Н.Н. СМИРНОВА, С.Д. ТАРАСОВ

ОСНОВЫ ПОСТРОЕНИЯ КОМПИЛЯТОРОВ

Министерство образования и науки Российской Федерации
Балтийский государственный технический университет «Военмех»
Кафедра информационных систем и компьютерных технологий

Н.Н. СМИРНОВА, С.Д. ТАРАСОВ

ОСНОВЫ ПОСТРОЕНИЯ КОМПИЛЯТОРОВ

Учебно-практическое пособие

Санкт-Петербург
2007

УДК 004.4' 422(075.8)
С50

Смирнова, Н.Н.

С50 Основы построения компиляторов: учебно-практ. пособие / Н.Н. Смирнова, С.Д. Тарасов; Балт. гос. техн. ун-т. – СПб., 2007. – 63 с.

ISBN 5-85546-284-6

Рассмотрены теоретические и практические аспекты построения компиляторов. Приведены основные алгоритмы, применяемые при построении лексических, синтаксических и семантических анализаторов, а также основные принципы генерации кода. Представлена практическая реализация лексического и синтаксического анализа, генератора кода и вычислителя арифметических выражений. Рассмотрены вопросы согласования и взаимодействия различных частей компилятора. В качестве примера практической реализации на языке C++ с использованием библиотеки классов представлен компилятор для простейшего языка.

Предназначено для студентов специальности «Автоматизированные системы обработки информации и управления», а также для магистрантов и аспирантов.

УДК 004.4' 422(075.8)

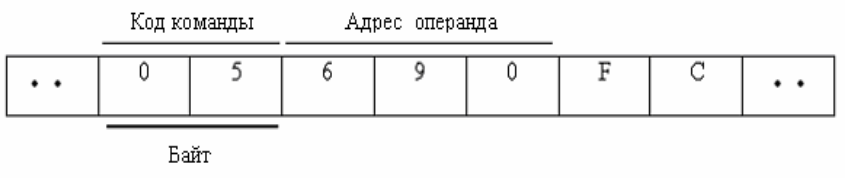
Р е ц е н з е н т д-р техн. наук, проф. БГТУ *А.Д. Ледовский*

Утверждено
редакционно-издательским
советом университета

ISBN 5-85546-284-6

ВВЕДЕНИЕ

Первые программы, которые создавались для ЭВМ первого поколения, писались непосредственно на языке машинных кодов. Фрагмент такой программы представлен ниже. В данном случае используется трехбайтная команда сложения содержимого регистра и ячейки памяти, один байт которой содержит код команды, а последующие два – двухбайтное значение операнда.



Написание программ в машинных кодах стало практически невозможным по мере усложнения алгоритмов и увеличения количества команд в программе. Развитие аппаратуры вычислительных систем также усложняло ситуацию: написанные в машинных кодах программы трудно было перенести с одного компьютера на другой, так как они содержали не просто алгоритмические инструкции, а непосредственно команды конкретного процессора. Из этого следовал вывод, что человек, даже если он специалист по вычислительной технике, не может и не должен говорить на языке машинных команд. Однако попытки научить ЭВМ говорить на человеческом языке не увенчались успехом. Поэтому все дальнейшее развитие программного обеспечения компьютеров неразрывно связано с возникновением и развитием компиляторов.

Первыми компиляторами были компиляторы с языков ассемблера или, как они назывались, мнемокодов. Мнемокоды значительно упростили процесс написания программ, заменив шестнадцатеричные коды команд процессора на более или менее доступный язык мнемонических обозначений. Например, рассмотренную выше команду сложения с регистром можно записать как

ADD AX, 669 h

Такая инструкция была уже более содержательной и, кроме того, не так жестко привязанной к типу процессора. Например, *ADD AX, 669 h* характерна для всех процессоров 86 семейства.

Однако исполнять сам мнемокод (язык ассемблера) ни один компьютер не способен, поэтому возникла необходимость в создании компилятора, переводящего текст программы, написанный на языке ассемблера, в машинные коды. Эти компиляторы достаточно просты, так как команды ассемблера представляют собой примитивные синтаксические конструкции, каждой из которых ставится в соответствие, как правило, одна машинная команда.

Задача компиляции ассемблерных программ усложнилась с появлением макрокоманд, которые в общем случае должны были быть приведены в произвольное число машинных инструкций.

Следующим этапом стало создание языков высокого уровня – большинство современных языков программирования, – представляющих собой некоторое промежуточное звено между чисто формальными языками и языками естественного общения людей. От первых им досталась строгая формализация синтаксических структур (предложений) языка, от вторых – значительная часть словарного запаса, семантика основных конструкций и выражений (с элементами математических операций, пришедшими из алгебры).

Появление языков высокого уровня (их часто называют алгоритмическими) существенно упростило процесс программирования, позволяя программисту сконцентрироваться на алгоритме решаемой задачи. Улучшилась и ситуация с переносимостью программ. Теперь в задачу компилятора входило правильно транслировать алгоритмические инструкции в машинные команды конкретной архитектуры процессора.

Как только возникла массовая потребность, стала развиваться теория формальных языков и грамматик, а также были разработаны алгоритмы генерации машинных кодов, оптимизации и распределения памяти. В настоящее время эта область дополняется новыми теоремами, алгоритмами, техническими решениями. Многие из них реализовались в коммерческих компиляторах ведущих фирм мира: Intel, Sun, Microsoft. Со временем компиляторы как автономные транслирующие программы были дополнены текстовыми редакторами, системами отладки, средствами поша-

гового выполнения программ, средствами визуального программирования и т.д. Такие системы получили название «системы программирования». Наиболее известные из них – Microsoft Visual Studio, Borland Delphi, KDevelop.

В настоящее время компиляторы являются неотъемлемой частью любой вычислительной системы. Программирование же непосредственно на языках машинных кодов происходит исключительно редко и только для решения очень узкого круга задач.

1. ОСНОВНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ КОМПИЛЯТОРОВ

1.1. Трансляторы, интерпретаторы, компиляторы

Транслятор – это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на результирующем (выходном) языке. Схема работы транслятора приведена на рис. 1

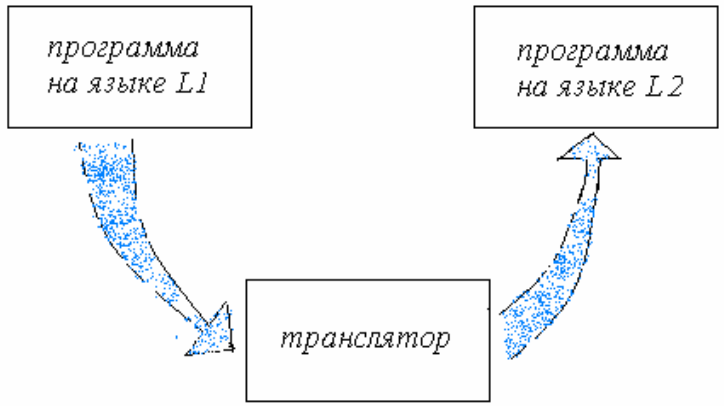


Рис. 1

Входной файл должен содержать текст программы, удовлетворяющий лексическим, синтаксическим и семантическим требованиям входного языка *(L1)*. Результирующая программа стро-

ится в соответствии с лексическими, синтаксическими и семантическими правилами выходного языка *(L2)*. Важным требованием в определении транслятора является эквивалентность входной и выходной программ. Эквивалентность означает совпадение смысла программ с точки зрения семантики входного (для исходной программы) и выходного языка (для результирующей программы). Без выполнения этого требования транслятор теряет всякий практический смысл.

