Министерство образования и науки Российской Федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра компьютерных систем в управлении и проектировании (КСУП)

М.А. Песков, С.И. Борисов

Лингвистическое программное обеспечение САПР

Учебное пособие

2010

Корректор: Осипова Е.А.

Песков М.А. , Борисов С.И.
Лингвистическое программное обеспечение САПР: Учебное пособие / Под общей ред. М.А. Пескова. — Томск: Факультет дистанционного обучения, ТУСУР, 2010. — 108 с.

© Песков М.А., Борисов С.И., 2010
© Факультет дистанционного обучения, ТУСУР, 2010

ОГЛАВЛЕНИЕ

1 Введение....5
1.1 Общее понятие транслятора....5
1.2 Основные стадии компиляции и интерпретации....7
1.3 Анализ исходного текста....8
1.4 Группирование стадий.... 11

2 Языки и грамматики.... 13
2.1 Основные определения.... 13
2.2 Языки.... 14

3 Конечные автоматы (КА).... 19
3.1 Преобразование недетерминированного КА в детерминированный.... 21
3.2 Минимизация конечного автомата.... 22
3.3 Регулярные языки и регулярные выражения.... 23
3.4 Лемма о разрастании для регулярных языков.... 26

4 Поиск регулярных множеств.... 27
4.1 Поиск подцепочки.... 27

5 Генератор лексических анализаторов Lex.... 29
5.1 Как устроен LEX.... 34

6 Контекстно-свободные языки.... 36
6.1 Лемма о разрастании для КС-языков.... 38
6.2 Преобразование КС-грамматик.... 38
6.3 Алгоритм Кока—Янгера—Касами.... 41
6.4 LL(k) языки и грамматики.... 42
6.5 Простейший LL(1)-компилятор формул.... 45
6.6 Разбор снизу-вверх. Сдвиг-свертка. Простое и операторное предшествование.... 48
6.7 Грамматики простого предшествования.... 49
6.8 Грамматики операторного предшествования.... 51
6.9 Линеаризация матрицы предшествования.... 52
6.10 Еще один компилятор формул (операторное предшествование).... 53

7 LR(k)-грамматики. SLR(1), LALR(1)-грамматики 56
7.1 LR(0)-грамматики 56
7.2 LR(k)-грамматики 60
7.3 SLR(1)- и LALR(1)-грамматики 63

8 YACC 67
8.1 Структура YACC-программы 67
8.2 Разрешение конфликтов 68
8.3 Семантические действия 70
8.4 Семантический стек 71
8.5 Кодировка лексем и интерфейс 72
8.6 Обработка ошибок 73
8.7 Разное 74
8.8 Пример простейшего интерпретатора формул 75

9 Лабораторные работы по первой части курса 77
9.1 Лабораторная работа 1 77
9.2 Лабораторная работа 2 84

10 Лабораторные работы по второй части курса 87
10.1 Лабораторная работа 3 87
10.2 Лабораторная работа 4 93

11 Курсовой проект 96
11.1 Варианты курсовых проектов 97
11.2 Варианты синтаксиса 98

Литература 108

1 ВВЕДЕНИЕ

1.1 Общее понятие транслятора

Под *транслятором* понимают программу, переводящую текст на языке программирования (исходный текст) в форму, пригодную для исполнения, то есть на другой язык. Как правило (но далеко не всегда!), такой формой является последовательность машинных команд компьютера. Различают несколько типов трансляторов.

Под *компилятором* понимают программу, переводящую с языка высокого уровня на машинный язык. То есть, фактически, между исходным текстом программы и результатом ее прогона есть один файл — исполняемая программа. Примеров компиляторов можно привести множество: C, C++, Pascal, Ada, Fortran, Algol и т.д. Разнообразие компиляторов на первый взгляд может показаться ошеломляющим. Тем не менее большинство из них выполняет действия, общие для многих компиляторов, и поэтому строится на одних и тех же базовых принципах.

Под *интерпретатором* понимают программу, которая переводит с некоторого языка программирования в действия на вычислительной платформе. Они отличаются от компиляторов тем, что не выдают целевой код, а исполняют проанализированные команды. Например, такие языки, как Smalltalk, php, ruby, являются интерпретируемыми. К интерпретаторам также относятся командные оболочки операционных систем, например COMMAND.COM. Интерпретатором в известной степени является и процессор.

Прочитав вышеприведенные определения и вспомнив, как работает, например, среда разработки Turbo Pascal, на которой училось программировать не одно поколение сегодняшних программистов, можно усомниться в их справедливости. Действительно, по умолчанию, когда вы запускаете среду разработки Turbo Pascal и набираете текст программы, вы можете сразу запустить ее на исполнение и увидеть результат, и никакого исполняемого файла при этом не получается. Более того, вы можете пронаблюдать исполнение программы по шагам, что позволяет

думать, что Turbo Pascal — интерпретатор. Однако это не так. Просто Turbo Pascal по умолчанию компилирует не на диск, а в оперативную память (и делает это очень быстро), кроме того, любая приличная среда разработки, в том числе и среда Turbo Pascal, обладает хорошим отладчиком, который позволяет отлаживать скомпилированную программу по шагам. Но тем не менее это компилятор.

Текст некоторых языков компилируется не в машинный код, а в специализированный код, который затем интерпретируется; в качестве примера можно назвать Java или ранние версии Clipper'a и FoxPro. По аналогичной схеме работают все компиляторы для среды .NET. Этот код, кстати, может не интерпретироваться, а, опять же, компилироваться в машинный код при помощи *just-in-time* (JIT) компиляторов. То есть сначала исходный текст компилируется в текст на промежуточном языке, а затем этот текст еще раз компилируется уже в машинный код. Зачем нужны такие сложности — тема отдельного курса.

Первые компиляторы появились в начале 50-х. С тех пор теория и техника построения компиляторов существенно развились. Если создание первого компилятора с Фортрана потребовало 18 человеко-лет, то сейчас эта задача вполне под силу одному студенту в качестве курсового проекта.

Также выделяют еще одну группу программ — *конвертеры*. Данные программы переводят текст с одного языка на другой. Например, известны трансляторы с языка Pascal на C. Существует также множество языков, переводящих со специального языка на общераспространенные, например системы lex и yacc, которые будут рассмотрены в данном курсе, переводят программу со специального языка для записи грамматик в программу на языке C.

Конечно, хотелось бы предположить, что такие же трансляторы существуют и для естественных языков. Однако, к сожалению, все не так радужно. Работа трансляторов, например с английского на русский, основана приблизительно на тех же принципах, что и перевод с Pascal на C, но анализ этих языков крайне затруднен, поэтому стопроцентных решений здесь не существует и в данном курсе они не рассматриваются.

1.2 Основные стадии компиляции и интерпретации

Обычно выделяют следующие стадии компиляции:
- лексический анализ;
- синтаксический анализ;
- семантический анализ;
- генерирование промежуточного кода;
- оптимизация кода;
- генерирование целевого кода.

На всех стадиях компилятору приходится работать с различными таблицами, такими, как таблицы символьных имен, меток и так далее, а также обрабатывать ошибки компиляции.

На практике некоторые стадии могут объединяться в одну, некоторые — как, например, оптимизация — могут отсутствовать. Не всегда генерируется промежуточный код, в некоторых случаях можно сразу выдавать целевой код. Весь вышеприведенный процесс можно сгруппировать в две большие стадии — анализ (лексический, синтаксический и семантический) и синтез. На первой стадии проводится анализ структуры исходного текста и создается некоторое промежуточное представление программы. На второй стадии по этому представлению синтезируется целевой код. Нетрудно понять, что из этих двух стадий синтез более специализирован, больше привязан к конкретной программно-аппаратной платформе.

Интерпретатор работает практически аналогично, за тем исключением, что вместо процесса синтеза реализуется немедленное исполнение команд. То есть и в том и в другом случае есть общее — анализ, поэтому в данном курсе основной упор будет делаться именно на анализ. Более того, многие программы, формально не являющиеся компиляторами или интерпретаторами, выполняют анализ исходного текста. Можно привести в качестве примеров:

1. Структурные редакторы. В отличие от обычных текстовых редакторов, эти редакторы ориентированы на конкретный тип документов. Например, популярный редактор emacs (особенно в среде Unix-подобных операционных систем) при вводе ис-

ходных текстов программ на «известных» ему языках программирования сам делает необходимые отступы во вложенных блоках, выделяет разным цветом элементы программы, может проверять соответствие скобок и т.п. То же самое делают и все современные среды разработки программного обеспечения.

2. Утилиты для обработки текстов. Например, программа grep ищет в тексте фрагменты, которые могут иметь достаточно сложную структуру, а программа awk выполняет сложные преобразования текстовых файлов.

3. Программы форматирования текстов. В качестве примера можно назвать систему подготовки научных текстов TeX. При наборе в текст вставляются специальные команды форматирования; компилятор TeX выдаёт описание документа в двоичном формате, которое потом интерпретируется программами просмотра и печати. Сюда же можно отнести программы, работающие с такими форматами, как RTF или HTML, а также различные препроцессоры текстов программ.

4. «Силиконовые компиляторы». Современные интегральные схемы настолько сложны и содержат так много логических элементов, что их уже давно никто не проектирует вручную от начала до конца. Вместо этого логика схемы описывается на специализированном языке высокого уровня, после чего специальные автоматы по этому описанию изготовляют матрицы для схем.

1.3 Анализ исходного текста

Можно выделить три основные стадии анализа исходного текста:

1. Лексический анализ, или сканирование. Поток символов читается последовательно и разбивается на лексические единицы, или лексемы (tokens). Рассмотрим анализ следующего оператора Паскаля:

position := initial + rate * 60

Можно выделить следующие лексемы:

Идентификатор position.

Символ присваивания := .

Идентификатор initial.

Знак + .

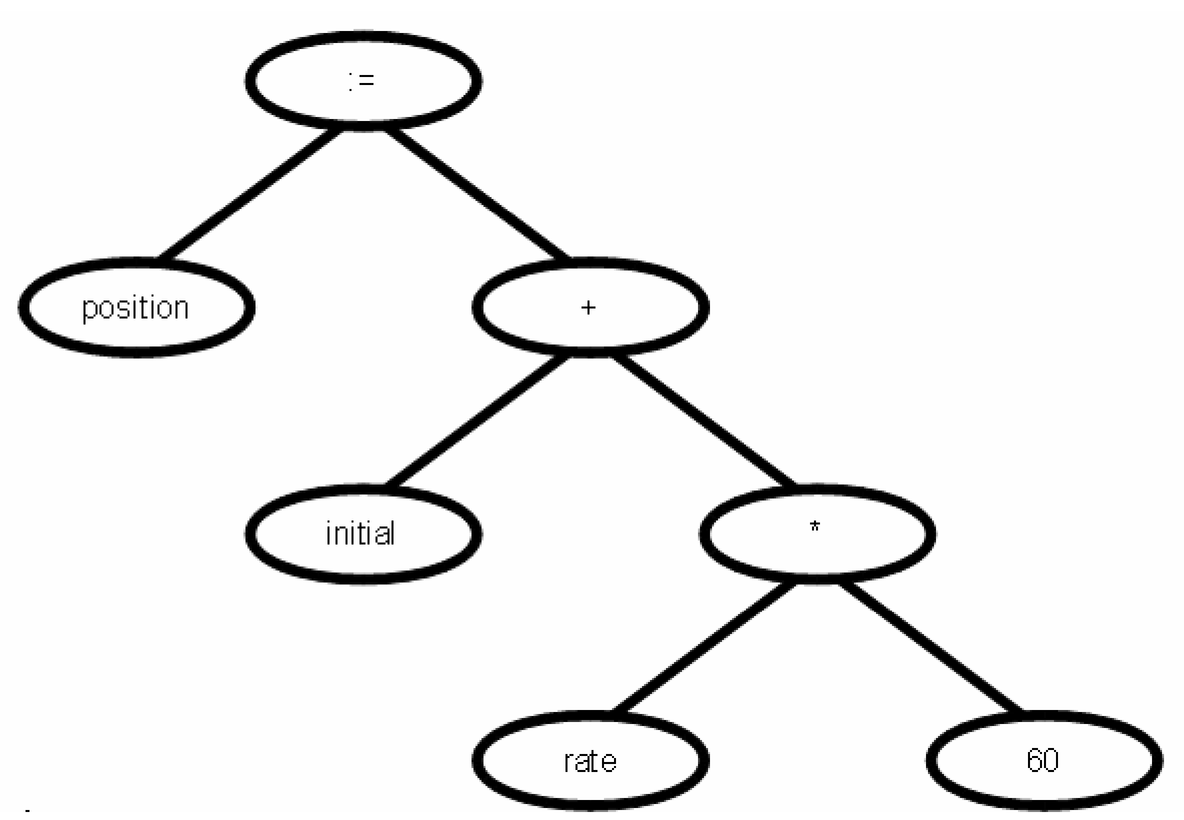
Идентификатор rate.

Знак * .

Число 60.

Разделяющие пробелы обычно выкидываются.

2. Синтаксический анализ, или разбор. Поток лексем группируется в иерархическую структуру, помогающую компилятору синтезировать выходной код. Как правило, это синтаксическое дерево наподобие следующего:



Такие структуры отражают логику выражений — например, тот факт, что сначала выполняется умножение rate на 60, а потом сложение.

3. Семантический анализ. На этой стадии программа проверяется на семантическую корректность и собирается информация для следующих этапов. Наиболее важная часть семантического

анализа — проверка типов: соответствуют ли типы операндов в каждом операторе спецификации языка. При этом компилятор может модифицировать синтаксические деревья, вставляя преобразования типов. Если в вышеприведенном примере переменная rate имеет тип real, то целое число 60 будет преобразовано к real.

Ряд действий может быть выполнен как на стадии лексического анализа, так и синтаксического. Обычно выбирают вариант, который делает проще анализ в целом. Например, в языке Pascal или в C информацию о типе переменной можно выделить уже на стадии лексического анализа, в то время как в PL/I такое возможно только при синтаксическом анализе (синтаксис данного языка переусложненный).

Первые три стадии уже рассмотрены в предыдущем параграфе; они составляют этап анализа. При этом не только формируется синтаксическое дерево, но и собирается информация, которая заносится в таблицы. В рассматриваемом примере выясняется, что в выражении участвуют переменные position, initial и rate. Во многих языках они к этому моменту должны быть описаны и соответственно включены в таблицу переменных. Для генерации кода существенны не имена переменных, а их адреса.

На долю обработчика ошибок выпадает определить местоположение ошибки, установить характер ошибки (а не просто сказать «произошла ошибка»), выдать соответствующее сообщение и попытаться как-то избавиться от ошибочной конструкции, чтобы продолжить анализ дальше.

Промежуточный код можно рассматривать как программу для некоторой абстрактной машины. (В некоторых случаях разработчики компилятора даже моделируют эту машину, чтобы проверять качество промежуточного кода.) Он должен обладать следующими свойствами: его должно быть легко генерировать и по нему должно быть легко строить целевой код.

Цель оптимизации — улучшить промежуточный код по некоторым параметрам (обычно по размеру либо по скорости). Некоторые оптимизации выглядят почти тривиально, но тем не менее способны существенно улучшить код. Разные компиляторы оптимизируют по-разному. Те из них, которые уделяют оптимизации значительное время, принято называть оптимизирующими

компиляторами. Многие компиляторы позволяют управлять процессом оптимизации.

Генерирование целевого кода — завершающая стадия компиляции. Переменным присваиваются конкретные адреса, при этом выбирается, какие переменные будут размещены в регистрах. Потом промежуточный код транслируется в целевой.

1.4 Группирование стадий

Обычно стадии объединены в две большие части, анализатор и синтезатор, как и было сказано выше. Анализатор выполняет стадии анализа и создает промежуточный код; эта часть зависит в основном от исходного языка и мало зависит (или совсем не зависит) от целевого. На этом этапе создаются таблицы символов, диагностируется основная часть ошибок и производится оптимизация кода.

Синтезатор практически не зависит от исходного языка. На этом этапе генерируется и оптимизируется целевой код.

Такое разделение наводит на идею семейств компиляторов. Предположим, у нас есть N языков и M машин. Вместо того, чтобы разрабатывать N*M компиляторов для каждой пары язык-машина, мы можем разработать единый промежуточный язык и потом комбинировать N анализаторов и M синтезаторов. Если появляется новый язык, достаточно написать один анализатор; для новой машины достаточно написать один синтезатор.

Известно несколько примеров реализации такого подхода. Например, в прошлом был весьма популярен пакет TopSpeed. В целом же данный подход имеет довольно ограниченное применение. Дело в том, что во многих случаях специализированные компиляторы, ориентированные на конкретный язык и конкретную платформу, выдают гораздо более эффективный код, чем «составные». Применение «универсального» промежуточного языка не позволяет использовать особенности конкретной машины; кроме того, исходный язык может включать в себя средства, сильно зависящие от машины: управление памятью, обработку прерываний, многопроцессность и т.п.

Идея «составных» компиляторов впервые была выдвинута ещё в 50-х годах. Тогда же образовался комитет по выработке универсального промежуточного языка UNCOL. Деятельность комитета успеха не имела.

Несколько успешнее эту проблему решила компания Microsoft в платформе .NET: единая платформа исполнения и промежуточный язык с компилятором и оптимизатором плюс множество компиляторов языков программирования, переводящих программы на данный промежуточный язык.

В данном курсе мы совсем не будем касаться оптимизации кода, поэтому не будем генерировать и промежуточный код, а генерация целевого кода будет осуществляться не для реального компьютерного процессора, а для некоторой виртуальной машины. Впрочем, это можно рассматривать с другой стороны — мы не будем изучать оптимизацию и генерацию целевого кода, а остановимся на генерации промежуточного кода для виртуальной машины, которая у нас будет реализована.

2 ЯЗЫКИ И ГРАММАТИКИ

2.1 Основные определения

Алфавит — любое множество символов. Будем обозначать данное множество знаком $\sum$ — большая греческая буква Сигма (в некоторой литературе можно встретить другое обозначение, например большая латинская буква Е). Понятие символа не определяется. Вообще говоря, под символом может пониматься что угодно, буква алфавита, рисунок, звук и т.д. Однако в данном курсе под символом мы будем понимать то, что можно легко представить и однозначно идентифицировать при помощи компьютера — символы из таблицы символов компьютера.

Цепочка символов — есть последовательность этих символов. Например, последовательность трех символов — 0, 1 и 2. Данную цепочку будем записывать как «012» или 012. Другие обозначения, которые будут использоваться в данном курсе:

x^R	цепочка x с символами в обратном порядке
x^n	цепочка x, повторенная n раз
$x*$¿	цепочка x, повторенная 0 или более раз
$x+$¿	цепочка x, повторенная 1 или более раз
xy	сцепление (конкатенация) цепочек x и y
x	длина (число символов) цепочки x
	пустая цепочка

Цепочку из одного символа будем обозначать самим символом. Буквы *x, y, z, v, w, t* будем применять для обозначения цепочек. Множество всех цепочек из элементов множества $\sum$ естественно обозначить через Е∗.

