высокого уровня.

В начале курсового проектирования студент должен модифицировать исходное описание языка (БНФ), выданное ему в качестве задания на лабораторные работы, добавив в него одну из конструкций языка высокого уровня (см. таблицу Б.1) в соответствии со своим вариантом.

Таблица Б.1

№ варианта	Конструкция языка
1.	IF <Выражение> THEN <Список операторов> {ELSE <Список операторов >} ENDIF
2.	FOR <Присваивание> TO <Выражение> DO <Список операторов> ENDFOR
3.	WHILE <Выражение> DO <Список операторов> ENDWHILE
4.	DO <Список операторов> WHILE <Выражение>
5.	CASE <Идентификатор> OF <Константа>: <Список операторов> {<Список альтернатив>} ENDCASE
6.	GO TO <метка>
7.	IF (<Выражение>) <Оператор> {ELSE <Оператор >}

№ варианта	Конструкция языка
8.	FOR <Присваивание> ТО <Выражение> <Оператор>
9.	WHILE (<Выражение>) <Оператор>

В вариантах 7-9 <Оператор> может быть простым или составным. Конструкции, ограничивающие составной оператор (begin – end, пара фигурных скобок и пр.) выбираются студентом.

Количество возможных сочетаний вариантов заданий для лабораторных работ и вариантов языковых конструкций достаточно велико, чтобы обеспечить неповторяющимися заданиями на курсовое проектирование одновременно несколько групп студентов. Для вариантов 8-9 заданий на лабораторные работы не рекомендуется добавлять конструкции 2-4 и 8-9 при выполнении курсового проекта, т.к. составление «разумных» программ с циклами в языке, где имеются только логические операции и типы данных едва ли возможно.

Целью курсового проектирования является разработка транслятора с ограниченного подмножества языка высокого уровня.

Основными техническими требованиями к курсовому проекту являются:

1. Язык является расширением варианта языка для лабораторных работ, включающим одну из языковых конструкций высокого уровня.
2. Транслятор реализуется по трехпроходной схеме.
3. Исходными данными для транслятора является текст программы на заданном языке.
4. Результатом работы транслятора должен быть файл, содержащий микрокод в заданной системе команд, либо текст программы на языке Ассемблера.
5. Для разработки транслятора рекомендуется применять язык программирования С или С++.

6. При написании транслятора использовать метод рекурсивного спуска (нисходящего разбора).

Пояснительная записка к курсовому проекту должна включать:

1. Описание синтаксиса реализуемого языка в форме Бэкуса-Наура.
2. LL(1)-грамматику языка.
3. Синтаксический граф языка.
4. Описание разработанного программного обеспечения, включающее
 - краткое описание лексического анализатора;
 - классы лексем различаемые лексическим анализатором;
 - примеры входного и выходного файлов для лексического анализатора;
 - краткое описание синтаксического анализатора (на уровне основных процедур и функций);
 - тестовые примеры обнаружения ошибок синтаксическим анализатором, а также пример работы без синтаксических ошибок;
 - краткое описание программы формирования постфиксной записи;
 - пример работы программы формирования постфиксной записи (входной файл лексем и выходной файл с постфиксной записью);
 - краткое описание программы генерации кода;
 - тестовый пример работы генератора кода (входной файл с постфиксной записью программы, выходной файл, содержащий микрокод в заданной системе команд или инструкции языка Ассемблера).

5. Приложение (распечатки разработанного программного обеспечения).

Тестовый пример, демонстрирующий правильность работы компилятора при корректном входном коде должен обязательно включать (кроме варианта с конструкцией GO TO) следующие элементы:

- оператор или группу операторов присваивания;
- конструкцию языка высокого уровня, содержащую вложенную конструкцию языка высокого уровня;
- конструкцию языка высокого уровня, внешнюю по отношению к предыдущим;

- оператор или группу операторов присваивания.

В случае оператора GO TO, помимо операторов присваивания, должно содержаться несколько переходов как вперед, так и назад.