Результатом работы транслятора будет программа на выходном языке только в том случае, если текст исходной программы является правильным, т.е. не содержит лексических, синтаксических и семантических ошибок. Если программа содержит хотя бы одну ошибку, то результатом работы транслятора должно быть сообщение об ошибке* (возможно, с дополнительными пояснениями и указанием места ошибки в тексте входной программы и т.д.).

Кроме понятия «транслятор» употребляется также и близкое ему по смыслу понятие «компилятор».

Компилятор – это транслятор, который осуществляет перевод исходной программы в эквивалентную ей объектную программу на языке машинных команд или на языке ассемблера. Таким образом, компилятор является частным случаем транслятора, так как его выходной язык *(L2)* строго фиксирован как «объектный код» или код на языке ассемблера. Однако во многих литературных источниках эти два понятия (транслятор и компилятор) не разделяются между собой.

Результирующая программа компилятора называется объектной программой или объектным кодом. Даже в том случае, когда результирующая программа порождается на языке машинных команд, между объектной программой (объектным файлом) и исполняемой (исполняемым файлом) есть существенная разница. Порожденная компилятором программа не может непосредственно выполняться на компьютере, так как она не привязана к конкретной области памяти, где должны располагаться ее код и данные, и содержит только ссылки на «внешние функции», реализованные во «внешних» библиотечных файлах. Объектный код

* Не все семантические ошибки можно выявить на этапе трансляции. Многие из них проявляются только на этапе выполнения. Это особенно характерно для языка Си.

должен быть обработан редактором связей и компоновщиком для подключенного объектного кода в исполняемый файл.

Компиляторы, безусловно, самый распространенный вид трансляторов. Они имеют самое широкое практическое применение, которым обязаны большому числу различных языков программирования (Pascal, C++, C и др.)

Интерпретатор – это программа, которая воспринимает входную программу на исходном языке и выполняет ее.

Интерпретатор, так же как и транслятор, анализирует текст исходной программы, однако он не порождает результирующую программу, а сразу же выполняет исходную в соответствие с ее смыслом, заданным семантикой входного языка. В этом принципиальная разница между транслятором и интерпретатором. К интерпретируемым языкам относят Basic, а также скриптовые языки Perl, PHP.

Следует отметить, что в современных системах программирования стали появляться компиляторы, в которых результирующая программа создается не на языке машинных команд и не на ассемблере, а на некотором промежуточном языке. Сам по себе он не может непосредственно исполняться на компьютере, а требует специального промежуточного интерпретатора для выполнения написанных на нем программ. Примерами могут служить компиляторы с языков Java, C#, Manag C++, требующие некой «виртуальной машины» для выполнения полученного промежуточного кода.

1.2. Этапы компиляции. Общая схема работы компилятора

Процесс компиляции можно разделить на два основных этапа: анализ и синтез (рис. 2). На этапе анализа выполняется распознавание текста исходной программы, создание и заполнение таблиц идентификаторов и т.д. Результатом является некоторое внутреннее представление программы, понятное компилятору. На этапе синтеза на основании внутреннего представления программы и информации, содержащейся в таблице идентификаторов, порождается текст результирующей программы. Результатом этого этапа является объектный код (см. рис. 2).

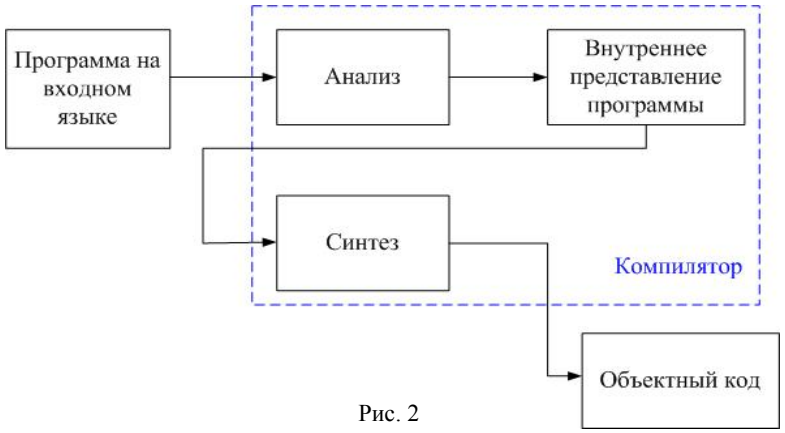


Рис. 2

Кроме того, в состав компилятора входит часть, ответственная за анализ и исправление ошибок. При наличии ошибки в тексте исходной программы она должна максимально полно информировать пользователя об ошибке и месте ее возникновения.

Этапы анализа и синтеза, в свою очередь, состоят из более мелких этапов, называемых фазами компиляции (рис. 3). Рассмотрим основные из них.

Лексический анализ – относительно простая фаза, в которой формируются символы (*tokens*) входного языка. Слова языка («*FOR», «DO», «WHILE»*), или пользовательские идентификаторы («*my_var», «i»*), или знаки препинания («+», «–» , «++», «= =») являются неделимыми лексическими единицами (лексемами), и их удобно воспринимать как один символ.

Лексема (*token*) – лексическая единица языка. Это структурная единица, которая состоит из элементарных символов языка и не содержит в своем составе других лексем.

Задача фазы лексического анализа или лексического анализатора (*scanner*) – переход от последовательности знаков к символам языка (лексем), с которыми в дальнейшем будут работать синтаксическая и семантическая фазы. Наряду с преобразованием последовательности знаков в символы лексический анализатор также обрабатывает пробелы и удаляет комментарии и любые другие символы, не имеющие смысловой нагрузки для последующих этапов анализа. Важно отметить, что лексический анализатор всего лишь формирует символы – их последовательность не имеет для него никакого значения.