2.2 Языки

С учётом вышеизложенных определений язык — это есть подмножество Е∗. Примеры языков:

– Си;

– Русский;

– $0^{n}1^{n}$ | n i 0.

Язык можно задать одним из следующих способов:

– перечислив все цепочки;

– написав программу-распознаватель, которая получает на вход цепочку символов и выдает ответ «да», если цепочка принадлежит языку, и «нет» в противном случае;

– с помощью механизма порождения — грамматики.

Под грамматикой понимается следующая четверка:

$$(E,N,S,P),$$

где *E* — множество символов алфавита (или терминальных символов). Будем обозначать их строчными символами алфавита и цифрами.

N — множество метасимволов (или нетерминальных символов), не пересекающееся с Е.

S — специально выделенный начальный символ. Будем обозначать такие символы прописными буквами.

P — множество правил вывода, определяющих правила подстановки для цепочек. Каждое правило состоит из двух цепочек (например, *x* и *y*), причем *x* должна содержать по крайней мере один нетерминал; и означает, что цепочку *x* в процессе вывода можно заменить на *y*. Вывод цепочек языка начинается с нетерминала *S*. Правило грамматики будем записывать в виде *x* *y*. Также употребляется запись *x*::=*y* или *x*:*y*.

Более строго, определим понятие выводимой цепочки:

– пусть *S* — выводимая цепочка;

– если *xyz* — выводимая цепочка и в грамматике имеется правило *x* *y*, то *xtz* — выводимая цепочка;

– определяемый грамматикой язык состоит из выводимых цепочек, содержащих только терминальные символы.

Примеры:

а) S S $0S1$

б) S S (S) S SS

Для сокращения записи принято использовать символ «или» — "|". Короткая форма записи предыдущих примеров:

а) S $0S1$

б) S (S) SS

Более сложный пример:

в) S $aSBC$ abC

Эта грамматика порождает язык $a^n b^n c^n$ $n>0$. Докажем этот факт индуктивно. Пронумеруем правила следующим образом:

1. S $aSBC$

2. S abC

3. CB BC

4. bB bb

5. cC cc

6. bC bc

Построим вывод для n=1. Строка при этом будет иметь следующий вид: abc. В процессе вывода будем обозначать правила, по которым производилось преобразование в следующем виде: $x \xrightarrow{k} y$, что означает, что строка x будет преобразована в строку y по правилу № k.

$S \xrightarrow{2} abC \xrightarrow{6} abc$, что и требовалось доказать. Пока что не говорим, почему мы выбрали в том или другом случае именно это правило. На самом деле, альтернатива была только на первом шаге.

Построим вывод для n=2. Строка при этом будет иметь следующий вид: $aabbcc$.

$S \xrightarrow{1} aSBC \xrightarrow{2} aabCBC \xrightarrow{3} aabBCC \xrightarrow{4} aabbCC \xrightarrow{6} aabbc$
$C \xrightarrow{5} Aaabbcc$, что и требовалось доказать.

Внимательно изучите данный вывод и уясните для себя, какие именно символы или пары символов менялись по указанному правилу на каждом шаге.

Покажем, что аналогичный алгоритм можно использовать для любого n. На втором шаге вместо использования правила № 2 можно было вторично воспользоваться правилом № 1 для симво-

ла S. В этом случае подстрока *aSBC aaSBBC* развернется до и далее до *aaabCBBC*, это приведет к тому, что шаги с 3-го по 6-й придется повторить еще раз и строка *aaabbbccc* также будет выводима из данной грамматики. Воспользовавшись на 2-м шаге правилом № 1 большее количество раз, можно вывести строку $a^n b^n c^n$ для любого *n*.

Грамматики, в свою очередь, образуют т.н. метаязык. Выше была описана «академическая» форма записи метаязыка. На практике также часто применяется другая форма записи, традиционно называемая нормальными формами Бэкуса—Наура (НФБН). Терминалы в НФБН записываются как обычные символы алфавита, а нетерминалы — как имена в угловых скобках <>. Например, грамматику целых чисел без знака можно записать в виде:

```
<число> : <цифра> | <цифра> <число>
<цифра> : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Грамматику метаязыка также можно записать на нем самом.

Рассмотрим простейший язык арифметических формул:

```
<формула> : (<формула>) | <число> | <формула><знак><формула>
<знак> : + | *
```

Почему `«3+5*2»` является формулой? Приведем последовательность преобразований цепочек (так называемый «разбор» или «вывод»):

```
<формула>                                                :
<формула>  <знак>  <формула>                             :
<формула>  <знак>  <формула>  <знак>  <формула>          :
<число>    <знак>  <формула>  <знак>  <формула>          :
3          <знак>  <формула>  <знак>  <формула>          :
3          +       <формула>  <знак>  <формула>          :
3          +       <число>    <знак>  <формула>          :
3          +       5          <знак>  <формула>          :
3          +       5          *       <формула>          :
3          +       5          *       <число>            :
3          +       5          *       2                  :
```

Сокращенно наличие вывода (цепочки преобразований) будем записывать в виде `<формула>::3+5*2`. Большинство грамматик допускают несколько различных выводов для одной и той же цепочки из языка. Другой вывод для цепочки «3+5*2» будет выглядеть следующим образом:

```
<формула>                                                   :
<формула>                        <знак>     <формула>       :
<формула>   <знак>   <формула>   <знак>     <формула>       :
<формула>   <знак>   <формула>   <знак>     <число>         :
<формула>   <знак>   <формула>   <знак>     2               :
<формула>   <знак>   <формула>   *          2               :
<формула>   <знак>   <число>     *          2               :
<формула>   <знак>   5           *          2               :
<формула>   +        5           *          2               :
<число>     +        5           *          2               :
3           +        5           *          2
```

Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен левый вывод цепочки. Аналогично определяется правый вывод. В предыдущих примерах построен сперва левый, а затем правый вывод.

Изобразим выполняемые замены цепочек в виде т.н. «дерева разбора» (или дерева вывода). По традиции дерево изображается «вверх ногами», так, как показано на рисунке 1.

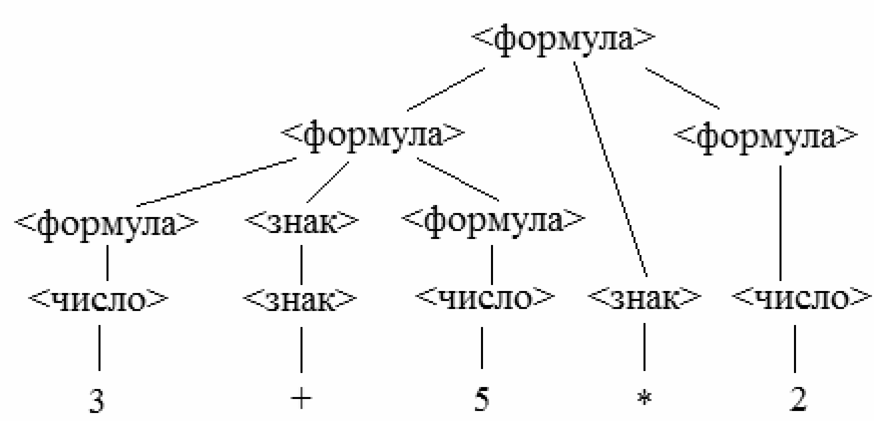


Рисунок 2.1 — Пример дерева разбора

Нарисованное дерево имеет ветви (линии) и узлы (помечены терминалами и нетерминалами), из которых растут ветви. Конечные узлы (терминалы) называются листьями. Понятия «поддерево», «корень дерева», видимо, не нуждаются в определении.

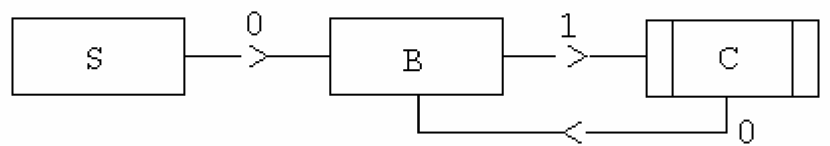
3 КОНЕЧНЫЕ АВТОМАТЫ (КА)

Рассмотрим другой способ задания регулярных языков. (Недетерминированный) конечный автомат задается:

- алфавитом входных символов E;
- множеством состояний S;
- тернарным отношением переходов на множестве S, E U e, S;
- начальным состоянием — выделенным состоянием в S;
- конечными состояниями — непустым подмножеством S.

Принято изображать автомат в виде ориентированного графа, узлы которого соответствуют состояниям (конечные состояния мы будем заключать в двойную рамку), а ребра, помеченные символами входного алфавита или e, изображают отношение переходов.

Цепочка допускается автоматом если и только если существует из начального в одно из конечных состояний, такой, что, прочитав метки ребер вдоль этого пути, мы получим исходную цепочку (буква e, естественно, не читается). Например, автомат



допускает цепочки (01)+. Этот язык можно описать регулярной грамматикой:

S : 0 B B : 1 C C : 0 B C : e

Несложно показать, что для каждого автомата можно построить регулярную грамматику, описывающую тот же самый язык, и наоборот.

Детерминированный конечный автомат — частный случай недетерминированного, в котором:

- нет e-переходов,
- отношение переходов является однозначной функцией

f:(S x E U e) -> S,

определенной, может быть, не для всех пар из SxEUe. В терминах графа это означает, что из одного состояния выходит не более одного ребра с одинаковой меткой.

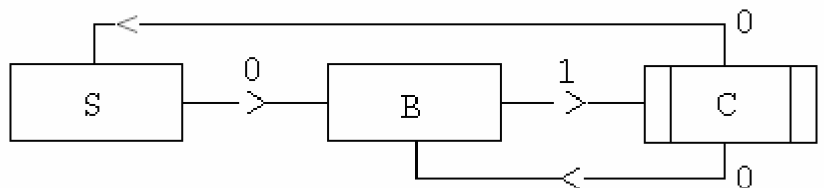
Для детерминированного автомата очень просто проверять принадлежность цепочки из E* языку. Добавив, если нужно, еще одно «тупиковое» состояние, можно сделать функцию переходов определенной на всех парах из S x E.

	0	1
S	B	F
D	F	C
C	B	F
F	F	F

```
s := начальное состояние
цикл для каждого символа цепочки c выполнить
. s := f( s, c )
конец цикла
ответ := s - конечное состояние
```

Такая интерпретация детерминированного конечного автомата более наглядна и общепринята: КА — устройство, которое может находиться в конечном множестве состояний и переходит из одного состояния в другое под действием внешних событий из алфавита E.

Можно ли подобным образом интерпретировать недетерминированный автомат (или хотя бы эффективно определять принадлежность цепочки языку)? Да, можно считать, что в том случае, когда возможен более чем один переход, создается необходимое число экземпляров КА, которые переводятся во все возможные в этой ситуации состояния. Цепочка считается допущенной, если хотя бы один из экземпляров оказался в конечном состоянии.



Заметим, что если несколько экземпляров недетерминированного КА оказались в одном состоянии, то в дальнейшем можно рассматривать только один из них. Таким образом, максимальное количество экземпляров не превосходит числа состояний.

Проверить, допускает ли недетерминированный конечный автомат цепочку символов, также несложно:

```
S := e-замыкание( начальное состояние )
цикл для каждого символа цепочки c выполнить
. S := e-замыкание( F( S, c ) )
конец цикла
ответ := ( S П конечные состояния ) — непусто
```

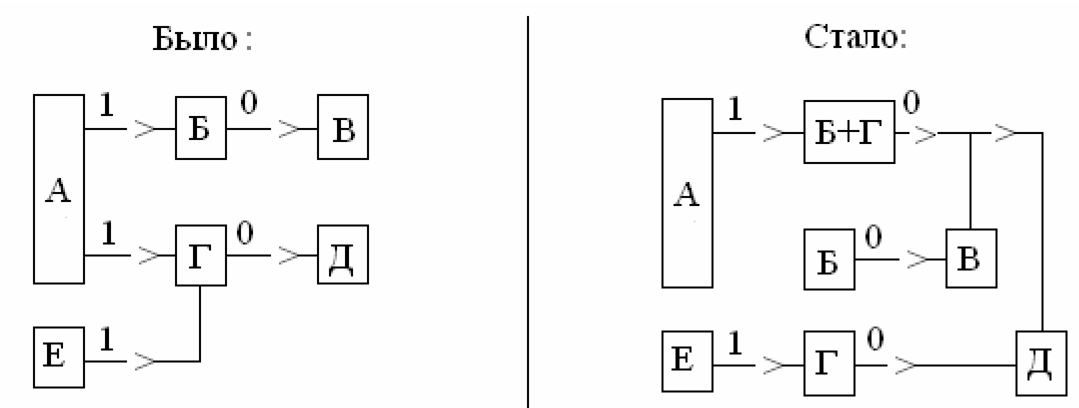
Здесь: S — подмножество множества состояний КА;
e-замыкание(S) — множество состояний, достижимых из S за 0 и более e-переходов;
F(S, c) — множество состояний, достижимых из S за один переход, помеченный символом c.

3.1 Преобразование недетерминированного КА в детерминированный

Недетерминированный КА всегда можно преобразовать в эквивалентный (т.е. допускающий то же множество цепочек) детерминированный КА. Рассмотрим новый КА, состояниями которого будут подмножества множества состояний исходного КА (если исходный автомат имел m состояний). Исходным для нового автомата будет состояние Начало, конечными — все состояния, содержащие исходное конечное состояние. Переход A -> B по символу d имеется в новом автомате тогда и только тогда, когда в исходном автомате для любого состояния b из B существует a из A такое, что по символу d возможен переход a -> b, и других переходов по d из A нет. Новый автомат будет детерминированным и эквивалентным исходному.

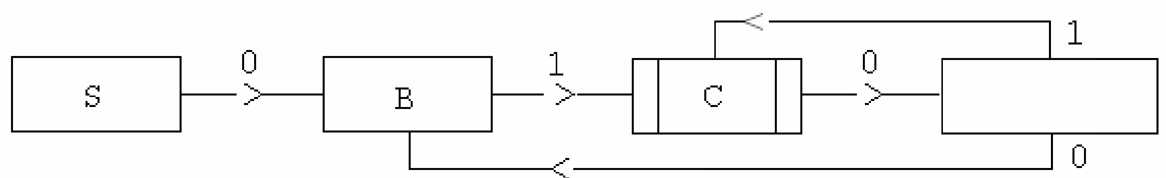
Действительно, состояние нового автомата a+b соответствует размещению экземпляров исходного недетерминированного КА в состояниях a и b, а переход в новом автомате соответствует переходам всех экземпляров недетерминированного КА.

Для практических целей применяется аналогичный алгоритм: назовем состояния неразличимыми, если в них происходит переход из одного и того же состояния при одном и том же входном символе. Возьмем любое множество неразличимых состояний и добавим в список состояний КА еще одно — их «сумму», перемещая переходы:



Получим новый (быть может, вновь недетерминированный) КА. Будем повторять наши действия, пока в КА остаются неразличимые состояния. Этот процесс в конце концов завершится. При этом мы получим детерминированный автомат, эквивалентный исходному.

Применив этот метод к недетерминированному, получим:

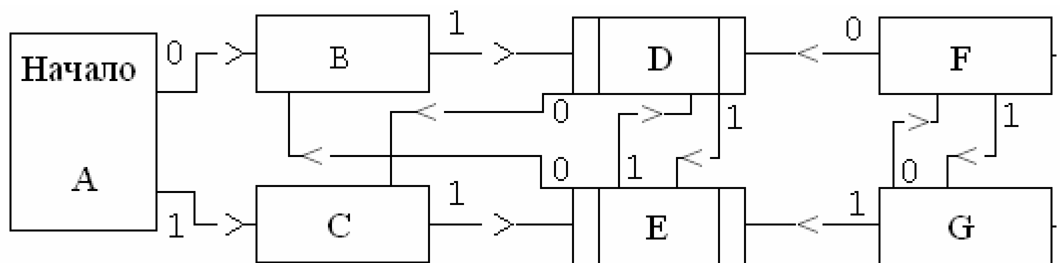


3.2 Минимизация конечного автомата

По конечному автомату часто можно построить автомат с меньшим числом состояний, эквивалентный исходному. Соответствующий процесс называется минимизацией конечного автомата. Вначале выбросим из автомата все состояния, недостижимые из начального. Затем разобьем все состояния КА на классы эквивалентности следующим способом: в первый класс отнесем все конечные состояния, а во второй — все остальные. Назовем эти состояния 0-эквивалентными. Теперь построим новое 1-эквивалентное разбиение, выделив те состояния, которые по одинаковым символам переходят в 0-эквивалентные состояния. Последовательно строя n+1-эквивалентные состояния по n-эквивалентным, мы будем увеличивать число классов эквивалентности. Прекратим этот процесс тогда, когда n+1-эквивалентное состоя-

ние совпадет с n-эквивалентным. Каждый полученный класс эквивалентности и будет состоянием нового минимизированного КА, эквивалентного исходному.

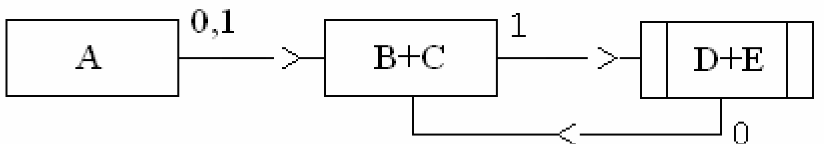
Рассмотрим, например, следующий КА:



Состояния F и G недостижимы (это можно выяснить, например, вычислив транзитивное замыкание отношения «есть переход»). Построим классы n-эквивалентности для оставшихся состояний:

n	Классы
0	(ABC) (DE)
1	(A) (BC) (DE)
2	(A) (BC) (DE)

Результат:



3.3 Регулярные языки и регулярные выражения

Рассмотрим специальный класс операций над языками — регулярные операции, множество языков, получаемое в результате применения конечного числа регулярных операций из элементарных языков, — регулярные множества, способ их описания — регулярные выражения и недетерминированные конечные автоматы, допускающие цепочки из этих множеств.

Регулярное множество	Регулярное выражение	Конечный автомат
Пустая цепочка	e	S —e→ E
Один символ a из E	a	S —a→ E
P Q	p¦q	S —e→ s Авт. P e —e→ E; S —e→ s Авт. Q e —e→ E
PQ (конкатенация)	pq	S Авт. P es Авт. Q E
P* (итерация)	p*	S —e→ s Авт. P e —e→ E; e (возврат); e (S → E)
P (просто скобки)	(p)	S Авт. P E

В регулярном выражении «*» имеет больший приоритет, чем конкатенация, а конкатенация — больший, чем «|». Примеры регулярных выражений: (0|1)*, (0|1)(0|1)*. Какие множества они описывают?

Для регулярного выражения предложен способ построения недетерминированного конечного автомата, допускающего соответствующее выражению регулярное множество. Предложенная конструкция не является самой экономной (автомат обычно содержит много «лишних» e-переходов), однако построенный автомат обладает следующими полезными свойствами:

- у него только одно конечное состояние,
- в начальное состояние не входит ни одно ребро,
- из конечного состояния не выходит ни одно ребро.

Таким образом, мы доказали, что регулярные множества < регулярные языки = языки, допускаемые КА. Покажем, что до-

пускаемое КА множество — регулярно. (Это утверждение называется теоремой Клини.)

Доказательство: Пусть S1,...Sn — состояния детерминированного КА, S1 — начальное состояние. Рассмотрим все пути в графе переходов с началом в Si, концом в Sj, промежуточными узлами из множества S1...Sk (1<=i<=n, 1<=j<=n, 0<=k<=n) и множества цепочек из E — L(i,j,k), соответствующие этим путям. Докажем индукцией по k, что множества L — регулярны.

L(i,j,0) — состоит из меток ребер, ведущих из Si в Sj, следовательно, регулярно.

Для 0<=k<=n-1

L(i,j,k+1) = L(i,j,k) U L(i,k+1,k) L(k+1,k+1,k)* L(k+1,j,k) получено с помощью регулярных операций из регулярных множеств, следовательно, регулярно.

Множество цепочек, допускаемых КА, представляет собой объединение цепочек L(1,j,n), таких, что Sj — конечное состояние КА, следовательно, регулярно. (Конец доказательства.)

Мы рассмотрели 4 способа описания языков:

- регулярные (автоматные, праволинейные) грамматики,
- недетерминированные КА,
- детерминированные КА,
- регулярные выражения,

и показали, что они описывают один класс языков — регулярные языки. Этот класс языков «устроен очень хорошо» — для всех типичных вопросов известны ответы и эффективные алгоритмы. Примеры таких вопросов:

- является ли объединение, пересечение, дополнение регулярных языков регулярным,
- является ли регулярный язык конечным, пустым,
- совпадают ли два регулярных языка,
- является ли один регулярный язык подмножеством другого и т.д.

(Замечание. Для других классов иерархии Хомского дела обстоят значительно хуже, например проблема эквивалентности для КС-языков алгоритмически неразрешима.)

3.4 Лемма о разрастании для регулярных языков

Так называмые «леммы о разрастании» для классов языков — удобный способ доказательства того, что конкретный язык не относится к данному классу.

Лемма для регулярного языка: существует такое число m, что любую цепочку x, |x| > m, принадлежащую языку, можно разбить на три части: x = uvw так, что 0<|v|<=m и цепочки uv*w также будут принадлежать языку. Доказательство: построим КА для распознавания языка. Пусть этот автомат имеет m состояний. Тогда при разборе цепочки x хотя бы одно из состояний (например, A) проходится дважды. Разобьем цепочку x на три части — до состояния A, от A до A, остальное. Это и есть цепочки u,v.

4 ПОИСК РЕГУЛЯРНЫХ МНОЖЕСТВ

Рассмотрим задачу поиска регулярного множества R в цепочке символов L. (Вряд ли она часто встречается при реализации компиляторов, но лежит в непосредственной близости от рассматриваемых вопросов и весьма популярна. Так, например, в системе UNIX имеется по крайней мере три программы для решения этой задачи: grep, egrep, fgrep — Get Regular Expression Pattern.)

Эта задача, очевидно, эквивалентна задаче принадлежности цепочки L регулярному множеству (.*)(R)(.*) и может быть решена построением детерминированного или недетерминированного конечного автомата и применением его к цепочке L. Какой из автоматов лучше? Ответ «конечно, детерминированный, т.к. он работает быстрее», в общем случае, неверен. Проблема заключается в том, что существуют семейства регулярных выражений, для которых число состояний минимального ДКА экспоненциально зависит от длины выражения, в то время как число состояний НКА, полученного прямо из регулярного выражения, линейно зависит от его длины. Это делает невозможным применение ДКА для поиска некоторых регулярных множеств, описываемых регулярными выражениями весьма умеренной длины.

Для таких приложений более эффективной чем НКА может оказаться так называемая «ленивая» техника. Она основана на тех же самых идеях, что и виртуальная или cache-память. В этом случае ДКА строится в процессе просмотра цепочки, причем вычисляются и заносятся в cache-буфер только необходимые для данной цепочки состояния и переходы ДКА. По-видимому, можно придумать класс задач, на которых этот метод будет эффективнее ДКА (большой, требующий много времени для вычисления ДКА, в котором реально используется небольшое подмножество).

4.1 Поиск подцепочки

Полученные результаты, примененные к задаче поиска подстроки R (частный случай регулярного выражения) в последовательности символов L дают неплохой результат: ДКА требует

время O(|L|+|R|) + время построения ДКА, что для длинных последовательностей лучше наивного алгоритма, который в худшем случае требует O(|L|*|R|). Весьма неожиданный результат, учитывая неадекватно сложные методы для столь элементарной задачи.

Вполне естественная задача — придумать столь же эффективный алгоритм поиска подстроки, не требующий трудоемкого процесса построения ДКА. Одно из решений — алгоритм Кнута—Морриса—Пратта.

Предобработка: для образца поиска R[1:k] вычисляется так называемая функция возвратов f[i] — длина самого длинного собственного суффикса цепочки R[1:i], совпадающего с началом R. Другими словами, f[i]=j 1<=i<=k, если и только если j (0<=j<i) — максимальное, такое, что R[1:j] = R[i-j+1:i]. Например:

	1	2	3	4	5	6
R	a	b	a	b	a	c
f	0	0	1	2	3	0

1 2 3 4 5 6

/* Вычисление функции возвратов f[1..k] для цепочки r[1..k] */