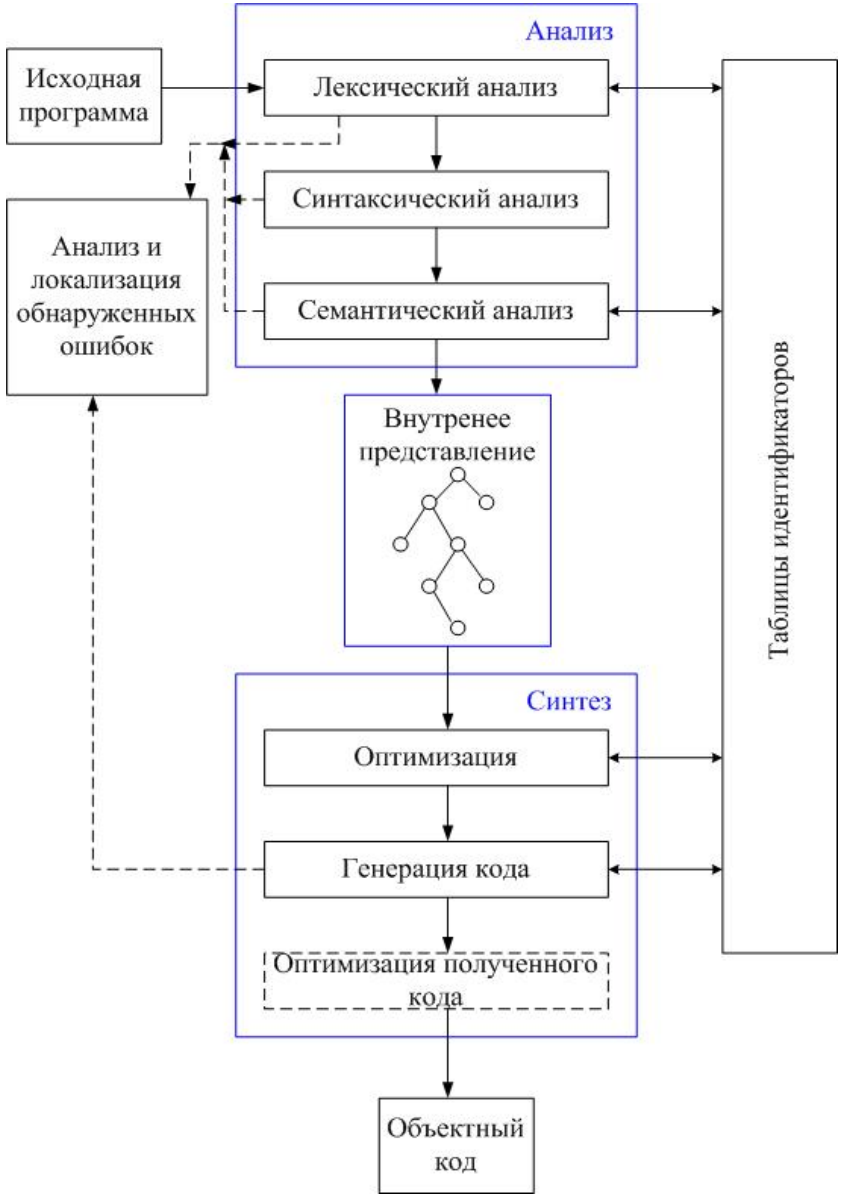

Анализ
Исходная программа
Лексический анализ
Синтаксический анализ
Семантический анализ
Анализ и локализация обнаруженных ошибок
Внутренее представление
Таблицы идентификаторов
Синтез
Оптимизация
Генерация кода
Оптимизация полученного кода
Объектный код

Рис. 3

Лексический анализатор также строит различные таблицы, среди которых следует выделить таблицы идентификаторов.

Таблицы идентификаторов – это специальным образом организованные наборы данных, служащие для хранения информации об элементах исходной программы, которые затем используются для порождения теста результирующей программы. В конкретной реализации компилятора могут быть одна или несколько таблиц идентификаторов. Элементами исходной программы, информацию о которых нужно хранить в процессе компиляции, являются переменные, константы, функции и т.д. Конкретный состав набора элементов зависит от используемого входного языка.

Синтаксический анализ – основная часть компиляции на этапе анализа. Синтаксический анализатор (*parser*) получает на вход результат работы лексического анализатора и разбирает его в соответствии с некоторой грамматикой. Основная задача синтаксического анализа – определить, принадлежит ли исходная цепочка символов (лексем) входному языку, т.е. является правильным (с точки зрения синтаксиса) предложением входного языка. Синтаксический анализ – одна из наиболее формализованных и хорошо изученных фаз компиляции.

Результатом работы синтаксического анализатора является некоторое промежуточное представление программы, обычно в виде дерева синтаксического разбора.

Семантический анализ обычно заключается в проверке правильности типов данных, используемых в программе. Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка. В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных: для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение. Другой пример контекстного условия – число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры. Такие контекстные условия не всегда могут быть проверены во время синтаксического анализа и потому обычно выделяются в отдельную фазу.

Оптимизация заключается в преобразовании промежуточного представления программы с целью повышения эффективности результирующего объектного кода. Нужно отметить, что существуют различные критерии эффективности, например скорость

исполнения или объем памяти, запрашиваемый программой. Очевидно, что все преобразования, осуществляемые на фазе оптимизации, должны приводить к программе, эквивалентной исходной. К основным видам оптимизации относят константные вычисления, уменьшение силы операций, выделение общих подвыражений, чистку циклов.

Генерация кода непосредственно связана с порождением команд, составляющих предложение выходного языка и в целом текст результирующей программы. Это основная фаза на этапе синтеза результирующей программы. Помимо собственно генерации кода на этом этапе необходимо решить множество сопутствующих проблем, например:

- распределение памяти, т.е. отображение имен исходной программы в адреса памяти;
- распределение регистров, т.е. определение для каждой точки программы множества переменных, которые должны быть размещены в регистрах;
- выбор такой последовательности записи значений в регистры, которая избавила бы от необходимости частой выгрузки и повторной загрузки.

Оптимизация полученного кода – необязательная фаза (ее задачи могут решаться на стадии генерации) оптимизации полученного кода с учетом конкретного набора регистров, скорости выполнения различных машинных команд и т.д.

1.3. Синтаксически управляемая трансляция

Представленные на рис. 3 фазы компиляции не всегда выполняются последовательно. Так как число проходов (просмотров исходного текста входной программы) важно с точки зрения скорости компиляции, на практике фазы стараются совместить так, чтобы это число было минимальным (в простейшем случае может быть реализован простейший компилятор). Так как основным в процессе компиляции является синтаксический анализ, широкое распространение получила так называемая «синтаксически управляемая трансляция» (СУТ). Сама по себе она базируется на достаточно сложном теоретическом базисе синтаксически управляемых определений (СУО), представляющих собой некоторую совокупность контекстно-свободной грамматики и множества семантических правил. Теоретические аспекты СУО и СУТ

подробно изложены в [1]. С практической же точки зрения синтаксически управляемая трансляция подразумевает, что основным управляющим элементом компилятора является синтаксический анализатор (СА). Лексический анализатор служит неким «поставщиком» потока лексем для синтаксического анализатора. Семантические процедуры, а также процедуры генерации кода применяются к построенному СА дереву разбора и под его управлением. Общий вид схемы трансляции приведен на рис. 4.

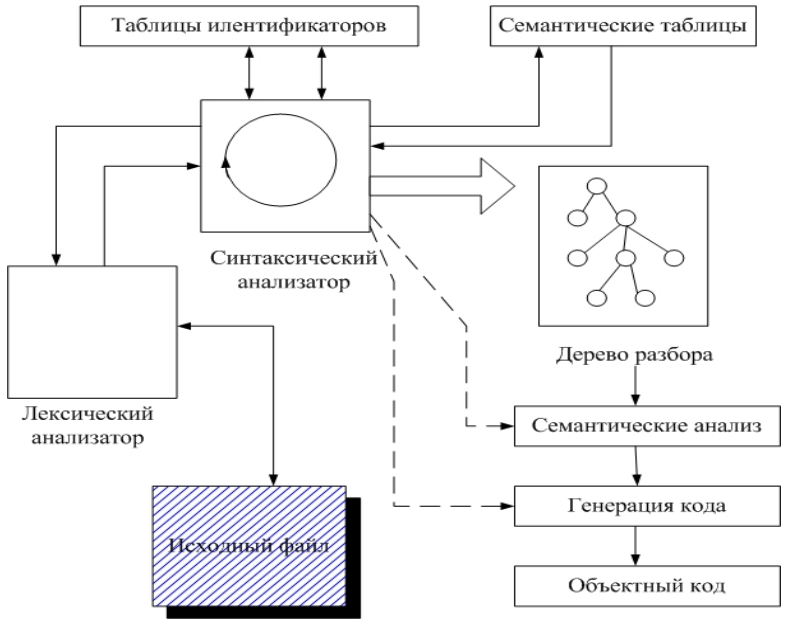


Рис. 4

2. ЛЕКСИЧЕСКИЙ АНАЛИЗ

2.1. Принципы построения лексического анализатора

Лексический анализатор (или сканер) – это часть компилятора, которая читает исходную программу и выделяет в ее тексте лексемы входного языка. На вход лексического анализатора поступает текст исходной программы, а выходная информация передается для дальнейшей обработки компилятором на этапе синтаксического анализа и является достаточно прямолинейным про-

цессом, который подобен автоматически производимому человеком при чтении. Благодаря сравнительно простой природе символов их всегда можно представить с помощью регулярных выражений или, эквивалентно, грамматик третьего типа. Помимо распознавания символов языка, лексический анализатор также выполняет некоторые другие задачи: удаление комментариев, введение нумерации строк, вычисление констант.