```
for( s=0, f[1]=0, i=1; i<k; i++ ) {
 while( s>0 && r[i+1]!=r[s+1] ) s=f[s];
 f[i+1] = r[i+1]==r[s+1] ? ++s : 0;
}
```

5 ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ LEX

Lex — программа для генерации сканеров (лексических анализаторов). Входной язык содержит описания лексем в терминах регулярных выражений. Результатом работы LEX'а является программа на языке Си, которая читает файл yyin (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие регулярным выражениям.

Рассмотрим способы записи регулярных выражений во входном языке Lex'a. Алфавит Lex'a совпадает с алфавитом используемой ЭВМ. Символ из алфавита, естественно, представляет регулярное выражение из одного символа. Специальные символы (в том числе +-*?()[]|/\^$.<>) записываются после специального префикса \. Кроме того, применимы все традиционные способы кодирования символа в языке С. Символы и цепочки можно брать в кавычки: Например:

a "a" \a — три способа кодирования символа a

\n \t \007 "continue"

Имеется возможность задания класса символов:

[0-9] или [0123456789]	— любая цифра
[A-Za-z]	— любая буква
[^0-7]	— любой символ, кроме восьмиричных цифр
.	— любой символ, кроме \n

Грамматика для записи регулярных выражений (в порядке убывания приоритета):

<p> : <p>*	— повторение 0 или более раз
<p> : <p>+	— повторение 1 или более раз
<p> : <p>?	— необязательный фрагмент
<p> : <p><p>	— конкатенация
<p> : <p>m,n	— повторение от m до n раз

<p> : <p>m	— повторение m раз
<p> : <p>m,	— повторение m или более раз
<p> : ^<p>	— фрагмент в начале строки
<p> : <p>$	— фрагмент в конце строки
<p> : <p>|<p>	— любое из выражений
<p> : <p>/<p>	— первое выражение, если за ним следует второе
<p> : (p)	— скобки, используются для группировки

Примеры:

[A-Za-z]([A-Za-z0-9]0,7)	— идентификатор (имя) в языке С
^#" "*define	— начало #define в языке С

Программа на входном языке Lex состоит из трех частей, разделенных символами %%:

Описания

%%

Правила

%%

Программы

Раздел описаний содержит определения макросимволов (метасимволов) в виде:

ИМЯ ВЫРАЖЕНИЕ

Если в последующем тексте в регулярном выражении встречается ИМЯ, то оно заменяется на ВЫРАЖЕНИЕ. Если строка описаний начинается с пробелов или заключена в скобки % ... %, то она просто копируется в выходной файл.

Раздел правил содержит строки вида

ВЫРАЖЕНИЕ ДЕЙСТВИЕ

ДЕЙСТВИЕ — это фрагмент программы на С, который выполняется тогда, когда обнаружена цепочка, соответствующая ВЫРАЖЕНИЮ. Действие, указанное в начале раздела без выражения, выполняется до начала разбора. Lex делает попытку выделить наиболее длинную цепочку из входного потока. Если несколько правил дают цепочку одинаковой длины, применяется первое правило. Так, при разборе по следующим правилам для цепочки «123» будет применено первое правило, а для цепочки «123.» — третье:

```
[0-9]+
(\+|\-)?[0-9]+
(\+|\-)?[0-9]+"."[0-9]+
```

Если ни одно из правил не удастся применить, входной символ будет скопирован в yyout. Если это нежелательно, в конец правил можно добавить, например, строки:

```
. { /* Ничего не делать */}
\n { }
```

Раздел программ может содержать произвольные тексты на С и целиком копируется в выходной файл. Обычно здесь записывается функция yywrap(), которая определяет, что делать при достижении автоматом конца входного файла. Ненулевое возвращаемое значение приводит к завершению разбора, нулевое — к продолжению (перед продолжением, естественно, надо открыть какой-нибудь файл как yyin).

Интерпретатор таблиц КА имеет имя yylex(). Автомат прекращает работу (происходит возврат из функции yylex()), если в одном из действий выполнен оператор return (результат yylex() будет совпадать с указанным в операторе) или достигнут конец файла и значение yywrap() отлично от 0 (результат yylex() будет равен 0).

Традиционный пример входной программы для Lex'a — подсчет числа слов и строк в файле:

```
    /**************** Раздел определений ********************/
    /* NODELIM означает любой символ, кроме разделителей слов */
NODELIM         [^" "\t\n]
                int l, w, c;          /* Число строк, слов, символов */

%% /******************* Раздел правил *********************/

                { l=w=c=0;            /* Инициализация */ }
NODELIM         { + w++; c+=yyleng;   /* Слово */ }
\n              { l++;                /* Перевод строки */ }
.               { c++;                /* Остальные символы */ }

%% /****************** Раздел программ ********************/

main() { yylex(); }

yywrap() {
    printf( " Lines - %d Words - %d Chars - %d\n", l, w, c );
    return( 1 );
}
```

Внутри действий в правилах можно использовать некоторые специальные конструкции и функции Lex'a:

yytext — указатель на отождествленную цепочку символов, терминированную нулем;

yyleng — длина этой цепочки

yyless(n) — вернуть последние n символов цепочки обратно во входной поток;

yymore() — считать следующие символы в буфер yytext после текущей цепочки

yyunput(c) — поместить байт с во входной поток

ECHO — копировать текущую цепочку в yyout

Программа для Lex'а, которая печатает все слова с переносами из входного потока:

```
NODELIM [^, :\.;\n\-\s]
HYPHEN [\-\s]
%%
{ NODELIM }+{ HYPHEN }\n{ NODELIM }+ {printf ("%s\n",yytext); }
{ NODELIM }*                          { /* Необязательно */ }
. | \n                                { }
%%
yywrap() { return(1); }
main() { yylex(); }
```

Если убрать необязательное правило из предыдущей программы, программа по-прежнему будет работать, но значительно медленнее, поскольку при этом механизм вызова действий будет вызываться для каждого символа (а не каждого слова).

В некоторых случаях бывает удобно описать необходимые действия в терминах нескольких разных состояний (т.е. разных конечных автоматов) с явным переключением с одного на другое. В этом случае набор имен состояний следует перечислить в специальной строке %Start, а перед выражениями записывать имя соответствующего состояния в угловых скобках. Переключение в новое состояние выполняется с помощью оператора BEGIN. Например, следующая программа удаляет комментарии из программ на С (out — вне комментария, in — внутри):

```
%Start out in
%%
                { BEGIN out; }
<out>"/*"       { BEGIN in; }
<out>.|\n       { printf("%s",yytext); }
```

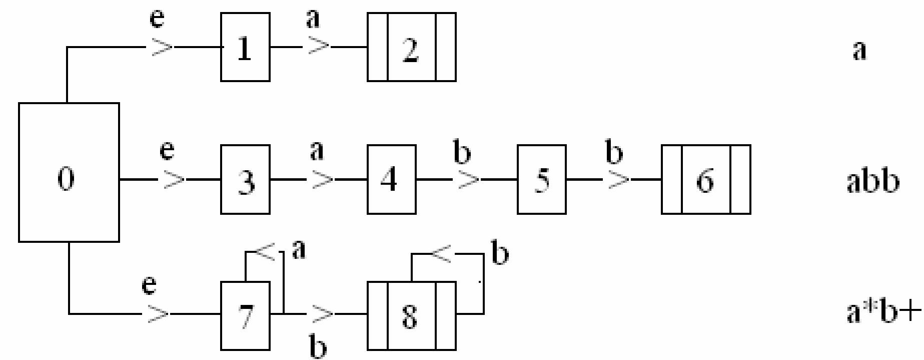
```
<in> "*/"          { BEGIN out; }
<in> .|\n          { }
%%
yywrap()  { return(1); }
main()  { yylex(); }
```

Для вызова программы Lex в ОС UNIX следует ввести команду «lex имя_исходного_файла». Выходной файл имеет имя «lex.yy.c».

5.1 Как устроен LEX

Мы попытаемся обсудить здесь то, как мог бы быть реализован генератор сканеров, из этого обсуждения не следует, что UNIX' овская программа LEX устроена именно так.

Для каждого из регулярных выражений r1, ... rn несложно построить недетерминированный конечный автомат, затем следует объединить полученные автоматы, создав новое начальное состояние с е-переходами в начальные состояния каждого регулярного выражения. (Полученный автомат будет распознавать выражение r1| r2|...rn.) Например, для выражений a, abb, a*b+ будет построен следующий автомат:



Для выделения лексем из входной цепочки можно применить немного модифицированный алгоритм моделирования недетерминированного КА. Отличие состоит в том, что наличие в текущем множестве состояний конечного состояния не является основанием для остановки автомата — не исключено, что цепочка может быть продолжена до более длинной лексемы. В этом

случае следует запомнить лексему и соответствующее ей регулярное выражение (более точно, то из подходящих выражений, которое было записано выше по тексту в исходной LEX-программе) и продолжить интерпретацию КА.

Этот процесс завершается, если из текущего множества состояний КА нет перехода для очередного символа из входной цепочки. Только теперь можно считать распознанной последнюю запомненную лексему, выполнить соответствующее ей действие, возвратить «лишние» просмотренные символы в входную цепочку, установить автомат в начальное состояние и продолжить поиск следующей лексемы. Построенный выше автомат при интерпретации цепочки aaba... пройдет следующие множества состояний:

0, 1, 3, 7 —a→ 2, 4, 7 —a→ 7 —b→ 8 —a→ stop

a (a) aab(a*b+)

В результате будет распознана лексема aab, как отвечающая регулярному выражению a*b+.

Детерминированный КА, полученный из построенного выше недетерминированного КА, также может быть использован для выделения лексем. Напомним, что состояние ДКА соответствует множеству состояний НКА. Если это множество содержит конечные состояния НКА, то следует пометить соответствующее ему (конечное) состояние ДКА первым регулярным выражением. Например, для рассмотренного выше примера может быть построен следующий ДКА:

Состояние	a	b	Рег. выражение
0,1,3,7	2,4,7	8	–
2,4,7	7	5,8	a
8	–	8	a*b+
7	7	8	–
5,8	–	6,8	a*b+
6,8	–	8	abb

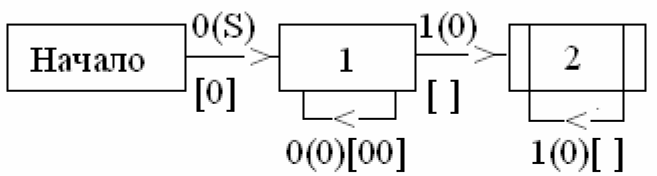
6 КОНТЕКСТНО-СВОБОДНЫЕ ЯЗЫКИ

Класс контекстно-свободных языков допускает распознавание с помощью недетерминированного КА со стековой (или магазинной) памятью.

Контекстно-зависимые языки — последний класс языков, которые можно эффективно распознать с помощью ЭВМ. Специалисты скажут, что они допускаются двусторонними недетерминированными линейно ограниченными автоматами. Для допуска цепочек языка без ограничений в общем случае требуется универсальный вычислитель (машина Тьюринга, машина с неограниченным числом регистров и т.п.). Контекстно-зависимые языки и языки без ограничений не используются при построении компиляторов и в дальнейшем рассматриваться не будут.

Автомат со стековой памятью в отличие от обычного КА имеет стек, в который можно помещать специальные т.н. «магазинные» символы. Переход из одного состояния в другое зависит не только от входного символа, но и от верхних символов магазина (на рисунке в круглых скобках). В начале работы в магазине лежит специальный символ S.

При выполнении перехода из магазина удаляются верхние символы, соответствующие правилу, и добавляется цепочка символов, соответствующих переходу (на рисунке изображены в квадратных скобках, первый символ цепочки становится верхним символом магазина). Допускаются также переходы, при которых входной символ игнорируется (и, тем самым, будет входным символом при следующем переходе). Эти переходы называются e-переходами. Автомат называется недетерминированным, если при одних и тех же состоянии, входном символе и вершине магазина возможен более чем один переход. Если при окончании цепочки автомат находится в одном из конечных состояний, а стек пуст, цепочка считается допущенной (после окончания цепочки могут быть сделаны e-переходы).



По контекстно-свободной грамматике легко строится недетерминированный КА с магазинной памятью, который допускает цепочки этого языка. Он использует только одно состояние и следующий набор переходов:

e(A)[x] для каждого правила грамматики A:x

a(a)[] для каждого терминала a

Можно построить и другой автомат, который также содержит одно состояние и имеет следующие переходы:

a(e)[a] для каждого терминала a

e(x)[A] для каждого правила грамматики A:x

Удобно считать, что в начале разбора магазин этого автомата пуст. Тогда в конце разбора в магазине останется символ S.

По недетерминированному КА с магазинной памятью можно построить КС-грамматику. Таким образом, класс КС-языков и класс языков, допускаемых автоматами с магазинной памятью, эквивалентны.

К сожалению, не каждый КС-язык допускает разбор с помощью детерминированного автомата. Например, язык цепочек-палиндромов из 0 и 1 не может быть допущен детерминированным КА с магазинной памятью. Таким образом, недетерминированные автоматы со стеком могут распознавать более широкий класс языков, чем детерминированные автоматы со стеком, — в этом их существенное отличие от КА. Практический интерес для реализации компиляторов представляют детерминированные КС-языки — собственное подмножество КС-языков, допускающее распознавание с помощью детерминированных автоматов с магазинной памятью.

Два (недетерминированных) автомата, построенных выше для КС-грамматики, и определяют два класса методов разбора КС-языков: сверху-вниз снизу вверх.

В первом случае при просмотре цепочки слева направо естественно будут получаться левые выводы. При детерминированном разборе проблема будет состоять в том, какое правило применить для раскрытия самого левого нетерминала.

Во втором случае детерминированный разбор удобно формализовать в терминах «сдвиг» (перенос символа из входной цепочки в магазин) и «свертка» (применение к вершине магазина какого-либо правила грамматики). При просмотре входной цепочки слева направо при этом будут получаться правые выводы. Проблема в этом случае состоит в выборе между сдвигом и сверткой и в выборе правила для свертки.

6.1 Лемма о разрастании для КС-языков

Для каждого КС-языка существует такое число m, что любую принадлежащую языку цепочку z : |z|>m можно разбить на 5 частей

z = uvwxy : |vx| > 0, |vwx|<=m и цепочки вида $uv^n wx^n y$, n>=0 также принадлежат языку.

Доказательство: Пусть КС-грамматика языка содержит q нетерминалов, а самая длинная цепочка в правой части правил имеет длину k. Рассмотрим дерево вывода для цепочки длины k**(2*q+3) и оставим в нем только те ветвления, которые ведут к терминальным листьям (игнорируем e-правила).

6.2 Преобразование КС-грамматик

Проблема эквивалентности двух языков, описываемых различными КС-грамматиками, в общем случае неразрешима. Однако часто удается преобразовать грамматику к требуемому для того или иного детерминированного разбора виду, не «испортив» описываемый ею язык.

Удаление бесполезных символов и правил

Назовем символ A бесполезным, если он не встречается ни в одном выводе цепочки терминалов из начального символа грамматики. Очевидно, что бесполезные символы и все содержащие их правила могут быть удалены из грамматики. Например, в грамматике

S : a | A ; A : Ab ; B : b

из нетерминала A нельзя вывести ни одной терминальной цепочки, а нетерминал B не может быть выведен из начального символа S.

Множество нетерминалов, из которых выводятся терминальные цепочки, легко вычислить:

```
K := {A | существует правило A:терминалы }; K1 := пусто;
цикл пока K != K1 выполнять
 K1 := K
 K := K U {A | существует правило
               A:(нетерминалы из K1 и и терминалы) }
конец цикла
```

Следствие: проблема пустоты КС-языка разрешима — язык пуст, если и только если S не принадлежит K.

После удаления из грамматики нетерминалов, не принадлежащих K, несложно найти множество L небесполезных символов:

```
L := {S}; L1 := пусто;
цикл пока L != L1 выполнять
 L1 := L
 L := L U {B | сущ. правило A:..B.., и A принадлежит L1 }
конец цикла
```

После удаления дополнения L из грамматики, она не будет содержать бесполезных символов. Будут ли удалены все бесполезные символы из грамматики, если применить сначала второй, а затем первый алгоритм?

Устранение e-правил

Правило вида A:e будем называть e-правилом. Если язык не содержит пустой цепочки, то из грамматики можно удалить все e-правила, в противном случае можно ввести новый начальный символ S1, правило S1 : S | e и удалить все остальные e-правила.

Устранение циклов и цепных правил

Обозначим символом :: выводимость одной цепочки из другой с помощью непустой последовательности правил. Назовем циклом вывод вида A::A... . Для устранения циклов можно удалить из грамматики все цепные правила вида A:B. Для каждого нецепного правила вида A:... добавим в грамматику правила B:..., для всех B, из которых с помощью цепных правил выводится A. После этого удаление цепных правил не изменит языка. Грамматику без циклов, e-правил и лишних символов называют приведенной.

Устранение левой рекурсии

Нетерминал A называется рекурсивным, если существует вывод: если x=e, то A называется леворекурсивным, если y=e, то праворекурсивным.

A :: xAy

Метод рекурсивного спуска не работал для леворекурсивных грамматик. От левой рекурсии всегда можно избавиться. (Это не означает, что метод рекурсивного спуска применим к любому языку.) Покажем вначале, как устранить прямую левую рекурсию, т.е. правила вида A:A... . Пусть

A : Aa1|...|Aan | b1|...|bn

все A-правила грамматики и ни одна из цепочек bi не начинается с A. Тогда, применяя A-правила к самому левому нетерминалу, из A можно вывести следующее регулярное множество цепочек:

(b1|...|bn)(a1|...|an)*

Это множество выводится и из преобразованной грамматики:

A : b1|...|bn | b1T|...|bnT

T : a1|...|an | a1T|...|anT

Устранить общую левую рекурсию можно с помощью следующего процесса, напоминающего последовательное исключение переменных при решении системы уравнений методом Гаусса.

```
цикл для i от 1 до n
 инвариант: для всех k<i (сущ. правило Ak:Al...) => l>k
 цикл для j от 1 до i-1
      инвариант: для всех k<j (сущ. правило Ak:Al...) =>
 l>k
      каждые правила Ai:Aj... заменить на правила Ai:Lj...,
      где Lj - все правые части Aj-правил
 конец цикла
 устранить прямую левую рекурсию для Ai
конец цикла
```

6.3 Алгоритм Кока—Янгера—Касами

Даны: КС-грамматика и цепочка символов. Требуется определить, выводима ли данная цепочка из грамматики. Напомним, что для регулярных языков аналогичная проблема решалась за линейное от длины цепочки время. Рассмотренный ниже алгоритм решает проблему принадлежности цепочки КС-языку за полиномиальное от длины цепочки время.

Вначале преобразуем грамматику в нормальную форму Хомского, т.е. в грамматику с правилами вида A:BC и A:a. Пусть a1a2...an — исходная цепочка символов. Построим треугольную таблицу Ti,j:

1<=i<=n, 1<=j<=n-i+1. Ti,j — множество нетерминалов, из которых выводится цепочка терминалов aj...aj+i-1 (подцепочка длины i, левый символ которой имеет индекс j).

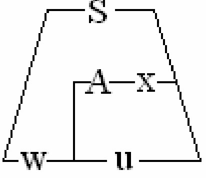
```
| T1,j нетерминалы, из которых выводятся односимвольные це-
почки
 цикл для j от 1 до n
  T1,j := {A | существует правило A:aj }
 конец цикла
| первым шагом вывода из A i-символьной цепочки (i>1) может
| быть только применение правила A:BC, где из B выводится
левая
```

```
    | подцепочка, а из C - правая, причем обе подцепочки имеют
длину,
    | меньшую i. Так называемое «динамическое программирова-
ние»:
    цикл для i от 2 до n
        цикл для j от 1 до n-i+1
            Tij := пусто
            цикл для k от 1 до i-1
                Tij := Tij U A | (существует правило A:BC)
                                и (B принадлежит Tk,j)
                                и (C принадлежит Ti-k,j+k)
            конец цикла
        конец цикла
    конец цикла
    цепочка выводима из грамматики := S принадлежит Tn,1
```

Алгоритм требует времени порядка n**3 и памяти n**2, где n — длина цепочки. Это делает его малопригодным для практических целей. На практике применяются специальные подклассы КС-грамматик, для которых существуют линейные алгоритмы разбора.

6.4 LL(k) языки и грамматики

Рассмотрим дерево вывода в процессе получения левого вывода цепочки. Промежуточная цепочка в процессе вывода состоит из цепочки из терминалов w, самого левого нетерминала A, недовыведенной части x:



Для продолжения разбора требуется заменить нетерминал A по одному из правил вида A:y. Если требуется, чтобы разбор был детерминированным (без возвратов), это правило требуется вы-

бирать специальным способом. Говорят, что грамматика имеет свойство LL(k), если для выбора правила оказывается достаточно рассмотреть только wAx и первые k символов непросмотренной цепочки u. Первая буква L (Left, левый) относится к просмотру входной цепочки слева направо, вторая — к используемому левому выводу.

Определим два множества цепочек:

FIRST(x) — множество терминальных цепочек, выводимых из x, укороченных до k символов.

FOLLOW(A) — множество укороченных до k символов терминальных цепочек, которые могут следовать непосредственно за A в выводимых цепочках.

Грамматика имеет свойство LL(k), если из существования двух цепочек левых выводов:

S :: wAx : wzx :: wu

S :: wAx : wtx :: wv

из условия FIRST(u)=FIRST(v) следует z=t.

В случае k=1 для выбора правила для A достаточно знать только нетерминал A и a — первый символ цепочки u:

следует выбрать правило A:x, если a входит в FIRST(x)

следует выбрать правило A:e, если a входит в FOLLOW(A)

LL(к)-свойство накладывает довольно сильные ограничения на грамматику. Например, LL(2) грамматика

S : aS | a

не обладает свойством LL(1), т.к. FIRST(aS)=FIRST(a)=a. В данном случае можно понизить величину k с помощью «факторизации» (вынесения множителя за скобку):

S : aA

A : S | e

Любая LL(k)-грамматика однозначна. Леворекурсивная грамматика не принадлежит классу LL(k) ни для какого k. Иногда

удается преобразовать не LL(1)-грамматику в эквивалентную ей LL(1)-грамматику с помощью устранения левой рекурсии и факторизации. Однако проблема существования эквивалентной LL(k)-грамматики для произвольной не LL(k)-грамматики неразрешима.

Существуют детерминированные языки, которые не являются LL(k) ни для какого k, например — $\{0^n\ 10^n\ U\ 0^n\ 10^{2n}\ |\ n>=1\}$.

Пример

Грамматика для арифметических формул:

Ф : T | Ф + T

T : M | T * M

M : (Ф) | a

не является LL(1), т.к. правила для Ф и T содержат прямую левую рекурсию, устраним ее:

	FIRST	FOLLOW
Ф : T Ф1	(a	) e
Ф1 : e | + T Ф1	+ e	) e
T : M T1	(a	+) e
T1 : e | * M T1	* e	+) e
M : (Ф) | a	(a	* +) e

Пустые цепочки порождают только нетерминалы Ф1 и T1. Более одного правила имеется для нетерминалов Ф1 и T1; множества FIRST(Ф1), FOLLOW(Ф1) и FIRST(T1), FOLLOW(T1) не пересекаются. Таким образом, преобразованная грамматика является LL(1).

Символы e, помещенные в множества FIRST и FOLLOW, имеют разный смысл и не приводят к конфликтам: e в множестве FIRST используется для обозначения возможности вывода пус-

той цепочки из данного нетерминала, e в множестве FOLLOW означает отсутствие следующего символа в конце строки терминалов. На практике роль e во втором случае может играть терминатор строки.

6.5 Простейший LL(1)-компилятор формул

Поскольку действия при LL(1)-разборе зависят только от пары «очередной нетерминал-очередной символ», этот разбор легко запрограммировать. Рекурсивный спуск — это один из способов кодировки LL(1)-разбора. Другой способ кодировки таблиц используется в следующем компиляторе.

Компилятор преобразует цепочки языка, соответствующего грамматике:

S: T {+T}

T: E {*E}

E: <операнд>|(S)

в постфиксную запись.

```
% {                           /* Коды лексем и метасимволов */
#define ID          1          /* Лексема — идентификатор */
#define META        100        /* Метасимволы грамматики */
#define S           (META+0)
#define T           (META+1)
#define E           (META+2)
                               /* Коды переходов */
#define NOGO        100        /* Коды завершения разбора */
#define OK          (NOGO+1)
#define ERR         110        /* Коды ошибок */
%}

%%
"+"|"*"|"("|")"     { return(*yytext);  }
[A-Z0-9]+           { printf("%s ",yytext); return(ID);  }
```

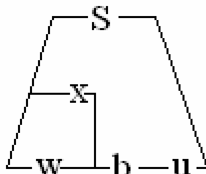
```
\n              { }
.               { printf("%s: ",yytext); error(0);  }
%%

static lexcode, oldcode;      /* Коды очередной и предыдущей
                              лексем */
main()  {
     lexcode = yylex();
     parse( S ); printf("\n");
     if ( lexcode ) error(4);      /* Завершился ли разбор по
                                   концу файла? */
}

/*
 * LL(1)-таблицы состоят из четырех полей:
 *   - номер лексемы или метасимвола для сопоставления
 *   - номер действия при успехе (семантическая программа)
 *   - переход при успехе (номер строки/успех/неуспех/ошибка)
 *   - переход при неуспехе
 */
static char stable [ ] = {         /* LL(1) — таблица для S */
     /* 00 */ T,    0,             1,   ERR+1,
     /* 01 */ '+',  1,             2,   OK,
     /* 02 */ T,    2,             1,   ERR+1,
};
static char ttable [ ] ={          /* LL(1) — таблица для T */
/* 00 */ E,         0,             1,   ERR+2,
/* 01 */ '*',       1,             2,   OK,
/* 02 */ E,         2,             1,   ERR+2,
};
static char etable [ ] ={          /* LL(1) — таблица для E */
/* 00 */ ID,        0,             OK,  1,
/* 01 */ '(',       0,             2,   NOGO,
/* 02 */ S,         0,             3,   ERR+1,
```

```
/* 03 */ ')',       0,              OK,  ERR+3,
};
/* Указатели на LL(1) — таблицы */
static char *blocks[] = {stable, ttable, etable, };

/*
 * Интерпретатор LL(1)-таблиц
 * Вход:       m — лексема или метасимвол
 * Результат:  1, когда сопоставление удалось
 */
parse (m) {
     register char *block, *p, sign;
          if ( m<META ) {
               if (lexcode != m) return(0);
               oldcode = lexcode; lexcode = yylex();
               return(1);
     }
     p = block = blocks[m-META];
     for(;;) {
          if ( parse( *p++ ) ) {
               switch ( *p++ ) {        /* Семпы */
                    case 0: break;
                    case 1: sign = oldcode; break;
                    case 2: printf("%c ", sign); break;
                    default: error(1);
               }
          } else p += 2;
          if ( *p < NOGO ) p = block+*p*4;
          else if ( *p <= OK ) return (*p-NOGO);
          else error(*p-ERR);
     }
}

static char *ermsg[] = {/* Тексты сообщений об ошибках */
```

```
    /* 00 */ "ошибочный символ", /* 01 */ "отказ",
    /* 02 */ "не найден операнд", /* 03 */ "не хватает )",
    /* 04 */ "не найден знак",
};

error(e)  {printf ("%s\n", ermsg[e]); exit(1); }
yywrap()  {return(1); }
```

6.6 Разбор снизу-вверх. Сдвиг-свертка. Простое и операторное предшествование

Рассмотрим разбор снизу-вверх, при котором промежуточные выводы перемещаются по дереву по направлению к корню. Если считывать символы цепочки слева направо, то дерево разбора будет выглядеть следующим образом:



Промежуточный вывод имеет вид xbu, где x — цепочка терминалов и нетерминалов, из которой выводится просмотренная часть терминальной цепочки w, bu — непросмотренная часть терминальной цепочки, b — очередной символ. Чтобы продолжить разбор, можно либо добавить символ b к просмотренной части цепочки (выполнить так называемый «сдвиг»), либо выделить в конце x такую цепочку z (x=yz), что к z можно применить одно из правил грамматики B:z и заменить x на цепочку yB (выполнить так называемую «свертку»):

После сдвига После свертки

Если свертку применять только к последним символам x, то мы будем получать правые выводы цепочки. Такой разбор получил название LR, где символ L (Left, левый) относится к просмотру цепочки слева направо, а R (Right, правый) относится к получаемым выводам.

Пример LR-разбора цепочки aabb в соответствии с грамматикой S : SaSb | e. Символ _ указывает на границу между просмотренной и непросмотренной частями цепочки: _aabb : S_aabb : Sa_abb : SaS_abb : SaSa_bb : SaSaS_bb : SaSaSb_b : SaS_b : SaSb_ : S_

Последовательность операций сдвига и свертки существенна — если бы в приведенном примере первой операцией был бы сдвиг, разбор не удалось бы довести до конца. Поэтому для детерминированного разбора требуется в каждый момент выбирать между сдвигом и сверткой (и различными правилами свертки). В курсе будут рассмотрены три способа выбора: простое и операторное предшествования и LR(k)-разбор.

6.7 Грамматики простого предшествования

Грамматика называется обратимой, если в ней нет двух правил с одинаковыми правыми частями. Напомним, что грамматика называется приведенной, если в ней нет e-правил, бесполезных символов и циклов.

Зададимся целью ввести на множестве терминальных и нетерминальных символов три отношения (так называемые «отношения предшествования») (будем обозначать их <', =' и >') так, чтобы в момент, когда требуется свертка цепочки z, отношения между символами построенной части вывода и очередным символом b были следующими:

<' или =' <' =' =' >'

y z b u

Если удастся построить такие отношения, то LR-разбор для обратимой грамматики можно проводить очень просто:

– сделать сдвиг, если последний символ x <' b или последний символ x =' b;

– сделать свертку, если последний символ x >' b, при этом правая часть правила z заключена между отношениями <' и >'.

Грамматика называется грамматикой простого предшествования, если она приведенная, обратимая и между любыми двумя терминалами или нетерминалами выполняется не более одного отношения предшествования.

Практически отношения предшествования можно вычислить следующим образом (X,Y — терминалы или нетерминалы, A,B,C-нетерминалы, y — терминал, ... — любая цепочка (м.б. пустая)):

X =' Y, если в грамматике есть правило A : ...XY...

X <' Y, если в грамматике есть правило A : ...XB...

и B :: Y...

X >' y, если в грамматике есть правило A : ...By...

или A : ...BC...

и B :: ...X

и C :: y...

Вычисляя отношения предшествования для грамматики S:aSSb|c, получим: a='S, S='S, S='b, {a,S} <' a,c, {b,c} >' {a,b,c}.

Эта грамматика является грамматикой простого предшествования.

На практике иногда удобно считать, что в начале и конце цепочки языка стоят символы-ограничители #. Для них отношения предшествования определяются так:

\# <' X, если S :: X...

X >' #, если S :: ...X

В нашем примере {b,c} >' #, # <' {a,c}.

Отметим, что отношения <',>',=' не похожи на обычные арифметические отношения <, >, = : =' не является отношением эквивалентности, <' и >' не обязательно транзитивны.

Отношения предшествования удобно занести в матрицу, в которой строки и столбцы помечены терминалами и нетерминалами грамматики.

6.8 Грамматики операторного предшествования

Если в правилах приведенной обратимой грамматики не встречаются рядом два нетерминала, говорят, что грамматика является операторной. Классический пример — грамматика арифметических формул. Для таких грамматик можно вычислять отношения предшествования только на множестве терминальных символов. Для этого модифицируем правила вычисления отношений предшествования:

a =' b, если в грамматике имеется правило A : ...ab...
или A : ...aBb...

a <' b, если в грамматике имеется правило A : ...aB...
и B :: b...
или B :: Cb...

a >' b, если в грамматике имеется правило A : ...Bb...
и B :: ...a
или B :: ...aC

<' a, если в грамматике имеется вывод S :: Ca...
или S :: a...

a >' #, если в грамматике имеется вывод S :: ...aC
или S :: ...a

Если между любыми двумя терминалами выполняется не более одного отношения предшествования, операторная грамматика называется грамматикой операторного предшествования.

Вычислим матрицу операторного предшествования для грамматики арифметических формул:

S : S + T | T
T : T * E | E
E : (S) | a

	(	a	*	+	)	#
)			>	>	>	>
a			>	>	>	>
*	>	>	>	>	>	>
+	>	>	>	>	>	>
(	>	>	>	>	=	
#	>	>	>	>		

Эти отношения позволяют определять терминалы, входящие в правую часть сворачиваемого правила, но не нетерминалы. Однако при практическом разборе формул нет необходимости различать нетерминалы S, T и E. Они были введены только для придания грамматике однозначности и учета приоритета и ассоциативности операций. Теперь, получив отношения предшествования, можно вновь заменить эти искуственно введенные нетерминалы на S:

S : S+S | S*S | (S) | a

и получить разбор, обрабатывая все нетерминалы одинаково.

6.9 Линеаризация матрицы предшествования

Для компактного хранения матрицы предшествования часто можно использовать следующий прием. По матрице M[n][n], элементы которой принимают только три значения (<' =' >'), попытаемся построить два целочисленных вектора f и g:

M[i][j] равно >', если f[i]>g[j]

M[i][j] равно <', если f[i]<g[j]

M[i][j] равно =', если f[i]=g[j]

Для получения этих векторов используется следующий метод:

– построить ориентированный граф, содержащий n вершин типа F и n вершин типа G;

– построить ребро графа F[i]->G[j], если i >' j

– построить ребро графа F[i]<-G[j], если i <' j

– склеить вершины F[i] и G[j], если i =' j

Если полученный граф циклический, то линеаризация невозможна. Иначе положить f[i] равным длине самого длинного пути из F[i], g[i] равным длине самого длинного пути из G[i].

Применив этот метод для построенной матрицы операторного предшествования, получим:

символ	#	a	+	*	(	)
f	0	4	2	4	0	4
g	0	5	1	3	5	0

6.10 Еще один компилятор формул (операторное предшествование)

Для реализации компилятора формул в польскую постфиксную запись свяжем с каждой сверткой действие:

a : Число или имя — напечатать число или имя

S : a | (S) — ничего не делать

S : S*S | S+S — напечатать знак операции

Построенный частичный вывод x будем хранить в стеке:

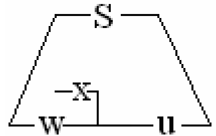
```
% {                  /* Коды лексем и метасимволов */
#define EF           0    /* конец файла */
#define ID     1     /* идентификатор или число */
#define PLUS   2     /* '+' */
#define MULT   3     /* '*' */
#define LP     4     /* '(' */
#define RP     5     /* ')' */
%}