Важно понимать, что лексический анализатор всего лишь распознает символы языка для передачи их синтаксическому анализатору. Порядок их следования для него абсолютно не важен.

2.2. Лексический анализатор на базе конечного автомата

Формальной основой для построения лексических анализаторов является *конечный автомат* (КА) – формальный математический аппарат, широко используемым в технике. Так, достаточно сложная логика последовательности действий может быть реализована в виде программного КА.

Простейший лексический анализатор, распознающий идентификаторы и числовые константы, представляется в виде детерминированного конечного автомата со следующей диаграммой состояний и переходов (рис. 5).

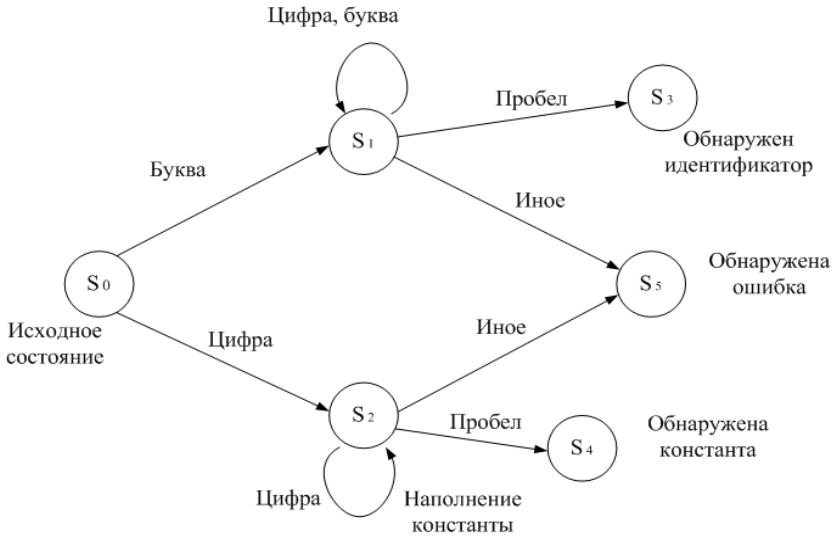


Рис. 5

Нужно отметить, что, с формальной точки зрения, КА характеризуется множеством состояний *S[n]* (одно из которых начальное), входным алфавитом (набором входных конечных сигналов) *I*, выходным алфавитом (множество выходных сигналов) *Q*, матрицей переходов M (рис. 6).

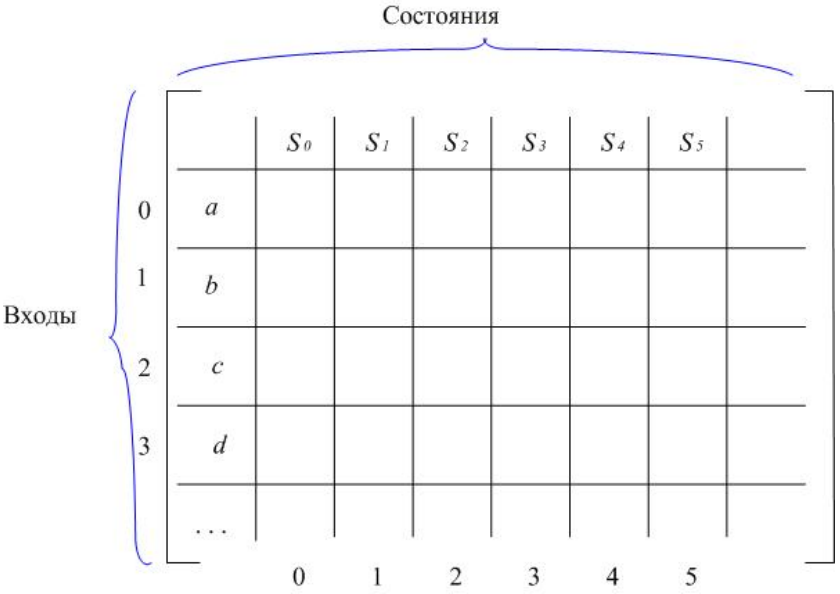


Рис.6

В качестве примера построим КА, распознающий следующие лексемы:

- идентификаторы (используются только буквы «*a –f*»)
- константы (десятичные, например «45», шестнадцатеричные – «0x*FF*», восьмеричные – «056»);
- знаки математических операций («+», «-», «*», «/»);
- скобки («(», «)»).

Для такого лексического анализатора КА будет представлен диаграммой состояний рис. 7.

Условия переходов в конечном состоянии S_{11} не показаны. Также не показаны условия переходов в конечное состояние с флагом «Ошибка».