%%
"+"            { return(PLUS);  }
"*"            { return(MULT);  }
"("            { return(LP);  }
")"            { return(RP);  }
[A-Z0-9]+      { printf("%s ",yytext); return(ID);  }
\n             { }
.              { printf("%s: ",yytext); error(0);  }
%%

static char *view[] = {/*Постфиксное представление терминалов
*/
```

```
    "",       /* конец файла */   "",       /*    идентификатор
*/
    "+ ",     /* '+' */           "* ",     /* '*' */
    "",       /* '(' */           "",       /* ')' */
};

/* Линеаризованная матрица операторного предшествования */
static f[] = { 0, 4, 2, 4, 0, 4, } ;
static g[] ={ 0, 5, 1, 3, 5, 0, };

main() {
     register c, d;
          stinit(); stpush(EF);
          do {
               c = yylex();
               while ( f[sttop()] > g[c] ) {
                    do {
                         d = stpop();
                         printf ("%s",view[d]);
                    } while ( f[sttop()]==g[d] );
               }
          stpush(c);
          } while(c!=EF);
}

/* Реализация стека терминалов */
#define MAXSTK 100
static int stack[MAXSTK], *stptr;

stinit() { stptr = stack+MAXSTK; }
stpush(e){
     if ( stptr==stack ) error(1);
     *--stptr = e;
}
```

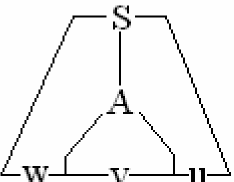
```
sttop() { stcheck(); return(*stptr);   }
stpop() { stcheck(); return(*stptr++); }
stcheck() { if (stptr==stack+MAXSTK) error(1); }

static char *ermsg[] = { /* Тексты сообщений об ошибках */
     /* 00 */ "ошибочный символ", /* 01 */ "отказ",
};

error(e) { printf ("%s\n", ermsg[e]); exit(1); }
yywrap() { return(1); }
```

7 LR(K)-ГРАММАТИКИ. SLR(1), LALR(1)-ГРАММАТИКИ

Если в процессе LR-разбора принять детерминированное решение о сдвиге/свертке удается, рассматривая только цепочку x и первые k символов непросмотренной части входной цепочки u (эти k символов называют аванцепочкой), говорят, что грамматика обладает LR(k)-свойством.



Различие между LL(k)- и LR(k)-грамматиками в терминах дерева вывода:



В случае LL(k)-грамматик однозначно определить правило, примененное к A, можно по w и первым k символам vu, а в случае LR(k)-грамматик — по w,v и первым k символам u. Это нестрогое рассуждение показывает, что LL(k)-языки < LR(k)-языки (опровергните его для k=0).

7.1 LR(0)-грамматики

Рассмотрим вначале наиболее простые грамматики этого класса — LR(0). При разборе строки LR(0)-языка можно вообще не использовать аванцепочку — выбор между сдвигом и сверткой делается на основании цепочки x (см. картинку). Так как в процессе разбора она изменяется только с правого конца, ее называют стеком. Будем считать, что в грамматике нет бесполезных символов и начальный символ не встречается в правых частях правил — тогда свертка к начальному символу сигнализирует об

успешном завершении разбора. Попробуем описать множество цепочек из терминалов и нетерминалов, появляющихся в стеке в процессе всех LR-разборов (другими словами — всех правых выводов из грамматики).

Определим следующие множества:

L(A:v) — левый контекст правила A:v — множество состояний стека, непосредственно перед сверткой v в A в ходе всех успешных LR-разборов. Очевидно, каждая цепочка из L(A:v) кончается на v. Если у всех таких цепочек отрезать хвост v, то получится множество цепочек, встречающихся слева от A в процессе всех успешных правых выводов. Обозначим это множество.

L(A) — левый контекст нетерминала A. Построим грамматику для множества L(A). Терминалами новой грамматики будут терминалы и нетерминалы исходной грамматики, нетерминалы новой грамматики обозначим <L(A)>,... — их значениями будут левые контексты нетерминалов исходной грамматики. Если S — начальный символ исходной грамматики, то грамматика левых контекстов будет содержать правило

<L(S)> : e — левый контекст S содержит пустую цепочку

Для каждого правила исходной грамматики, например

A : B C d E

добавим в новую грамматику правила:

<L(B)> : <L(A)> — L(B) включает L(A)

<L(C)> : <L(A)> B — L(C) включает L(A) B

<L(E)> : <L(A)> B C d — L(E) включает L(A) B C d

Полученная грамматика имеет специальный вид (такие грамматики называются леволинейными), следовательно, множества левых контекстов — регулярны. Из этого следует, что принадлежность цепочки левому контексту какого-либо нетерминала можно вычислять индуктивно с помощью конечного автомата, просматривая цепочку слева направо. Опишем этот процесс конструктивно.

Назовем LR(0)-ситуацией правило грамматики с одной отмеченной позицией между символами правой части правила. Например, для грамматики S:A; A:aAA; A:b существуют следующие LR(0)-ситуации: S:_A; S:A_; A:_aAA; A:a_AA; A:aA_A; A:aAA_; A:_b; A:b_. (позиция обозначена символом подчеркивания).

Будем говорить, что цепочка x согласована с ситуацией A:b_c, если x=ab и a принадлежит L(A). (Другими словами, LR-вывод может быть продолжен x_... = ab_... :: abc_... :: aA_... :: S_ .) В этих терминах L(A:b) — множество цепочек, согласованных с ситуацией A:b_, L(A) — цепочки, согласованные с ситуацией A:_b, для любого правила A:b.

Пусть V(u) — множество ситуаций, согласованных с u. Покажем, что функция V — индуктивна.

Если в множество V(u) входит ситуация A:b_cd, то ситуация A:bc_d принадлежит V(uc) (c — терминал или нетерминал; b, d — последовательности (может быть пустые) терминалов и нетерминалов). Других ситуаций вида A:b_d с непустым b в V(uc) нет. Осталось добавить в V(uc) ситуации вида C:_... для каждого нетерминала C, левый контекст которого содержит uc. Если ситуация A:..._C... (C-нетерминал) принадлежит множеству V(uc), то uc принадлежит L(C) и V(uc) включает в себя ситуации вида C:_... для всех C-правил грамматики.

Пример: построим функцию V для грамматики S:A; A:aAA; A:b.

0 V(e) = { S:_A; A:_aAA, A:_b }

1 V(A) = { S:A_ }

2 V(a) = { A:a_AA; A:_aAA, A:_b } V(aa)=V(a); V(ab)=V(b);

3 V(b) = { A:b_ }

4 V(aA) = { A:aA_A; A:_aAA, A:_b } V(aAa)=V(a);V(aAb)=V(b);

5 V(aAA) = { A:aAA_ }

Наконец, мы готовы дать определение LR(0)-грамматики. Пусть u — содержимое стека в процессе LR-разбора, V(u)-множе-

ство LR(0) ситуаций, согласованных с u. Если V(u) содержит ситуацию вида A:x_ (x-последовательность терминалов и нетерминалов), то u принадлежит L(A:x) и допустима свертка x в A. Если V(u) содержит ситуацию A:..._a... (a-терминал), то допустим сдвиг. Говорят о конфликте сдвиг-свертка, если для одной цепочки u допустимы и сдвиг, и свертка. Говорят о конфликте свертка-свертка, если допустимы свертки по различным правилам.

Грамматика называется LR(0), если для всех состояний стека в процессе LR-вывода нет конфликтов сдвиг-свертка или свертка-свертка.

Рассмотренная выше грамматика является LR(0)-грамматикой. Ее функция V принимает 6 различных значений (вычисляется конечным автоматом с 6 состояниями). В состояниях 0,2,4 возможен только сдвиг, в состоянии 3 — свертка по правилу A:b, в состоянии 5 — свертка A:aAA, в состоянии 1 — свертка S:A — т.е. успешное завершение разбора.

Остается показать, как можно построить парсер, разбирающий предложения LR(0)-языка. Чтобы не вычислять значение функции V заново для каждого нового состояния стека, будем хранить в стеке вместе с каждым символом xi значение V на цепочке (x0...xi).

```
стек.сделать пустым
стек.добавить ( '#', начальное состояние )
цикл
.  выбор из V ( стек.вершина.состояние ).действие
. .  "сдвиг" =>
. .  прочитать очередной символ в ( новый символ )
. .  "свертка" =>
. .  удалить правую часть правила из стека
. .  новый символ := левая часть правила
. .  "успех" =>
. .  стоп( "успех" )
. конец выбора
```

. старое состояние := V (стек.вершина.состояние)

. новое состояние := старое состояние . переход [новый символ]

. если новое состояние = "Ошибка" то стоп("ошибка") конец если

. стек.добавить (новый символ, новое состояние)

конец цикла

Таблицы LR(0)-парсера для грамматики 1) S:A; 2) A:aAA; 3) A:b

	Префикс	Действие	Переход		
			A	a	b
0	e	сдвиг	1	2	3
1	A	успех	Ошибка	Ошибка	Ошибка
2	a	сдвиг	4	2	3
3	b	свертка 3,A	Ошибка	Ошибка	Ошибка
4	aA	сдвиг	5	2	3
5	aAA	свертка 2,A	Ошибка	Ошибка	Ошибка

7.2 LR(k)-грамматики

Для выбора между сдвигом или сверткой в LR(0)-разборе используется только состояние стека. В LR(k)-разборе учитывается также k-первых символов непросмотренной части входной цепочки (так называемая аванцепочка). Для обоснования метода следует аккуратно повторить рассуждения предыдущего параграфа, внеся изменения в определения.

Будем включать в левый контекст правила также аванцепочку. Если в правом выводе применяется вывод wAu : wvu, то пара {wv,FIRSTk(u)} принадлежит Lk(A:v), а пара {w,FIRSTk(u)} — Lk(A). Множество левых контекстов, как и в случае LR(0), можно

вычислять с помощью индукции по левой цепочке. Назовем LR(k)-ситуацией пару: правило грамматики с отмеченной позицией и аванцепочку длины не более k. Отделять правило от аванцепочки будем вертикальной чертой.

Будем говорить, что цепочка x согласована с ситуацией A:b_c|t если существует LR-вывод: x_yz = ab_yz :: abc_z :: aA_z :: S_, и FIRSTk(z)=t. Правила индуктивного вычисления множества состояний Vk следующие:

Vk(e) содержит ситуации S:_a|e для всех правил S:a, где S — это начальный символ. Для каждой ситуации A:_Ba|u из Vk(e), каждого правила B:b и цепочки x, принадлежащей FIRSTk(au), надо добавить в Vk(e) ситуацию B:_b|x.

Если в Vк(w) входит ситуация A:b_cd|u, то ситуация A:bc_d|u будет принадлежать Vk(wc). Для каждой ситуации A:b_Cd|u из Vk(wc), каждого правила C:f и цепочки x, принадлежащей FIRSTk(du), надо добавить в Vk(wc) ситуацию C:_f|x.

Пример: построим функцию V1 для грамматики S:A; A:AaAb|e.

0 V1(e)   = { S:_A|e; A:_AaAb|e,a, A:_|e,a }

1 V1(A)   = { S:A_|e, A:A_aAb|e,a }

2 V1(Aa)   = { A:Aa_Ab|e,a; A:_AaAb|a,b, A:_|a,b }

3 V1(AaA)   = { A:AaA_b|e,a, A:A_aAb|a,b }

4 V1(AaAa)   = { A:Aa_Ab|a,b; A:_AaAb|a,b, A:_|a,b }

5 V1(AaAb)   = { A:AaAb_|e,a }

6 V1(AaAaA)   = { A:AaA_b|a,b A:A_aAb|a,b } V1(AaAaAa)=V1(AaAa)

7 V1(AaAaAb)   = { A:AaAb_|a,b }

(A:_AaAb|e,a — сокращенная запись двух LR(1)-ситуаций:

A:_AaAb|e и A:_AaAb|a)

Используем построенные множества LR(k)-состояний для разрешения вопроса сдвиг-свертка. Пусть u — содержимое стека,

а x — аванцепочка. Очевидно, что свертка по правилу A:b может быть проведена, если Vk(u) содержит ситуацию A:b_|x. Решение вопроса о допустимости сдвига требует аккуратности, если в грамматике имеются e-правила. В ситуации A:b_c|t (c не пусто) сдвиг возможен, если c начинается с терминала и x принадлежит FIRSTk(ct). Неформально говоря, можно занести в стек самый левый символ правой части правила, подготавливая последующую свертку. Если c начинается с нетерминала (ситуация имеет вид A:b_Cd|t), то занести в стек символ, подготавливая свертку в C, можно только в случае, если C не порождает пустую цепочку. Например, в состоянии V(e)={ S:_A|e; A:_AaAb|e,a, A:_|e,a } нет допустимых сдвигов, т.к. при выводе из A терминальных цепочек на некотором шаге требуется применить правило A:e к нетерминалу A, находящемуся на левом конце цепочки.

Определим множество EFFk(x), состоящее из всех элементов множества FIRSTk(x), при выводе которых нетерминал на левом конце x (если он есть) не заменяется на пустую цепочку. В этих терминах сдвиг допустим, если в множестве Vk(u) есть ситуация A:b_c|t, c не пусто и x принадлежит EFFk(ct).

Грамматика называется LR(k)-грамматикой, если ни одно LR(k)-состояние не содержит двух ситуаций A:b_|u и B:c_d|v, таких, что u принадлежит EFFk(dv). Такая пара соответствует конфликту свертка-свертка, если d пусто, и конфликту сдвиг-свертка, если d не пусто.

LR(k)-парсер устроен аналогично LR(0). Действие из множества сдвиг, свертка, успех, ошибка, выполняемое на очередном шаге LR-разбора, есть функция от состояния стека Vk(u) и аванцепочки x:

сдвиг, если A:b_c|t содержится в Vk(u), c!=e, x из EFF(ct);

свертка, если A:a_|x содержится в Vk(u);

успех, если S:A|e содержится в Vk(u);

ошибка в остальных случаях.

Таблицы LR(1)-парсера для грамматики S:A; A:AaAb|e

	Префикс	Действие			Переход		
		a	b	e	a	b	A
0	e	Св. 0,A	Ошибка	Св. 0,A	Ошибка	Ошибка	1
1	A	Сдвиг	Ошибка	Успех	2	Ошибка	Ошибка
2	Aa	Св. 0,A	Св. 0,A	Ошибка	Ошибка	Ошибка	3
3	AaA	Сдвиг	Сдвиг	Ошибка	4	5	Ошибка
4	AaAa	Св. 0,A	Св. 0,A	Ошибка	Ошибка	Ошибка	6
5	AaAb	Св. 4,A	Ошибка	Св. 4,A	Ошибка	Ошибка	Ошибка
6	AaAaA	Сдвиг	Сдвиг	Ошибка	4	7	Ошибка
7	AaAaAb	Св. 4,A	Св. 4,A	Ошибка	Ошибка	Ошибка	Ошибка

На практике LR(k)-грамматики при k>1 не применяются. На это имеются две причины. Первая: очень большое число LR(k)-состояний. Вторая: для любого языка, определяемого LR(k)-грамматикой, существует LR(1)-грамматика; более того, для любого детерминированного КС-языка существует LR(1)-грамматика.

Число LR(1)-состояний для практически интересных грамматик также весьма велико. LR(0)-свойством такие грамматики обладают редко. На практике чаще всего используются промежуточные между LR(0) и LR(1) методы, известные под названиями SLR(1) и LALR(1).

7.3 SLR(1)- и LALR(1)-грамматики

В основе этих двух методов лежит одна и та же идея. Построим множество канонических LR(0)-состояний грамматики. Если это множество не содержит конфликтов, то можно применить LR(0)-парсер. Иначе попытаемся разрешить возникшие конфликты, рассматривая односимвольную аванцепочку. Другими словами, попробуем построить LR(1)-парсер с множеством LR(0)-состояний.

В SLR(1)-грамматиках (Simple LR(1) — простых LR(1)-грамматиках) для разрешения конфликтов используется множество FOLLOW(X) — множество терминалов, встречающихся после X. Если в состоянии имеется ситуация A:b_, свертка допускается, если только аванцепочка принадлежит FOLLOW(A).

Грамматика является SLR(1)-грамматикой, если для двух любых LR(0)-ситуаций из одного состояния A:a_b и B:c_d выполняется одно из следующих условий:

- b!=e, d!=e (конфликта нет, требуется сдвиг);
- b=d=e и FOLLOW(A) не пересекается с FOLLOW(B) (конфликт "свертка/свертка" может быть устранен с учетом аванцепочки);
- b=e, d<>e и FOLLOW(A) не пересекается с EFF(tFOLLOW(B)) (конфликт "сдвиг/свертка" может быть устранен с учетом аванцепочки).

Построим LR(0)-состояния для грамматики арифметических формул: S:E; E:E+T|T; T:T*F|F; F:(E)|a.

0 V(e) = { S:_E; E:_E+T, E:_T, T:_T*F, T:_F, F_a, F:_(E) }

1 V(E) = { S:E_, E:E_+T }

2 V(T) = { E:T_, T:T_*F,

3 V(F) = { T:F_ }

4 V(a) = { F:a_ }

5 V(() = { F:(_E); E:_E+T, E:_T, T:_T*F, T:_F, F_a, F:_(E) }

6 V(E+) = { E:E+_T; T:_T*F, T:_F, F_a, F:_(E) }

7 V(T*) = { T:T*_F; F:_a, F:_(E) }

8 V((E) = { F:(E_), E:E_+T }   (T=T; (F=F; (a=a; ((=(

9 V(E+T) = { E:E+T_, T:T_*F }   E+F=F; E+a=a; E+(=(

10 V(T*F)= { T:T*F_ }   T*a=a; T*(=(

11 V((E))= { F:(E)_ }   (E+=E+; E+T*=T*

LR(0)-конфликты возникают в состояниях 1,2,9, но

1: FOLLOW(S) = {e}, FIRST(+T) = {+}

2: FOLLOW(E) = {+,),e} FIRST(*F) = {*}

9: FOLLOW(E) = {+,),e} FIRST(*F) = {*},

следовательно, конфликты разрешаются с использованием SLR(1)-техники и грамматика является SLR(1)-грамматикой.

Таблицы SLR(1)-парсера для грамматики арифметических формул

	Действие						Переход							
	e	+	*	a	(	)	+	*	a	(	)	E	T	F
0	Ош	Ош	Ош	Сдв	Сдв	Ош	Ош	Ош	4	5	Ош	1	2	3
1	Усп	Сдв	Ош	Ош	Ош	Ош	6	Ош	Ош	Ош	Ош	Ош	Ош	Ош
2	1,E	1,E	Сдв	Ош	Ош	1,E	Ош	7	Ош	Ош	Ош	Ош	Ош	Ош
3	1,T	1,T	1,T	Ош	Ош	1,T	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош
4	1,F	1,F	1,F	Ош	Ош	1,F	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош
5	Ош	Ош	Ош	Сдв	Сдв	Ош	Ош	Ош	4	5	Ош	8	2	3
6	Ош	Ош	Ош	Сдв	Сдв	Ош	Ош	Ош	4	5	Ош	Ош	9	3
7	Ош	Ош	Ош	Сдв	Сдв	Ош	Ош	Ош	4	5	Ош	Ош	Ош	10
8	Ош	Сдв	Ош	Ош	Ош	Сдв	6	Ош	Ош	Ош	11	Ош	Ош	Ош
9	3,E	3,E	Ош	Ош	Ош	3,E	Ош	7	Ош	Ош	Ош	Ош	Ош	Ош
10	3,T	3,T	3,T	Ош	Ош	3,T	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош
11	3,F	3,F	3,F	Ош	Ош	3,F	Ош	Ош	Ош	Ош	Ош	Ош	Ош	Ош

LALR(1)-метод (Look Ahead — заглядывание вперед) заключается в следующем. Введем на множестве LR(1)-ситуаций отношение эквивалентности: будем считать две ситуации эквивалентными, если они различаются только аванцепочками. Например, ситуации A:Aa_Ab|e и A:Aa_Ab|a эквивалентны. Построим каноническое множество LR(1)-состояний и объединим состояния, состоящие из множества эквивалентных ситуаций.

Например, LR(1)-состояния 2 и 4 грамматики S:A; A:AaAb|e эквивалентны:

2 V1(Aa) = { A:Aa_Ab|e,a; A:_AaAb|a,b, A:_|a,b }

4 V1(AaAa) = { A:Aa_Ab|a,b; A:_AaAb|a,b, A:_|a,b }

и могут быть объединены в одно состояние

2+4 V1(Aa) = { A:Aa_Ab|e,a,b; A:_AaAb|a,b, A:_|a,b }

Также эквивалентны состояния 3,6 и 5,7 этой грамматики.

Если полученное множество состояний не содержит LR(1)-конфликтов и, следовательно, позволяет построить LR(1)-парсер, то говорят, что грамматика обладает свойством LALR(1).

Например, грамматика S:A; A:AaAb|e является LALR(1).

Таблицы LALR(1)-парсера для грамматики S:A; A:AaAb|e

	Префикс	Действие			Переход		
		a	b	e	a	b	A
0	e	Св. 0,A	Ошибка	Св. 0,A	Ошибка	Ошибка	1
1	A	Сдвиг	Ошибка	Успех	2	Ошибка	Ошибка
2+4	Aa	Св. 0,A	Св. 0,A	Ошибка	Ошибка	Ошибка	3+6
3+6	AaA	Сдвиг	Сдвиг	Ошибка	4	5+7	Ошибка
5+7	AaAb	Св. 4,A	Св. 4,A	Св. 4,A	Ошибка	Ошибка	Ошибка

Заметим, что при слиянии канонических LR(1)-состояний, различающихся только аванцепочками, получается множество канонических LR(0)-состояний, для каждой ситуации которого вычислено множество допустимых аванцепочек. Следовательно, SLR(1)- и LALR(1)-методы при успешном применении дают одинаковые таблицы парсера. Метод LALR(1) применим к более широкому классу грамматик, чем SLR(1). Это объясняется тем, что отношение FOLLOW(A), применяемое при вычислении допустимых аванцепочек в SLR(1)-методе, не использует всю доступную информацию — оно не зависит от левого контекста правила A:... Действительно, существуют LALR(1)- грамматики, не принадлежащие классу SLR(1).

8 YACC

Программа YACC (Yet Another Compiler Compiler) предназначена для построения синтаксического анализатора контекстно-свободного языка. Анализируемый язык описывается с помощью грамматики в виде, близком форме Бэкуса—Наура (НФБН). Результатом работы YACC'а является программа на Си, реализующая восходящий LALR(1)-распознаватель.

8.1 Структура YACC-программы

YACC-программа состоит из трех секций, разделенных символом %%, — секции описаний, секции правил, в которой описывается грамматика, и секции программ, содержимое которой просто копируется в выходной файл. Пример простейшей программы на YACC'е:

```
%token name
%start e
%%
e : e '+' m | e '-' m | m ;
m : m '*' t | m '/' t | t ;
t : name | '(' e ')' ;
%%
```

Секция правил содержит информацию о том, что символ name является лексемой (терминалом) грамматики, а символ e — ее начальным нетерминалом.

Грамматика записана обычным образом — идентификаторы обозначают терминалы и нетерминалы, символьные константы типа '+' и '-'.

- терминалы. Символы : | ; принадлежат метаязыку и читаются «есть по определению», «или» и «конец правила» соответственно.

8.2 Разрешение конфликтов

Написанная грамматика (она обладает свойством LALR(1)) задает язык арифметических формул, в котором приоритет '*' и '/' выше приоритета '+' и '-', а все операции левоассоциативны. Для указания этих свойств языка в грамматику введены дополнительные нетерминалы m и t. Другая грамматика этого языка:

```
e : e '+' e | e '-' e | e '*' e | e '/' e | '(' e ')' | name ;
```

не однозначна (и, следовательно, не LALR(1)). Попытка применить YACC для анализа данной грамматики приведет к многочисленным (16) неразрешенным конфликтам типа «сдвиг/свертка» (shift/reduce) в построенном автомате. Если рассмотреть конфликты более подробно, выясняется, что в каждом случае можно однозначно выбрать между сдвигом или сверткой, основываясь на приоритетах операций и порядке выполнения равноприоритетных операций слева направо. (Аналогично простому и операторному предшествованию.)

YACC позволяет дополнить грамматику информацией такого рода и получить бесконфликтный распознаватель:

```
%token name
%left '+' '-'
%left '*' '/'
%%
e : e '+' e | e '-' e | e '*' e | e '/' e
| '(' e ')' | name ;
%%
```

Предложения %left, %right и %nonassoc в секции описаний приписывают всем перечисленным за ними символам одинаковый приоритет и соответствующее значение ассоциативности. (Отсутствие ассоциативности означает недопустимость выраже-

ний вида a @ b @ c.) Приоритет увеличивается сверху вниз для каждого нового предложения.

LALR(1)-конфликты «сдвиг/свертка» или «свертка/свертка» разрешаются выбором более приоритетного действия. Приоритет сдвига равен приоритету считываемой лексемы. Приоритет свертки равен приоритету самой правой лексемы в правиле, по которому производится свертка. Можно также явно указать приоритет свертки, написав «%prec <лексема>» справа от правила.

Добавить в язык формул операцию унарного минуса, более приоритетную, чем бинарные операции, можно следующим образом:

```
%token name
%left '+' '-'
%left '*' '/'
%left UMIN
%%
e : e '+' e | e '-' e | e '*' e | e '/' e
  | '(' e ')' | name ;
e : '-' e %prec UMIN ;
%%
```

Фиктивная лексема UMIN используется только для задания приоритета свертки по правилу e : '-' e ;

Итак, YACC разрешает конфликты (если они возникнут) по следующим правилам:

– если приоритеты альтернативных действий определены и различны, то выполняется действие с бóльшим приоритетом;

– если приоритеты действий определены и одинаковы, то в случае левой ассоциативности выбирается свертка, в случае правой ассоциативности — сдвиг, если они неассоциативны — возбуждается ошибочная ситуация;

– иначе (приоритет хотя бы одной из альтернатив не специфицирован) в случае конфликта «сдвиг/свертка» выполняется сдвиг, а в случае конфликта «свертка/свертка» — свертка по правилу, определенному выше по тексту в описании грамматики, в

обоих случаях YACC сообщает о неразрешенном конфликте в этом состоянии.

Отметим, что для конфликтной грамматики с правилами

```
s : if '(' e ')' s
  | if '(' e ')' s else s
  ;
```

предпочтение сдвига «правильно» разрешает конфликт при разборе выражения

```
if( e ) if( e ) s _ else s
```

- else будет отнесено к ближайшему if'у, как и принято в алголоподобных языках.

Для конфликтной грамматики арифметических формул эти правила приводят к вычислению выражения справа-налево без учета приоритетов операций.

8.3 Семантические действия

С каждым правилом грамматики может быть связано действие, которое будет выполнено при свертке по данному правилу. Оно записывается в виде заключенной в фигурные скобки последовательности операторов языка Си, расположенной после правой части соответствующего правила.

```
statement  : IF '(' expr ')' statement       { if_ctr++; }
           | WHILE '(' expr ')' statement    {
      while_ctr++;}
           | assign_st                       {   ass_ctr++;
      }
           ;
```

В этом примере действие if_ctr++ будет выполнено после разбора всего оператора if. При необходимости выполнить се-

мантические действия, например сразу после вычисления выражения expr, можно поместить их между символами правой части правила.

```
statement: IF '(' expr {action_1} ')'
             statement {action_2} ;
```

В этих случаях YACC автоматически преобразует грамматику, вводя дополнительные нетерминалы и соответствующие им правила с пустой правой частью. При их свертке и будут выполнены действия, расположенные между символами исходной грамматики.

```
statement: IF '(' expr void_1 ')' statement { action_2 } ;
void_1: { action_2 } ;
```

8.4 Семантический стек

Для естественного обмена данными между действиями каждый терминал или нетерминал может иметь значение. Для доступа к нему в действиях используются псевдопеременные $$ — значение левого символа правила, $<i> — значение i-ого символа правой части правила (символы нумеруются слева направо, начиная с 1). Другими словами, кроме обычного стека состояний, построенный YACC'ом анализатор содержит «семантический» стек, содержащий значения символов. Значения имеют тип YYSTYPE, который по умолчанию определяется как int. Действие

```
expr : expr '+' expr { $$ = $1 + $3; };
```

может быть использовано в интерпретаторе формул, в котором значение нетерминала «выражение» есть его вычисленное значение.

Если для правила не указано никакого действия или действие не содержит упоминания псевдопеременной $$, то значение левой части правила становится равным значению первого сим-

вола правой части, т.е. неявно выполняется действие $$ = $1; . Значение очередной лексемы копируется из переменной int yylval, в которую его обычно заносит сканер.

Различные символы грамматики могут иметь значения разных типов. Для этого следует определить тип YYSTYPE как union и специфицировать тип терминалов и нетерминалов в разделе описаний. При этом будет осуществляться контроль типов при использовании псевдопеременных, а обращение к ним будет транслироваться в обращение к соответствующему полю union.

```
%{
#define YYSTYPE yys
typedef union {
 int       intval;
 long      longval;
 nodeptr  *ptrval;
} yys;
%{
%token <intval> ICONST
%token <intval> LCONST
%type  <ptrval> expr
}
```

Если в качестве внутреннего представления программы используется дерево, удобно иметь в качестве значения нетерминала указатель на соответствующий ему узел дерева.

8.5 Кодировка лексем и интерфейс

Файл, порождаемый YACC'ом в процессе работы, содержит таблицы LALR(1)-анализатора и Си-текст функции int yyparse(void), реализующей интерпретатор таблиц и семантические действия. Для запуска парсера достаточно вызвать эту функцию. В случае успешного разбора она возвращает 0, в случае ошибки — 1.

Для получения очередной лексемы парсер вызывает функцию int yylex(void). Она должна возвратить код лексемы и поместить ее значение в переменную YYSTYPE yylval.

Код лексемы — положительное целое число. Лексемам, заданным в виде символьных констант, соответствует их код в наборе символов ЭВМ (обычно ASCII), лежащий в диапазоне 0…255. Лексемам, имеющим символические имена, присваиваются коды начиная с 257.

Выходной файл содержит операторы #define, определяющие имена лексем как их коды. Если имена лексем требуются в других файлах, следует указать ключ -d при запуске YACC'а, и он продублирует эти определения в файле y.tab.h. Этот файл следует включить в другие файлы программы (например, сканер), использующие коды лексем.

8.6 Обработка ошибок

Если анализируемое предложение не соответствует языку, то в некоторый момент возникнет ошибочная ситуация, т.е. парсер окажется в состоянии, в котором не предусмотрено ни сдвига, ни свертки для полученной ошибочной лексемы. Обычная реакция парсера — вызов функции void yyerror(const char *) с аргументом «Syntax error» и завершение работы — возврат из функции yyparse с значением 1. Реализация функции yyerror возлагается на пользователя, и он может попытаться организовать в ней выдачу более разумной диагностики (при использовании YACC-парсера это не является тривиальной задачей).

Во многих случаях желательно как-нибудь продолжить разбор. Для восстановления после ошибки YACC содержит следующие средства. Имеется специальная лексема с именем error, которая может употребляться в грамматике. При возникновении ошибки устанавливается флаг ошибки, вызывается функция yyerror, а затем из стека состояний удаляются элементы, пока не встретится состояние, допускающее лексему error. При обнаружении такого состояния выполняется сдвиг, соответствующий лексеме error в этом состоянии и разбор продолжается. Если при установленном флаге ошибки снова возникает ошибочная ситуа-

ция, то для избежания многократных сообщений yyerror не вызывается, а ошибочная лексема игнорируется. Флаг ошибки сбрасывается только после трех успешно считанных лексем.

Специальными действиями в правилах, обрабатывающих ошибочные ситуации, можно более активно вмешиваться в этот процесс.

yyerrok() — сбрасывает флаг ошибки

yyclearin() — удаляет просмотренную вперед ошибочную лексему

Макро YYERROR явным образом возбуждает ошибочную ситуацию.

Пример:

```
statement : ....
| error ';'
```

при возникновении ошибки внутри statement продолжение разбора возможно только начиная с ';' — в результате будут пропущены все лексемы до точки с запятой, которая затем будет свернута в нетерминал statement.

8.7 Разное

Запустить YACC в OS UNIX можно командой:

```
yacc [-v] [-d] имя_файла.
```

Результат работы (текст на Си) записывается в файл y.tab.c

Ключи:
v — создать текстовый файл y.output с описанием состояний и конфликтов построенного анализатора
d — создать файл y.tab.h с определениями лексем.

В версиях YACC'а для других систем имена файлов и ключей могут быть другими, например:

-i -d, имя_входного_файла.(i,c,h) в RSX11M и RT11

y.out, ytab.c ytab.h в системах с файловой системой MS-DOS

Фрагмент файла y.output:

```
state 3
      stat : expr_ (4)
      expr : expr_+ expr
      + shift 11
      . reduce 4
```

Описано состояние (state) 3, соответствующее двум основным ситуациям. В этом состоянии символ '+' вызывает сдвиг (shift) и переход в состояние 11, а любой другой символ — свертку (reduce) по правилу 4 — stat : expr.

8.8 Пример простейшего интерпретатора формул

```
%token ICONST
%left '+' '-'
%left '*' '/'

%%
p : /* empty */
  | p s
  ;

s : e '\n' { printf( "%d\n", $1 ); }
  | error '\n'
  ;
e : e '+' e    { $$ = $1 + $3; }
  | e '-' e    { $$ = $1 - $3; }
  | e '*' e    { $$ = $1 * $3; }
  | e '/' e    { $$ = $1 / $3; }
```

```
  | '(' e ')'  {$$ = $2; }
  | ICONST;
  ;
%%

#include <stdio.h>
main() { yyparse(); }
yyerror( mes ) char *mes; { printf( "%s\n", mes ); }

yylex() {
    int c, d;
    while((c=getchar())==' '); /* Skip spaces */
    if( c>='0' && c<='9' ) {   /* Integer constant */
     for( d=c-'0'; (c=getchar()) >='0' && c<='9'; ) d=d*10+c-
     '0'; ungetc( c, stdin );
     yylval = d;
     return ICONST;
    }
    return c;                /* Others */
}
```

9 ЛАБОРАТОРНЫЕ РАБОТЫ ПО ПЕРВОЙ ЧАСТИ КУРСА

9.1 Лабораторная работа 1

Цель работы

Целью данной лабораторной работы является применение на практике алгоритма рекурсивного спуска на основе библиотеки обработки входной строки.

Инструменты

Для выполнения данной лабораторной работы Вам потребуется любой компилятор языка C, способный генерировать 32-разрядные программы для той ОС, с которой Вы работаете. Для ОС Windows рекомендуется использовать бесплатную среду разработки *Microsoft Visual C++ Express*.

Основная Ваша работа будет заключаться в формировании программы перевода выражения в постфиксную запись, которая была разобрана на лекции. Для первичной обработки строк предлагается использовать модуль match.

Процедура рекурсивного спуска

```
prim()
{
    int s;
    if (match("(")) {
        if (expr()==0) return 0;
        if  (!match(")"))  {  cerror(E_RIGHT_BRACKET);
return 0; }
    }
    else if (a=numb(&s)) {
        printf("%d ", s);
    } else { cerror(E_EXPRESS_SINTAX); return 0; }
```

```
    return 1;
}
mult()
{
    if (prim()==0) return 0;
    while (1) {
        if (match("*")) {
            if (prim()==0) return 0;
            printf("* ");
        } else if (match("/")) {
            if (prim()==0) return 0;
            printf("/ ");
        } else break;
    }
    return 1;
}
add()
{
    if (mult()==0) return 0;
    while (1) {
        if (match("+")) {
            if (mult()==0) return 0;
            printf("+ ");
        } else if (match("-")) {
            if (mult()==0) return 0;
            printf("- ");
        } else break;
    }
    return 1; }
```

Модуль match

Данный модуль предназначен для первичной обработки строк и позволяет выделять из строки лексемы. Здесь описаны интерфейсные функции модуля match.

```
void match_init(char* line);
```

Процедура инициализации. Разбор строки всегда начинается с вызова этой функции. В качестве параметра передается входная строка.

```
void match_done(void);
```

Процедура деинициализации. После разбора строки необходимо вызвать данную функцию.

```
int match(char *lit);
```

Поиск указанной подстроки в начале входной строки. Если есть, то найденная подстрока удаляется и функция возвращает 1.

```
int amatch(char *lit, int len);
```

Аналогично предыдущему, но дополнительно указывается длина подстроки. Это используется для того, чтобы различать выражения типа ’for’ и ’formula’, где первое, это ключевое слова, а второе — идентификатор.

```
int symname(char *sname);
```

Ищет идентификатор (метку) во входной строке. Метка начинается с подчеркивания или буквы, далее следуют буквы, подчеркивание или цифры.

```
int number(int val[]);
```

Ищет целое знаковое число во входной строке

```
void skipblanks(void);
```

Пропускает последовательность пробельных символов (пробел, табуляция) во входной строке.

```
skipchars(void);
```

Пропускает лексемы и следующие пустые символы. Используется для обработки ошибок.

Задание

1. Реализовать программу рекурсивного спуска выражения и перевода в постфиксный вид.
2. Реализовать главную программу, считывающую строки из входного потока и выдающую в конце сообщения «OK» в случае успешного разбора или «FAIL» в случае неуспешного разбора.
3. Добавить в процедуру разбора задание по номеру варианта.
4. (Дополнительное.) Изучить разработку make файлов и разработать make файл для сборки проекта из двух модулей.
5. (Дополнительное.) Функция разбора вещественного числа в инженерной форме.

Список вариантов

1. Комплексные числа

Обеспечить поддержку вещественных и комплексных чисел. Числа вводить в нижеследующем формате

```
<вещ.число>::=<число>
<вещ.число>::=<число>.<число>
<комплксн.число>::=i
<комплксн.число>::=i+<вещ.число>
<комплксн.число>::=<вещ.число>i
<комплксн.число>::=<вещ.число>i+<вещ.число>
```

На выходе комплексные числа представлять в следующем виде:

```
complex(<комплексн.часть>,<реальн.часть>)
```

где комплексн.часть и реальн.часть — вещественные числа. Вещественные числа выводить без изменений.

Например

5.6i+4.9 на выходе complex(5.6,4.9)

0.6i на выходе complex(0.6,0)

i+7 на выходе complex(1,7)

2. Матрицы

Обеспечить поддержку матриц, состоящих из целых чисел. Матрицы вводить в нижеследующем формате:

```
<матрица>::=[<строки>]
<строки>::=<строка> | <строка> ; <строки>
<строка>::=<число> | <число> , <строка>
```

Например:
Матрица будет записана в виде [1,2,3;4,5,6]
вектор-столбец (матрица) будет записан в виде [1;4;5]
запись вида [1,2,3;5,6] будет считаться ошибочной.
На выходе матрицы представлять в виде:
matrix(<строк>,<столбцов>,<значения>)
где <строк> и <столбцов> это целые числа — количество строк и столбцов соответственно, а значения, это перечисленные через запятую слева направо и сверху вниз, значения элементов матрицы — целые числа.

3. Строки

Обеспечить поддержку строки. Строки записываются в кавычках. Дополнительно необходимо обеспечить ввод кавычки в виде последовательности из двух кавычек. На выходе строки выдавать в следующем формате:

```
string("<строка>")
```

где строка есть последовательность символов. Кавычка, если она была в исходной строке, должна быть заменена последовательностью символов \".

Например

"Hello World!" на выходе string("Hello World!")

"Hello ""World""!" на выходе string("Hello \"World!\"")

"""" на выходе string("\"")

4. Массивы

Обеспечить поддержку массивов чисел. Массивы вводить в нижеследующем формате:

```
<массив>::=( <числа> )
<числа>::=<число> , <числа> | <число>
```

На выходе массивы записывать в следующем виде:

array(<размер>, <числа>)

где, размер — это целое число — размер массива, числа — это элементы массива, перечисленные через запятую.

Например

(1, 2, 3, 4, 5) на выходе array(5,1,2,3,4,5)

5. Диапазоны

Обеспечить поддержку диапазонов (аналогично MathCAD). Диапазоны записывать в следующем формате:

```
<диапазон>::=<старт> ... <финиш>
<диапазон>::=<старт>, <приращение> ... <финиш>
```

где старт — целое число — начало диапазона, финиш — целое число — конец диапазона, приращение — целое число — приращение (по умолчанию 1 или –1, в зависимости от того, что больше старт или финиш). Приращение может быть отрицательным. Обеспечить контроль существования диапазона.

На выходе диапазоны записывать в следующем виде:

range(<старт>, <приращение>, <финиш>)

Например

5...10 на выходе range(5,1,10)

5...0 на выходе range(5,-1,0)

5,2...10 на выходе range(5,2,10)

5,-2...10 выдает ошибку «Неправильно задан диапазон»

6. Функции многих переменных

Обеспечить поддержку функций с любым количеством параметров (от 0).

```
<функция>::=<имя>(<параметры>)
<параметры>::=<выражение> | <выражение> < параметры > | e
```

На выходе функцию необходимо записывать в следующем виде:

Перечислить ее параметры слева направо (выражения должны вычисляться), а затем имя функции.

Например

pow(3,5) на выходе 3 5 pow
pow(4+1,2) на выходе 4 1 + 2 pow

Дополнительно после обработки всех строк вывести список всех функций, использованных в строках.

9.2 Лабораторная работа 2

Цель работы

Изучение принципов построения виртуальных машин (ВМ) на основе реальных процессоров.

Исходные данные

Введем следующие обозначения абстракций ВМ.

данные (value)	вещественное число (float или double)
адреса (addr)	целое 16-битное число

В связи с тем, что ОУ принимает на вход вещественные числа, нам необходимо в нашей виртуальной машине сделать поддержку вещественных чисел. Однако полностью отказываться от поддержки целочисленных операций тоже нельзя. Она потребуется, по крайней мере, для вычисления адресов. Поэтому фактически приходится в виртуальной машине делать 2 блока арифметики — целочисленный и вещественный.

Самый простой способ, как можно поступить в этой ситуации, заключается в выделении вещественной арифметики в проблемно-ориентированный блок виртуальной машины. А в системном блоке оставить только целочисленную арифметику. Так поступали практически во всех реальных вычислительных системах, так поступают обычно и при разработке ВМ.

Структура части ВМ, отвечающая за вещественную арифметику (назовем ее блок вещественной арифметики (БВА)), может выглядеть по-разному. Но в любом случае она включает некоторое вещественное арифметико-логическое устройство (ВАЛУ). Главный вопрос, который нужно решить проектировщику, — откуда ВАЛУ должно брать операнды. В зависимости от типа виртуальной машины вообще и желания проектировщика в частности, это могут быть внутренний стек данных БВА, пара (тройка) внутренних регистров БВА или заданный адрес в основной памяти виртуальной машины.

Набор проблемно-ориентированных команд у разных типов виртуальных машин принципиально различаться не будет и обычно сводится к следующим группам:

- вещественные арифметические операции (сложение, вычитание, умножение, деление);
- функции (корень квадратный, тригонометрия и так далее);
- команды управления ОУ.

Набор системных команд по группам обычно выглядит так:

- целочисленные арифметические операции (целочисленные операции сложения, вычитания, умножения, иногда деления);
- управление логикой (переходы условные и безусловные, вызовы подпрограмм, возвраты из подпрограмм);
- операции со стеком (загрузка значения (целочисленного!), запись значения с вершины стека в указанный адрес).

Задание

Разработать стековую машину.

Системная часть:

- память команд — 64 килобайта;
- стек адресов в основной памяти, обычно в конце блока;
- регистры:
  - SP — указатель стека,
  - PC — указатель команд.

Проблемно-ориентированная часть:

- память данных — в основной памяти, обычно после кода;
- стек данных — 256 ячеек данных;
- регистры:
  - SPF — указатель стека (8 бит),
  - PSW — слово состояния.

Разработка стековой виртуальной машины — одно из самых простых заданий. Все арифметические команды работают со стеком, то есть не имеют операндов. Целочисленные арифметические команды работают со стеком адресов. Вещественные арифметические команды работают со стеком данных. Команд, обладающих операндами, минимум:

Load <addr> Команда укладывает значение параметра в стек адресов.

Push <addr> Команда укладывает значение по адресу addr в стек адресов.

Pull <addr> Команда вытаскивает из стека адресов значение и записывает его по указанному в качестве параметра адресу.

10 ЛАБОРАТОРНЫЕ РАБОТЫ ПО ВТОРОЙ ЧАСТИ КУРСА

10.1 Лабораторная работа 3

Цель работы

Целью данной лабораторной работы является разработка программы рекурсивного спуска с использованием генератора лексических анализаторов — Lex.

Общие сведения

Lex — программа для генерации сканеров (лексических анализаторов). Входной язык содержит описания лексем в терминах регулярных выражений. Результатом работы LEX'а является программа на языке Си, которая читает файл yyin (обычно это стандартный ввод) и выделяет из него последовательности символов (лексемы), соответствующие регулярным выражениям.

Рассмотрим способы записи регулярных выражений во входном языке Lex'а. Алфавит Lex'а совпадает с алфавитом используемой ЭВМ. Символ из алфавита, естественно, представляет регулярное выражение из одного символа. Специальные символы (в том числе +-*?()[]{}|/\^$.<>) записываются после специального префикса \. Кроме того, применимы все традиционные способы кодирования символа в языке С. Символы и цепочки можно брать в кавычки:

Например:

a "a" \a — три способа кодирования символа a
\n \t \007 "continue"

Имеется возможность задания класса символов:

[0-9] или [0123456789] — любая цифра
[A-Za-z] — любая буква
[^0-7] — любой символ, кроме восьмиричных цифр
. — любой символ, кроме \n

Грамматика для записи регулярных выражений (в порядке убывания приоритета):

<p> : <p>*	повторение 0 или более раз
<p> : <p>+	повторение 1 или более раз
<p> : <p>?	необязательный фрагмент
<p> : <p><p>	конкатенация
<p> : <p>{m,n}	повторение от m до n раз
<p> : <p>{m}	повторение m раз
<p> : <p>{m,}	повторение m или более раз
<p> : ^<p>	фрагмент в начале строки
<p> : <p>$	фрагмент в конце строки
<p> : <p>|<p>	любое из выражений
<p> : <p>/<p>	первое выражение, если за ним следует второе
<p> : (p)	скобки, используются для группировки

Примеры:

[A-Za-z]([A-Za-z0-9]{0,7}) — идентификатор (имя) в языке C до 8 символов.

^#" "*define — начало #define в языке C

Программа на входном языке Lex состоит из трех частей, разделенных символами %%:

Описания

%%

Правила

%%

Программы

Раздел описаний содержит определения макросимволов (метасимволов) в виде:

ИМЯ ВЫРАЖЕНИЕ

Если в последующем тексте в регулярном выражении встречается {ИМЯ}, то оно заменяется на ВЫРАЖЕНИЕ. Если строка описаний начинается с пробелов или заключена в скобки %{ ... }%, то она просто копируется в выходной файл.

Раздел правил содержит строки вида

ВЫРАЖЕНИЕ {ДЕЙСТВИЕ}

ДЕЙСТВИЕ — это фрагмент программы на С, который выполняется тогда, когда обнаружена цепочка, соответствующая ВЫРАЖЕНИЮ. Действие, указанное в начале раздела без выражения, выполняется до начала разбора. Lex делает попытку выделить наиболее длинную цепочку из входного потока. Если несколько правил дают цепочку одинаковой длины, применяется первое правило. Так, при разборе по следующим правилам для цепочки «123» будет применено первое правило, а для цепочки «123.» — третье:

```
[0-9]+
(\+|\-)?[0-9]+
(\+|\-)?[0-9]+"."[0-9]+
```

Если ни одно из правил не удастся применить, входной символ будет скопирован в yyout. Если это нежелательно, в конец правил можно добавить, например, строки:

```
. {/* Ничего не делать */}
\n { }
```

Раздел программ может содержать произвольные тексты на С и целиком копируется в выходной файл. Обычно здесь записывается функция yywrap(), которая определяет, что делать при достижении автоматом конца входного файла. Ненулевое возвращаемое значение приводит к завершению разбора, нулевое —

к продолжению (перед продолжением, естественно, надо открыть какой-нибудь файл как yyin).

Интерпретатор таблиц КА имеет имя yylex(). Автомат прекращает работу (происходит возврат из функции yylex()), если в одном из действий выполнен оператор return (результат yylex() будет совпадать с указанным в операторе) или достигнут конец файла и значение yywrap() отлично от 0 (результат yylex() будет равен 0).

Традиционный пример входной программы для Lex'а — подсчет числа слов и строк в файле (обратите внимание на пробелы, этот текст нормально компилируется Lex'ом):

```
 /********** Раздел определений ***********/

 /* NODELIM означает любой символ, кроме разделителей
слов */
NODELIM [^" "\t\n]
 int l, w, c; /* Число строк, слов, символов */

%%      /********************      Раздел      правил
**************/

 { l=w=c=0; /* Инициализация */ }
{NODELIM}+ { w++; c+=yyleng; /* Слово */ }
\n { l++; /* Перевод строки */ }
. { c++; /* Остальные символы */ }

%% /*************** Раздел программ ************/

main() { yylex(); }

yywrap() {
```

```
 printf( " Lines - %d Words - %d Chars - %d\n", l, w,
c );
 return( 1 );
}
```

Внутри действий в правилах можно использовать некоторые специальные конструкции и функции Lex'а:

char* yytext — указатель на отождествленную цепочку символов, терминированную нулем;

int yyleng — длина этой цепочки

void yyless(int n) — вернуть последние n символов цепочки обратно во входной поток;

void yymore() — считать следующие символы в буфер yytext после текущей цепочки

void yyunput(char c) — поместить байт c во входной поток

ECHO — копировать текущую цепочку в yyout

В некоторых случаях бывает удобно описать необходимые действия в терминах нескольких разных состояний (т.е. разных конечных автоматов) с явным переключением с одного на другое. В этом случае набор имен состояний следует перечислить в специальной строке %Start, а перед выражениями записывать имя соответствующего состояния в угловых скобках. Переключение в новое состояние выполняется с помощью оператора BEGIN. Например, следующая программа удаляет комментарии из программ на C (out — вне комментария, in — внутри):

```
%Start out in
%%
{ BEGIN out; }
<out>"/*"{ BEGIN in; }
<out>.|\n{ printf("%s",yytext); }
<in> "*/"{ BEGIN out; }
<in> .|\n{ }
%%
```

```
yywrap() { return(1); }
main() { yylex(); }
```

Обычным приемом при разработке трансляторов с использованием Lex является следующий способ. Записываются правила для поиска нужных лексем, в действиях этих правил ставят оператор return для выхода из функции yylex(). Данный оператор должен вернуть одно из предопределенных значений. Функция или функции, организующие разбор входной строки для получения очередной лексемы, пользуются функцией yylex(), если результат этой функции (идентификатор лексемы) совпадает с ожидаемой в данной точке лексемой, то считается, что лексема получена, иначе нет. В этой технологии есть две тонкости. 1. При обработке лексемы (в действии) зачастую требуется запомнить значение полученной лексемы, для того чтобы передать её в вызывающую функцию. 2. Всегда требуется помнить текущую лексему, так как если ее значение не совпало с требуемым, то это еще не значит, что выявлена ошибка. Вполне вероятно, что в другом месте программы (в другой функции по иерархии или в следующей проверке в этой же функции) полученное значение лексемы окажется именно искомым.

В данной лабораторной работе необходимо пользоваться именно этим механизмом.

Для вызова программы Lex следует ввести команду «lex имя_исходного_файла». Выходной файл по умолчанию имеет имя «lex.yy.c».

Задание

Ваша задача сформировать регулярную грамматику для языка из лабораторной работы №1, записать ее в виде исходного текста для lex и сгенерировать программу, на вход (входной поток) которой подается последовательность строк, предположительно принадлежащих данной грамматике. Каждая строка заканчивается символом \n. Программа должна выдать результат в виде числа (0 или 1) для каждой строки о ее принадлежности к грамматике.

10.2 Лабораторная работа 4

Цель работы

Целью данной лабораторной работы является разработка калькулятора на базе алгоритма рекурсивного спуска с использованием генератора лексических анализаторов — Lex.

Общие сведения

Существует два вида трансляторов — интерпретаторы и компиляторы. Первые из них выполняют программу сразу же в процессе ее разбора. Соответственно, если в программе есть какая-то синтаксическая или другая ошибка, интерпретатор обнаружит ее, только дойдя до этой строчки. Это одновременно и плюс, и минус. Плюс в том, что теоретически можно предложить пользователю тут же исправить ошибку и продолжить дальше (многие интерпретаторы, снабженные средой разработки, так и поступают, например старые версии языка Basic). Компилятор, напротив, сперва переводит весь текст программы в последовательность команд некоторой реальной или виртуальной машины, а затем процессор или виртуальная машина (программа) исполняет эту последовательность.

В данной работе Вам предстоит разработать калькулятор — это такая программа, которая позволяет производить вычисления арифметических выражений. Всего есть два варианта реализации калькулятора — интерпретатор и компилятор. В первом случае код, отвечающий за вычисления, должен находиться непосредственно в алгоритме рекурсивного спуска. Например, так:

```
add()
{
    double op1,op2;
    if (mult()==0) return 0;
    while (1) {
        if (match("+")) {
```

```
            if (mult()==0) return 0;
            op1 = pull();
            op2 = pull();
            push(op2+op1);
        } else if (match("-")) {
            if (mult()==0) return 0;
            op1 = pull();
            op2 = pull();
            push(op2-op1);
        } else break;
    }
    return 1;
}
```

Для компилятора все чуть-чуть сложнее. Вместо вычислений нужно генерировать некоторый исполняемый код. Будем считать, что генерируем код для некоторой виртуальной машины, поддерживающей некоторый небольшой набор операций.