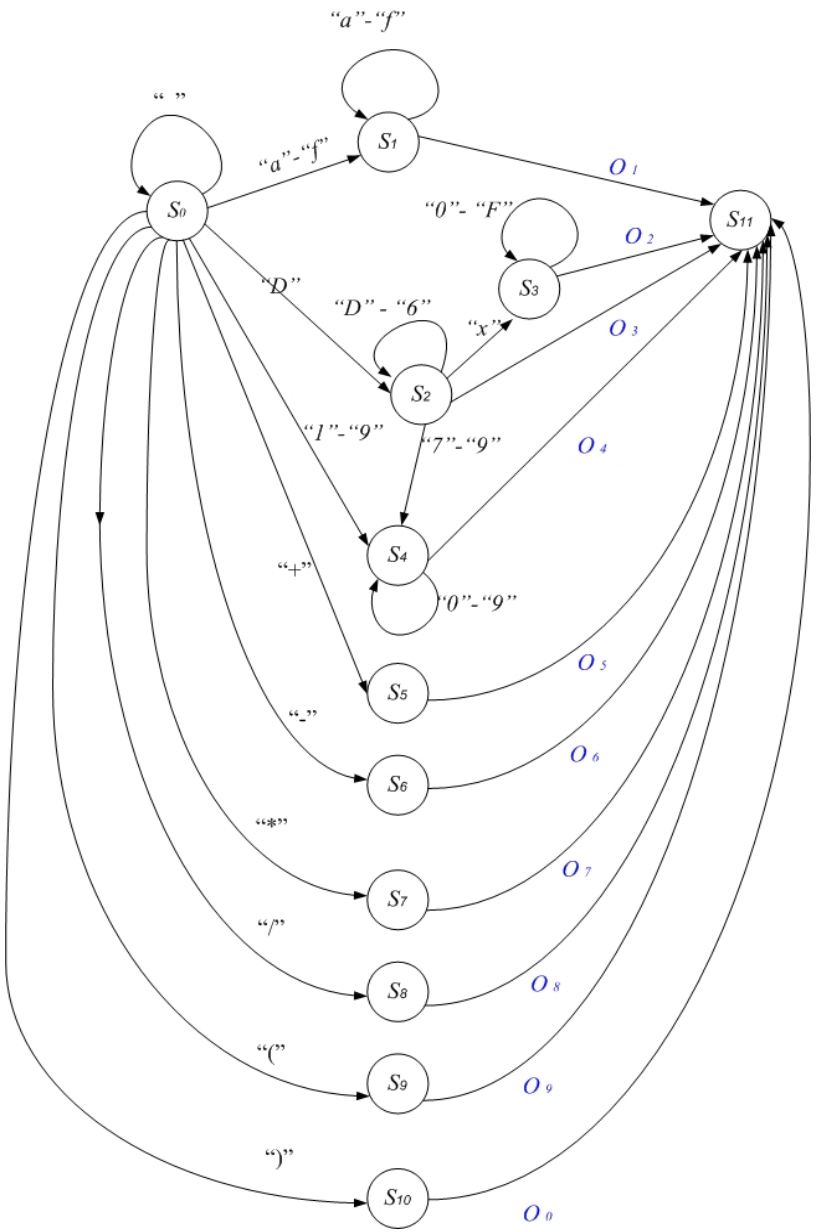

“a”-“f”
“ ”
“a”-“f”
“0”- “F”
“D”
“D” - “6”
“x”
“1”-“9”
“7”-“9”
“+”
“0”-“9”
“-”
“*”
“/”
“(”
“)”
S0
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
O 1
O 2
O 3
O 4
O 5
O 6
O 7
O 8
O 9
O 0

Рис. 7

Определим этот КА формально:

	a	b	c	d	e	f	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	x	+	-	*	/	(	)
S 0	S 1	S1	S1	S1	S1	S1	S2	S4	S4	S4	S4	S4	S4	S4	S4	S4	O10	O10	O10	O10	O10	O10	O10	S5	S6	S7	S8	S9	S10	S0
S 1	S 1	S1	S1	S1	S1	S1	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O10	O1	O1	O1	O1	O10	O1	O1
S 2	O10	O10	O10	O10	O10	O10	S2	S2	S2	S2	S2	S2	S2	S4	S4	S4	O4	O4	O4	O4	O4	O4	S3	O4	O4	O4	O4	O10	O4	O4
S 3	O10	O10	O10	O10	O10	O10	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	S3	O10	O2	O2	O2	O2	O10	O2	O2
S 4	-	-	-	-	-	-	S4	S4	S4	S4	S4	S4	S4	S4	S4	S4	-	-	-	-	-	-	-	O2	O2	O2	O2	-	O2	O2
S 5	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	O4	-	-	-	-	-	-	-	-	-	-	-	O4	-	O4
S 6	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	O5	-	-	-	-	-	-	-	-	-	-	-	O5	-	O5
S 7	O6	O6	O6	O6	-	-	-	-	-...																					
S 8	...																													
S 9	...																													
S 10																														
S 11	...																													

Рис.8

S={S0 , S1 ,..., S11 } S0 , S11 – соответственно начальное и конечное состояния;

I= {“a” – “f”, “0” – “F”,” ×”, “+”, “-“ , “*”, “/”, “(“, “)”, “ “ };

Q={ O1 –O2 };

O1 – обнаружен идентификатор;

O2 – обнаружена шестнадцатеричная константа;

O3 – обнаружена десятичная константа;

O4 – обнаружен восьмеричная константа;

O5 – обнаружен знак «+»;

O6 – обнаружен знак «-»;

O7 – обнаружен знак «*»;

O8 – обнаружен знак «/»;

O9 – обнаружен знак «(»;

O10 – обнаружен знак «)»;

O11 – обнаружена ошибка.

Матрица переходов M_{29x11} приведена на рис. 8.

У лексического анализатора реального компилятора размерность матрицы *M* будет больше, так как он воспринимает большее число входных символов и лексем и, как следствие, имеет большее число состояний.