```
add()
{
    double op1,op2;
    if (mult()==0) return 0;
    while (1) {
        if (match("+")) {
            if (mult()==0) return 0;
            put_commd(&ptr, CADD);
        } else if (match("-")) {
            if (mult()==0) return 0;
            put_commd(&ptr, CSUB);
        } else break;
```

```
 }
 return 1;
}
```

где функция put_cmd размещает в буфере или выводит во внешний (бинарный) файл код этой команды. Примеры системы команд, реализации виртуальной машины, а также пример функций кодогенератора приведены в архиве lab3vm.rar. Эта виртуальная машина поддерживает работу с вещественными числами, строками и переменными. А также команды ввода и вывода. Поддерживаются 4 стандартные математические операции.

Задания

1. Разработать калькулятор-интерпретатор. Поддерживаемые операции — из предыдущей лабораторной работы + (обязательно для всех) обеспечить работу с переменными, добавить операцию (или оператор) присваивания. Те, у кого был вариант №4, поддерживаемые функции берут из варианта № 3. Обеспечить консольный ввод и вывод переменных. Текст разбираемой программы теперь многострочный. Необходимо также обеспечить возможность записи комментариев языка Си /* … */.
2. Разработать калькулятор-компилятор. Поддерживаемые операции — из предыдущей лабораторной работы. Те, у кого был вариант №4, поддерживаемые функции берут из варианта №3. Обеспечить консольный ввод и вывод переменных. Текст разбираемой программы теперь многострочный. Если в виртуальной машине нет какой-то необходимой вам операции (например, возведение в степень) — добавьте ее. Необходимо также обеспечить возможность записи комментариев языка С++ (от // … и до конца строки). Обратите внимание, что виртуальная машина — это другая исполняемая программа. И передача информации от вашего компилятора к виртуальной машине осуществляется через внешний двоичный файл.

11 КУРСОВОЙ ПРОЕКТ

Заданием на курсовой проект является разработка транслятора одного из семи языков. Все задания на курсовой проект разделяются на четыре группы:

Тип транслятора:

1. Интерпретатор.
2. Компилятор и виртуальная машина.

Способ реализации транслятора:

- Рекурсивный спуск
- Генератор лексического и синтаксического анализатора (lex+yacc или какой-либо другой генератор, который Вам больше нравится).

Ниже приведена таблица, по которой Вы можете узнать тип языка, тип транслятора и способ реализации Вашего транслятора по номеру варианта. А далее описаны семь вариантов языка, один из которых Вам необходимо реализовать.

Результатом Вашей работы, которую Вы пришлете на проверку, должны быть:

- Исходный текст Вашего транслятора: все *.c, *.cpp, *.h файлы, исходные тексты на lex и yacc (если Вы реализуете данный вариант) и виртуальной машины, если Вы реализуете компилятор и виртуальную машину.
- Не менее трех примеров корректного исходного текста, то есть такого текста, который обрабатывается Вашим транслятором как правильный.
- Не менее трех примеров некорректного исходного текста, то есть такого текста, который корректно обрабатывается Вашим транслятором как неправильный, то есть выдает ошибки.
- Makefile или пакетный файл, при помощи которого происходит сборка вашего проекта и прогон тестовых примеров.
- Сопровождающий readme.txt, в котором описано назначение каждого файла, а также для каждого тестового примера описан результат, который выдает Ваш транслятор. Обязательно необходимо указать, какими компиля-

торами (C/C++, а также lex и yacc) и какой версии Вы пользовались для сборки Вашего проекта.

- Документ, оформленный обычным образом, как курсовой проект. Данный документ должен включать:
  - описание грамматики;
  - описание семантики операторов и операций Вашего языка.

11.1 Варианты курсовых проектов

№ варианта	Вариант синтаксиса	Тип транслятора	Способ реализации
1	1	Интерпретатор	Рекурсивный спуск
2	2	Компилятор	Генератор анализаторов
3	3	Интерпретатор	Рекурсивный спуск
4	4	Компилятор	Генератор анализаторов
5	5	Интерпретатор	Рекурсивный спуск
6	7	Интерпретатор	Рекурсивный спуск
7	1	Компилятор	Генератор анализаторов
8	2	Интерпретатор	Рекурсивный спуск
9	3	Компилятор	Генератор анализаторов
10	4	Интерпретатор	Рекурсивный спуск
11	5	Компилятор	Генератор анализаторов
12	6	Интерпретатор	Рекурсивный спуск
13	7	Компилятор	Генератор анализаторов
14	1	Интерпретатор	Генератор анализаторов
15	2	Компилятор	Рекурсивный спуск
16	3	Интерпретатор	Генератор анализаторов
17	4	Компилятор	Рекурсивный спуск
18	5	Интерпретатор	Генератор анализаторов
19	7	Интерпретатор	Генератор анализаторов
20	1	Компилятор	Рекурсивный спуск
21	2	Интерпретатор	Генератор анализаторов
22	3	Компилятор	Рекурсивный спуск
23	4	Интерпретатор	Генератор анализаторов
24	5	Компилятор	Рекурсивный спуск
25	6	Интерпретатор	Генератор анализаторов
26	7	Компилятор	Рекурсивный спуск

11.2 Варианты синтаксиса

Вариант № 1. Pascal-подобный алгоритмический язык

В данном задании необходимо реализовать относительно простой язык программирования. За основу возьмите синтаксис языка Паскаль. Вам необходимо реализовать поддержку нижеследующих элементов:

- Стандартная структура программы языка Pascal (program name; переменные, подпрограммы, begin … end).
- Типы данных real, boolean.
- Подпрограммы: процедуры, функции. Можно не делать поддержку вложенных процедур и функций, достаточно будет обеспечить поддержку процедур и функций на верхнем уровне (уровне программы). Обязательно необходимо обеспечить поддержку локальных переменных в подпрограммах. Для проверки реализуйте на Вашем языке рекурсивный алгоритм вычисления чисел Фибоначчи, он должен работать корректно.
- Операторы:
  - for … to … [step …] do
  - if … then … [else …]
- Системные процедуры:
  - write, writeln: достаточно будет, если эти процедуры будут принимать на вход ровно два параметра — строку и вещественное выражение для печати.
  - read, readln: эти процедуры должны принимать ровно один параметр: переменную для ввода значения.
  - Оператор присваивания. Слева от символа :=, а справа выражение с операциями, которые описаны ниже.
- Операции:
  - Арифметические (для вещественных чисел): +, -, *, /.
  - Сравнения (операнды могут быть вещественного типа, а результат операции булев): <, >, <>, ><, <=, =<, >=, =>, =
  - Логические (операнды булевы, результат булев): and, or, xor

Примечание: Ваш транслятор должен быть реализован как консольное приложение, принимающее на вход (не в качестве входного потока, а как параметр приложения) исходный текст Вашей программы. Процедуры read/readln и write/writeln должны работать соответственно с входным и выходным потоками, что позволит делать автоматическое тестирование на основе заранее заготовленных примеров. Сообщения об ошибках должны попадать в поток stderr (для языка C) или cerr (для C++). В случае если была хотя бы одна ошибка, программа должна возвращать значение –1, а если ошибок не было, то 0.

Вариант № 2. Экранный калькулятор выражений

В данном задании Вам необходимо реализовать простой калькулятор выражений, аналогичный тому, что Вы делали в рамках лабораторных работ. Однако ввод выражения может осуществляться при помощи экранной клавиатуры, похожей на ту, что можно увидеть у обычного калькулятора. Также необходимо обеспечить:

1) возможность загрузки готового выражения из файла;
2) выгрузки результата вычисления в файл;
3) копирование в буфер обмена и из буфера обмена;
4) возможность ввода многострочного выражения;
5) операции +, -, *, /, возведение в степень;
6) нахождение корня квадратного и корня любой степени;
7) основные тригонометрические функции, sin, cos, tg, ctg, arcsin, arcos, arctg, arcctg;
8) гиперболические функции;
9) скобки;
10) поддержку работы с переменными: для этого нужно обеспечить операцию присваивания, все переменные с их значениями должны отображаться в виде таблицы в калькуляторе, количество переменных может быть ограничено.

Примечание: если Вам необходимо реализовать данный калькулятор на основе lex+yacc, то необходимо подменить в lex функции ввода символа. Сделать это можно, например, следующим образом:

```
int get_next_simbols(char* buf,int max_size)
{
        if (max_size == 0) return 0;
        if (*bufptr  == 0) return 0;
        *buf = *bufptr++;
        return 1;
}
#define YY_INPUT(buf,result,max_size) \
        { \
                result = get_next_simbols(buf,max_size); \
        }
```

Данный текст, вставленный в секцию определений в скобки %{ … %}, позволит подменить ввод символов из входного потока на ввод символов из буфера bufptr, в который Вы должны загрузить буфер с символами до начала разбора.

Вариант № 3. Консольный текстовый процессор

Задача данного текстового процессора — обработка строк входного файла. Проблемной частью команд виртуальной машины, встроенной в интерпретатор или внешней (для компилятора), будут являться команды навигации по тексту и обработки текста, как это обычно происходит в обычном текстовом редакторе, однако в отличие от обычного текстового редактора никакого графического или даже консольного отображения данного текста не требуется. Вся обработка должна происходить в оперативной памяти. Ближайший аналог такого текстового процессора — sed.

Команды навигации:

- перейти в начало файла, в конец файла, в начало строки, в конец сроки;
- перейти на строку N (по возможности сохранив текущую позицию);
- перейти на позицию N в текущей строке;
- перейти на N строк ниже, на N строк выше;
- перейти на N символов влево, на N символов вправо;

- перейти на N слов влево, на N слов вправо.

Команды работы с блоками:

- начать выделение (затем после нескольких команд перехода), закончить выделение;
- выделить текущий символ;
- выделить текущее слово;
- выделить текущую строку;
- выделить текущую строку от начала до текущей позиции;
- выделить текущую строку от текущей позиции до конца;
- запомнить блок (аналогично копированию в буфер обмена);
- вставить блок в текущую позицию (аналогично вставке из буфера обмена).

Команды редактирования:

- удалить текущий символ или выделенный блок;
- вставить заданную последовательность символов.

Поиск:

- найти заданную строку от начала файла;
- найти заданную строку от текущей позиции до конца документа;
- найти заданную строку от текущей позиции до начала документа;
- найди следующее вхождение.

Примечание 1:

- не важно, как эти команды будут называться — придумайте разумные, понятные Вам названия;
- если блок уже выделен (то есть закончено выделение блока), то любая операция навигации сбрасывает выделенный блок (разумеется, это не касается блока, скопированного в буфер;
- было бы неплохо сделать поддержку нескольких блоков, но это на Ваше усмотрение;

- в качестве дополнительного параметра для команд поиска можно указать — требуется ли найденную строку выделить как блок;
- каждая команда возвращает значение — удалось ли ей выполнить задачу, это значение может быть использовано в команде if или проигонорировано.

Общие команды:

- Команда if — проверяет, успешно ли завершилась предыдущая команда, и позволяет выполнить какой-то блок действий в случае успеха. Конкретный синтаксис команды остается на Ваше усмотрение.
- Цикл пока. По синтаксису данная команда должна быть похожа на команду if, но все команды, включенные в блок, выполняются пока выполняется условие (по желанию).

Примечание 2: Ваш транслятор должен быть реализован как консольное приложение, исходный текст на Вашем языке передается в качестве параметра транслятора (см. примечание для варианта №1). Обрабатываемый текстовый файл подается на входной поток, а обработанный файл, который получается после работы Вашей программы, выдается на выходной поток.

Вариант № 4. Графический язык

В данном варианте Вашей задачей является создание языка, который описывает в виде алгоритмов и графических примитивов векторный рисунок (приблизительно на таких принципах строится, например, отображение TTF шрифтов). Входным файлом для данного транслятора будет являться исходный текст на описанном здесь языке, а выходным — либо окно с нарисованным рисунком, либо графический файл одного из популярных форматов (BMP, PNG etc). См. Примечание к варианту языка № 2.

Графические команды:

- задать параметры вывода:
  - пера (толщина, цвет);

  - кисти (тип, цвет);
  - вывода текста (шрифт, размер);
- сохранить параметры вывода (в стек);
- восстановить параметры вывода (из стека);
- переместить перо в заданную позицию (x, y);
- переместить перо на заданную позицию (дельта x, дельта y);
- нарисовать линию от текущей позиции до указанной (перо перемещается в эту позицию);
- нарисовать линию от текущей позиции до позиции в виде смещения от текущей позиции (перо перемещается в эту позицию);
- нарисовать прямоугольник с заданными координатами;
- нарисовать прямоугольник от текущей позиции до позиции, заданной в виде смещения;
- нарисовать эллипс с заданными координатами;
- нарисовать эллипс от текущей позиции до позиции, заданной в виде смещения;
- нарисовать ломанную линию (каждая узловая точка ломанной задается в виде смещения предыдущей точки).

Команды и конструкции общего назначения:
- задать глобальную переменную;
- вычислить выражение (стандартный минимум арифметических операций);
- повторить блок N раз (аналог цикла for);
- оператор if (конкретный синтаксис остается на Ваше усмотрение);
- оператор while (конкретный синтаксис остается на Ваше усмотрение);
- процедура (подпрограмма);
- обеспечить работу с локальными переменными.

Примечания:

- все замкнутые фигуры, такие, как прямоугольник, эллипс и т.д., рисуются с учетом текущей кисти, но она может иметь тип «нет заливки», тогда фигура должна рисоваться без заливки;
- в качестве любого параметра проблемно-ориентированных команд могут использоваться выражения.

Вариант № 5. Язык управления «черепахой»

Данный язык является вариацией на тему известного языка для обучения программированию Лого. Рисовать «черепашку» можно в виде простого треугольничка. См. Примечание к варианту языка № 2.

Для данного варианта Ваш интерпретатор или виртуальная машина для компилятора должны представлять собой GUI-приложение, в котором требуется обеспечить как минимум команду открытия файла (бинарного для компилятора и текстового для интерпретатора). Также требуется обспечить возможность указания параметра задержки в миллисекундах, который будет использоваться для отображения после выполнения каждой команды, а также требуется отобразить и дать возможность изменить любой параметр — азимут «черепашки», ее текущие координаты. На выходе Вы должны получить, в отличие от всех остальных вариантов, не файл, а процесс движения «черепашки».

Команды «черепашки»:

- опусти хвост (после этого в процессе движения «черепашки» за хвостом остается линия), параметры — цвет и толщина линия;
- подними хвост (линия не остается);
- повернись на заданный азимут (азимут задается в градусах от севера, север — направление вверх);
- повернись на заданный угол (то есть к текущему азимуту нужно прибавить указанный);
- подвинься на заданное число шагов вперед (в ту сторону, куда она смотрит);
- функция «впереди след хвоста».

Команды и конструкции общего назначения:

- создать переменную;
- вычислить выражение (стандартный минимум арифметических операций);
- условный оператор;
- цикл «пока»;
- подпрограмма (желательно, но не обязательно, обеспечить работу с локальными переменными).

Примечание: в качестве относительно простого примера Вы можете реализовать построение какого-то лабиринта, а в качестве более сложного примера — выход из любого лабиринта, с любой позиции.

Вариант № 6. Интерпретатор командной строки

Основным назначением данного языка должна стать замена привычного Вам cmd.exe. command.com или shell. Соответственно, основными абстракциями, с которыми работает данный интерпретатор, являются внешние программы, которые он может выполнять, передавая им параметры командной строки, и получать от них результат в виде кода завершения. Для примера можете изучить работу csh. Настоятельно рекомендую Вам реализовать сборочный скрипт на Вашем языке, вместо использования обычных BAT или CMD файлов.

Необходимо обеспечить:

- запуск произвольного приложения со всеми его параметрами;
- работу перенаправления ввода-вывода для входного потока, выходного потока и потока вывода ошибок, в том числе работу канала передачи выхода одного приложения входу другого при помощи «|»;
- сохранение результата работы программы (кода завершения) в переменную;
- операции над переменными:
  - конкатенацию переменных (то есть работу с ними как со строками);

  - арифметические и логические операции над переменными (то есть работу с ними, как с числами, стандартный минимум арифметических операций);
- печать на экране (echo);
- поддержку параметров скриптового файла (то есть возможность передачи внешних параметров при запуске Вашего интерпретатора) и использование этих параметров внутри скрипта как переменных;
- работу подпрограмм и локальных переменных;
- условный оператор;
- цикл «пока».

Вариант № 7. С-подобный алгоритмический язык

Требуется создать алгоритмический язык программирования, с синтаксисом, аналогичным языку С. Вам необходимо реализовать поддержку нижеследующих элементов.

- Стандартная структура программы языка С (глобальные переменные, функции, главная функция с именем main).
- Типы данных int и указатель на int.
- Функции. Обязательно необходимо обеспечить поддержку локальных переменных в функциях. Для проверки реализуйте на Вашем языке рекурсивный алгоритм вычисления чисел Фибоначчи, он должен работать корректно.
- Операторы:
  - for
  - if … [else …]
- Системные процедуры:
  - print: достаточно, если эта функция будет принимать на вход ровно два параметра — строку и целочисленное выражение для печати;
  - scanf: эта функция должна принимать ровно один параметр: переменную для ввода значения.
- Операции:
  - Арифметические: +, -, *, /, %.

- Сравнения (операнды могут быть целого типа, результат целого типа): <, >, !=, <=, >=, ==
- Бинарные: &, |, ^, ~
- Логические (операнды булевы, результат булев): &&, ||, ^^, !
- Операции с указателем: *, &

Примечание:

- поддержка препроцессора и заголовочных файлов не требуется.

ЛИТЕРАТУРА

1. Пратт Теренс, Зелковец Марвин. Языки программирования: разработка и реализация. — 4-е изд. — СПб.: Питер, 2002. — 688 с.

2. Компиляторы: принципы, технологии и инструментарий / Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. — 2-е изд. — М.: «Вильямс», 2008.