2.3. Пример реализации лексического анализатора

В качестве примера реализации рассмотрим класс-утилиту, реализующий анализатор на основе конечного автомата. Построенный лексический анализатор будет предназначен для использования синтаксическим в качестве вспомогательного средства получения лексем. Реализованный класс, помимо служебных конструктора и деструктора, имеет всего одну функцию *GetToken ()*, получающую из входного файла очередную лексему:

```
#include<map>
#include<iostream>
#include<string>
#include<fstream>

using namespace std;
```

```
    typedef enum Statement {S0, S1, S2, S3, S4, S5,
S6, S7, S8, S9, S10, S11}; // Набор состояний
    typedef enum Output {O0, O1, O2, O3, O4, O5, O6,
O7, O8, O9, O10, O11}; // Набор выходных сигналов

    typedef enum TokenType {IDENT, HEX, DEC, OCT,
PLUS, MINUS, MUL, DIV, LBR, RBR}; //Типы лексем

    typedef pair<Statement, Output> Result; // Пара
(Новое Состояние & Выходной сигнал)
    typedef pair<Statement, char> Input;      // Со-
ставной ключ (Текущее Состояние & Входной символ)

    typedef pair<Input, Result> Item; //Элемент таб-
лицы переходов

    extern map<Input, Result> M; // Таблица перехо-
дов КА

    //Класс-исключение ошибки лексического анализа-
тора
    class LexErr {
    public:
      inline LexErr(string str)
      {
        cout << "Ошибка лексического анализатора: "
<< str << endl;
      }
    };


    //Функция инициализации должна заполнить таблицу
переходов конечного автомата
    void Init(void);

    //Класс "Лексема"

    class Token {
    public:
      string _value; // Строковое значение
      TokenType _type; //Тип лексемы
    };

    //Класс лексического анализатора
```

```
class Lex {
public:
  Lex(const char *fname); //Конструктор
  ~Lex();     //Деструктор
  bool GetToken(Token &token); //Функция получе-
ния очередной лексемы

private:
  ifstream _file; //Файловый поток для чтения
символов входной цепочки
};

#include "lex.h"
#include <map>
#include <string>
#include <iostream>
#include <fstream>

map<Input, Result> M; //Таблица переходов КА

//Функция-помощник, удаляет пробелы в конце и
начале строки.

string trim(const string &str) // Принимает на
вход строку
{
  string::const_iterator it1 = str.begin();
//Итератор it1 указывает на начало строки
  string::const_iterator it2 = str.end();
//Итератор it2 указывает на конец строки
  while(it1 != str.end() && *it1 == ' ') it1++;
//Пока пробел и не конец строки, сдвигаем итераторы
  while(it2 != str.begin() && *it2 == ' ') it2--;
  return string(it1,it2); // Создаем строку на
основе 2 итераторов, указывающих на начало и конец
строки без пробелов
}

void Init(void)
{
  // Заполнение таблицы
  M.insert(Item(Input(S0, 'a'), Result(S1, O1)));
  M.insert(Item(Input(S0, 'b'), Result(S1, O1)));
  M.insert(Item(Input(S0, 'c'), Result(S1, O1)));
```

```
M.insert(Item(Input(S0, 'd'), Result(S1, O1)));
M.insert(Item(Input(S0, 'e'), Result(S1, O1)));
M.insert(Item(Input(S0, 'f'), Result(S1, O1)));
M.insert(Item(Input(S0, '0'), Result(S2, O3)));
M.insert(Item(Input(S0, '1'), Result(S4, O3)));
M.insert(Item(Input(S0, '2'), Result(S4, O3)));
M.insert(Item(Input(S0, '3'), Result(S4, O3)));
M.insert(Item(Input(S0, '4'), Result(S4, O3)));
M.insert(Item(Input(S0, '5'), Result(S4, O3)));
M.insert(Item(Input(S0, '6'), Result(S4, O3)));
M.insert(Item(Input(S0, '7'), Result(S4, O3)));
M.insert(Item(Input(S0, '8'), Result(S4, O3)));
M.insert(Item(Input(S0, '9'), Result(S4, O3)));
M.insert(Item(Input(S0, 'A'), Result(S11, O11)));
M.insert(Item(Input(S0, 'B'), Result(S11, O11)));
M.insert(Item(Input(S0, 'C'), Result(S11, O11)));
M.insert(Item(Input(S0, 'D'), Result(S11, O11)));
M.insert(Item(Input(S0, 'E'), Result(S11, O11)));
M.insert(Item(Input(S0, 'F'), Result(S11, O11)));
M.insert(Item(Input(S0, 'x'), Result(S11, O11)));
M.insert(Item(Input(S0, '+'), Result(S5, O5)));
M.insert(Item(Input(S0, '-'), Result(S6, O6)));
M.insert(Item(Input(S0, '*'), Result(S7, O7)));
M.insert(Item(Input(S0, '/'), Result(S8, O8)));
M.insert(Item(Input(S0, '('), Result(S9, O9)));
M.insert(Item(Input(S0, ')'), Result(S10, O10)));
M.insert(Item(Input(S0, ' '), Result(S0, O0)));


M.insert(Item(Input(S1, 'a'), Result(S1, O1)));
M.insert(Item(Input(S1, 'b'), Result(S1, O1)));
M.insert(Item(Input(S1, 'c'), Result(S1, O1)));
M.insert(Item(Input(S1, 'd'), Result(S1, O1)));
M.insert(Item(Input(S1, 'e'), Result(S1, O1)));
M.insert(Item(Input(S1, 'f'), Result(S1, O1)));
M.insert(Item(Input(S1, '0'), Result(S11, O11)));
M.insert(Item(Input(S1, '1'), Result(S11, O11)));
M.insert(Item(Input(S1, '2'), Result(S11, O11)));
M.insert(Item(Input(S1, '3'), Result(S11, O11)));
M.insert(Item(Input(S1, '4'), Result(S11, O11)));
M.insert(Item(Input(S1, '5'), Result(S11, O11)));
M.insert(Item(Input(S1, '6'), Result(S11, O11)));
M.insert(Item(Input(S1, '7'), Result(S11, O11)));
M.insert(Item(Input(S1, '8'), Result(S11, O11)));
```

```
      M.insert(Item(Input(S1, '9'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'A'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'B'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'C'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'D'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'E'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'F'), Result(S11, O11)));
      M.insert(Item(Input(S1, 'x'), Result(S11, O11)));
      M.insert(Item(Input(S1, '+'), Result(S11, O1)));
      M.insert(Item(Input(S1, '-'), Result(S11, O1)));
      M.insert(Item(Input(S1, '*'), Result(S11, O1)));
      M.insert(Item(Input(S1, '/'), Result(S11, O1)));
      M.insert(Item(Input(S1, '('), Result(S11, O1)));
      M.insert(Item(Input(S1, ')'), Result(S11, O1)));
      M.insert(Item(Input(S1, ' '), Result(S11, O1)));

      /*
      ...
      */
    }

    Lex::Lex(const char *fname) //Конструктор прини-
мает на вход имя файла с входной цепочкой
    {
      _file.open(fname); // Пытаемся открыть файл в
поток
      if(!_file) throw LexErr("Ошибка открытия фай-
ла"); //Если неудача, возбуждаем исключительную си-
туацию

      _file.unsetf(ios::skipws);
    }

    Lex::~Lex()
    {
      _file.close();  //Деструктор закрывает файловый поток
    }

    bool Lex::GetToken(Token &token)
    {
      Statement current = S0; //Начальное состояние - S0
      TokenType type;
      string value;
      Result res;
```

```
      char c;
      map<Input, Result>::const_iterator it;

      while(!_file.eof()) //Пока не конец файла
      {
        _file.get(c); // Получаем очередной символ

        it = M.find(Input(current, c)); // Пытаемся
найти ячейку таблицы, соответствующую ключу Текущее
Состояние current, Входной Символ c

        if(it == M.end()) throw LexErr("Некорректный
символ " + c); // Если такой ячейки нет, возбуждаем
исключительную ситуацию
        res = it->second; // Получаем пару (Новое
Состояние & Выходной сигнал)
        if(res.second       ==       O11)       throw
LexErr("Лексическая ошибка"); // Если выходной сиг-
нал O11 - ошибка

        if(res.first == S11 || _file.peek() <= 0)
//Если следующее состояние финальное или "виден"
конец файла
        {
          if(res.first != S11) //Текущее состояние
финальное
          {
            value += c;
          }
          switch(res.second) //Выходной сигнал
          {
            case O1:
              type = IDENT;
              break;

            case O2:
              type = HEX;
              break;

            case O3:
              type = DEC;
              break;

            case O4:
```

```
                type = OCT;
                break;

            case O5:
                type = PLUS;
                break;

            case O6:
                type = MINUS;
                break;

            case O7:
                type = MUL;
                break;

            case O8:
                type = DIV;
                break;

            case O9:
                type = LBR;
                break;

            case O10:
                type = RBR;
                break;

            default:
                throw LexErr("Лексическая ошибка");
          }
          // Формируем лексему
          token._type = type;
          token._value = trim(value);
          _file.seekg(-1,  ios::cur);  //"Возвращаем
захваченный нами лишний символ"
          return true;
        }

        current = res.first; // Меняем текущее состояние
        value += c;     //Накапливаем строковое значение
      }
      return false;
    }
    #include "lex.h"
    #include <iostream>
```

```
using namespace std;

int main(void)
{
  Init();
  Token token;
  try
  {
    Lex lex("in.txt");
    cout << endl;
    while(lex.GetToken(token))
    {
      cout << "Лексемма: '" << token._value <<
"' " << "Тип " << token._type << endl;
    }
  }
  catch(LexErr err)
  {
    exit(1);
  }
  return 0;
}
```

Построение лексического анализатора на базе КА дает ряд преимуществ: достаточно компактная реализация, высокая скорость, но самое главное – расширяемость. Добавление новых или модификация существующих типов лексем затронет только несколько участков кода (определение состояний и выходов, формирование матрицы переходов и т.д.).

3. СИНТАКСИЧЕСКИЙ АНАЛИЗ

На этапе синтаксического анализа нужно установить, имеет ли цепочка лексем структуру, заданную синтаксисом языка, и зафиксировать ее. Структура таких конструкций, как выражение, описание, оператор и т.п., более сложна, чем структура идентификаторов и чисел. Поэтому для описания синтаксиса языков программирования нужны более мощные грамматики, чем регулярные. Обычно для этого используют укорачивающие контекстно-свободные грамматики (УКС-грамматики), правила которых имеют вид

$$A \to \alpha,$$

где $A \subset VN$, $\alpha \subset (VT \cup VN)^*$.

Грамматики этого класса, с одной стороны, позволяют достаточно полно описать синтаксическую структуру реальных языков программирования, с другой стороны, для разных подклассов УКС-грамматик существуют достаточно эффективные алгоритмы разбора.

Теоретически существует алгоритм, который по любой данной КС-грамматике и данной цепочке выясняет, принадлежит ли цепочка языку, порождаемому этой грамматикой. Но время работы такого алгоритма (синтаксического анализа с возвратами) экспоненциально зависит от длины цепочки, что с практической точки зрения совершенно неприемлемо.

Существуют табличные методы анализа, применимые ко всему классу КС-грамматик и требующие для разбора цепочек длины n времени $cn3$ (алгоритм Кока-Янгера-Касами) либо $cn2$ (алгоритм Эрли). Их разумно применять только в том случае, если для интересующего нас языка не существует грамматики, по которой можно построить анализатор с линейной временной зависимостью.

Алгоритмы анализа, расходующие на обработку входной цепочки линейное время, применимы только к некоторым подклассам КС-грамматик. Рассмотрим один из таких методов.

3.1. Метод рекурсивного спуска

Пусть дана грамматика G =({a,b,c, ⊥}, {S,A,B}, P, S),

где P: $S \to AB\perp$

$A \to a \mid cA$

$B \to bA$,

и нужно определить, принадлежит ли цепочка caba языку L(G). Построим вывод этой цепочки:

$$S \to AB\perp \to cAB\perp \to caB\perp \to cabA\perp \to caba\perp$$

Следовательно, цепочка принадлежит языку L(G). Последовательность применений правил вывода эквивалентна построению дерева разбора методом "сверху вниз" (рис. 9).

Метод рекурсивного спуска (РС-метод) реализует этот способ практически "в лоб": для каждого нетерминала грамматики создается своя процедура, носящая его имя. Ее задача – начиная с указанного места исходной цепочки, найти подцепочку, которая выводится из этого нетерминала.

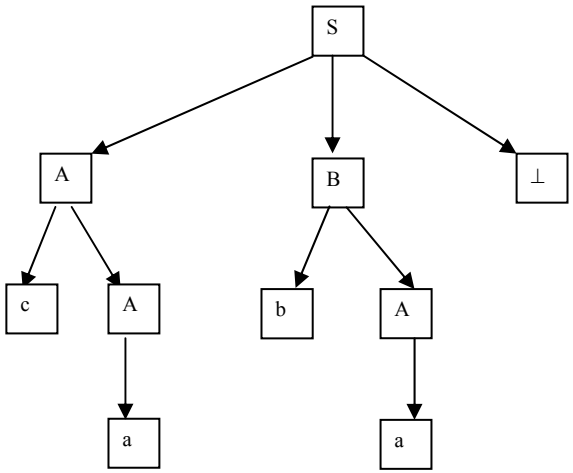


Рис. 9

Если такую подцепочку считать не удается, то процедура завершает свою работу вызовом процедуры обработки ошибки, которая выдает сообщение о том, что цепочка не принадлежит языку, и останавливает разбор. Если подцепочку удалось найти, то работа процедуры считается нормально завершенной и осуществляется возврат в точку вызова. Тело каждой такой процедуры пишется непосредственно по правилам вывода соответствующего нетерминала: для правой части каждого правила осуществляется поиск подцепочки, выводимой из этой правой части. При этом терминалы распознаются самой процедурой, а нетерминалы соответствуют вызовам процедур, носящих их имена.

3.2. Применимость метода рекурсивного спуска

Метод рекурсивного спуска применим в том случае, если каждое правило грамматики имеет вид:

либо $A \rightarrow \alpha$, где $\alpha \subset (VT \cup VN)^*$ и это единственное правило вывода для этого нетерминала;

либо $A \rightarrow a_1\alpha_1 \mid a_2\alpha_2 \mid ... \mid an\alpha n$, где $a_i \subset VT$ для всех $i = 1,2,...,n$; $ai \neq aj$ для $i \neq j$; $\alpha_i \subset (VT \cup VN)^*$, т. е. если для нетерминала A правил вывода несколько, то они должны начинаться с терминалов, причем все эти терминалы должны быть различными.

Ясно, что если правила вывода имеют такой вид, то рекурсивный спуск может быть реализован по вышеизложенной схеме.

Возникает естественный вопрос: если грамматика не удовлетворяет этим условиям, то существует ли эквивалентная КС-грамматика, для которой метод рекурсивного спуска применим? К сожалению, алгоритма, отвечающего на поставленный вопрос, не существует,т.е. это алгоритмически неразрешимая проблема.

Изложенные выше ограничения являются достаточными, но не необходимыми. Попытаемся ослабить требования на вид правил грамматики.

1. При описании синтаксиса языков программирования часто встречаются правила, описывающие последовательность однотипных конструкций, отделенных друг от друга каким-либо знаком-разделителем (например, список идентификаторов при описании переменных, список параметров при вызове процедур и функций и т.п.). Общий вид этих правил:

$$L \rightarrow a \mid a, L \text{ (либо в сокращенной форме } L \rightarrow a \{,a\})$$

Формально здесь не выполняются условия применимости метода рекурсивного спуска, так как две альтернативы начинаются одинаковыми терминальными символами.

Действительно, в цепочке *a*, *a*, *a, a, a* из нетерминала *L* может выводиться и подцепочка *a*, и подцепочка *a*, *a*, и вся цепочка *a*, *a*, *a*, *a*, *a*. Неясно, какую из них выбрать в качестве подцепочки, выводимой из *L*. Если принять решение, что в таких случаях будем выбирать самую длинную подцепочку (что и требуется при разборе реальных языков), то разбор становится детерминированным.

Важно, чтобы подцепочки, следующие за цепочкой символов, выводимых из *L*, не начинались с разделителя (в нашем примере – с запятой), иначе процедура *L* попытается считать часть исходной цепочки, которая не выводится из *L*. Например, она может порождаться нетерминалом *B* – ”соседом” *L* в сентенциальной форме, как в грамматике:

$S \rightarrow LB\perp$
$L \rightarrow a \{, a\}$
$B \rightarrow ,b$

Если для этой грамматики написать анализатор, действующий РС-методом, то цепочка *a*, *a*, *a*, *b* будет признана им оши-

бочной, хотя в действительности это не так. Нужно отметить, что в языках программирования ограничителем подобных серий всегда является символ, отличный от разделителя, поэтому подобных проблем не возникает.

2. Если грамматика не удовлетворяет требованиям применимости метода рекурсивного спуска, то можно попытаться преобразовать ее, т.е. получить эквивалентную грамматику, пригодную для анализа этим методом.

а) Если в грамматике есть нетерминалы, правила вывода которых леворекурсивны, т.е. имеют вид

$$A \rightarrow A\alpha 1 \mid ... \mid A\alpha_n \mid \beta_1 \mid ... \mid \beta_m,$$

где $\alpha_i \subset (VT \cup VN)^+$, $\beta_j \subset (VT \cup VN)^*$, $i = 1,2,...,n$; $j = 1,2,...,m$, то непосредственно применять РС-метод нельзя.

Левую рекурсию всегда можно заменить правой:

$$A \rightarrow \beta_1 A' \mid ... \mid \beta_m A'$$
$$A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \varepsilon .$$

Будет получена грамматика, эквивалентная данной, так как из нетерминала A по-прежнему выводятся цепочки вида $\beta_j \{\alpha_i\}$, где $i = 1,2,...,n$; $j = 1,2,...,m$.

б) Если в грамматике есть нетерминал, у которого несколько правил вывода начинаются одинаковыми терминальными символами, т.е. имеют вид

$$A \rightarrow a\alpha_1 \mid a\alpha_2 \mid ... \mid a\alpha_n \mid \beta_1 \mid ... \mid \beta_m,$$

где $a \subset VT$; $\alpha_i, \beta_j \subset (VT \cup VN)^*$, то непосредственно применять РС-метод нельзя. Можно преобразовать правила вывода данного нетерминала, объединив их с общими началами в одно правило:

$$A \rightarrow aA' \mid \beta 1 \mid ... \mid \beta_m$$
$$A' \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n .$$

Будет получена грамматика, эквивалентная данной.

с) Если в грамматике есть нетерминал, у которого несколько правил вывода, и среди них есть начинающиеся нетерминальными символами, т.е. имеют вид

$$A \rightarrow B_1\alpha_1 \mid ... \mid B_n\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m$$
$$B_1 \rightarrow \gamma_{11} \mid ... \mid \gamma_{1k}$$
$$. . .$$
$$B_n \rightarrow \gamma_{n1} \mid ... \mid \gamma_{np},$$

где $B_i \subset VN$; $a_j \subset VT$; α_i, β_j, $\gamma_{ij} \subset (VT \cup VN)^*$, то можно заменить вхождения нетерминалов B_i их правилами вывода в надежде, что правило нетерминала A станет удовлетворять требованиям метода рекурсивного спуска:

$$A \rightarrow \gamma_{11}\alpha_1 \mid ... \mid \gamma_{1k}\alpha_1 \mid ... \mid \gamma_{n1}\alpha_n \mid ... \mid \gamma_{np}\alpha_n \mid a_1\beta_1 \mid ... \mid a_m\beta_m .$$

г) Если допустить в правилах вывода грамматики пустую альтернативу, т.е. правила вида

$$A \rightarrow a_1\alpha_1 \mid ... \mid a_n\alpha_n \mid \varepsilon ,$$

то метод рекурсивного спуска может оказаться неприменимым (несмотря на то, что в остальном достаточные условия применимости выполняются). Например, для грамматики

$$G = (\{a,b\}, \{S,A\}, P, S),$$

где P: $S \rightarrow bAa$
$A \rightarrow aA \mid \varepsilon,$

РС-анализатор, реализованный по обычной схеме, будет примерно таким:

```
void S()
{if (c == 'b') {c = getc(); A();
                if (c != 'a') ERROR();}
 else ERROR();
}
void A()
{
    if (c == 'a') {c = getc(); A();}
}
```

Тогда при анализе цепочки *baaa* функция A() будет вызвана три раза; она прочитает подцепочку *aaa*, хотя третий символ *a* – это часть подцепочки, выводимой из *S*. В результате окажется, что *baaa* не принадлежит языку, порождаемому грамматикой, хотя в действительности это не так.

Проблема заключается в том, что подцепочка, следующая за цепочкой, выводимой из *A*, начинается таким же символом, как и цепочка, выводимая из *A*.

Однако в грамматике G = ({a,b,c}, {S,A}, P, S),
где P: $S \rightarrow bAc$
$A \rightarrow aA \mid \varepsilon,$

применение метода рекурсивного спуска возможно. Выпишем условие, при котором ε-правило вывода делает неприменимым РС-метод. Определение: множество FIRST(A) – это множество терминальных символов, которыми начинаются цепочки, выводимые из *A* в грамматике

$G = (VT, VN, P, S)$, т.е. $FIRST(A) = \{ a \subset VT \mid A \rightarrow a\alpha, A \subset VN, \alpha \subset (VT \cup VN)^* \}$.

Определение: множество FOLLOW(A) – это множество терминальных символов, которые следуют за цепочками, выводимыми из *A* в грамматике

$G = (VT, VN, P, S)$, т.е. $FOLLOW(A) = \{ a \subset VT \mid S \rightarrow \alpha A \beta, \beta \rightarrow a\gamma, A \subset VN, \alpha, \beta, \gamma \subset (VT \cup VN)^* \}$.

Тогда, если $FIRST(A) \cap FOLLOW(A) \neq \varnothing$, то метод рекурсивного спуска неприменим к данной грамматике. Если

$$A \rightarrow \alpha_1 A \mid ... \mid \alpha_n A \mid \beta_1 \mid ... \mid \beta_m \mid \varepsilon$$
$$B \rightarrow \alpha A \beta$$

и $FIRST(A) \cap FOLLOW(A) \neq \varnothing$ (из-за вхождения *A* в правила вывода для *B*), то можно попытаться преобразовать такую грамматику:

$$B \rightarrow \alpha A'$$
$$A' \rightarrow \alpha_1 A' \mid ... \mid \alpha_n A' \mid \beta_1 \beta \mid ... \mid \beta_m \beta \mid \beta$$
$$A \rightarrow \alpha_1 A \mid ... \mid \alpha_n A \mid \beta_1 \mid ... \mid \beta_m \mid \varepsilon$$

Полученная грамматика будет эквивалентна исходной, так как из *B* по-прежнему выводятся цепочки вида $\alpha \{\alpha_i\} \beta_j \beta$ либо $\alpha \{\alpha_i\} \beta$.

Однако правило вывода для нетерминального символа *A*’ будет иметь альтернативы, начинающиеся одинаковыми терминальными символами, следовательно, потребуются дальнейшие преобразования и успех не гарантирован.

Метод рекурсивного спуска применим к достаточно узкому подклассу КС-грамматик. Известны более широкие подклассы, для которых существуют эффективные анализаторы, обладающие тем же свойством, что и анализатор, написанный методом рекурсивного спуска, – входная цепочка считывается один раз слева

направо и процесс разбора полностью детерминирован, в результате на обработку цепочки длины *n* расходуется время *cn*. К таким грамматикам относятся LL(k)- и LR(k)-грамматики, грамматики предшествования и некоторые другие.

3.3. Метод рекурсивного спуска для простейшего языка

В качестве примера рассмотрим простейший язык, позволяющий объявлять и инициализировать целочисленные переменные, а также вычислять простейшие арифметические выражения, содержащие знаки математических операций и скобки, и построим синтаксический анализатор для цепочек этого языка. В дальнейшем мы рассмотрим процедуры генерации кода для этого языка, а также оптимизации полученного кода. Таким образом, соединив все вместе, можно будет получить простейший компилятор.

Итак, базовая программа на нашем языке будет выглядеть примерно так:

```
var a=5, b=11;
var c=a+b, d;
d=c+15;
```

Опишем грамматику нашего языка:

```
P → D ⇒ S ⇒ ⊥
D → ”var” ⇒ DLIST ⇒ “;” [⇒ D]
DLIST → I ⇒ [“=”⇒ E] [⇒ “,” ⇒ DLIST]
S → I ⇒ “=”⇒ E ⇒ “;” [⇒ S]
E → T ⇒ {“+-*/”⇒ T}*
T → “(“ ⇒ T ⇒ “)”
T → I | N
I → (a | b | c | d | e | f)+
N → (0 | 1 | 2 | 3| 4 | 5 | 6 | 7 | 8 | 9)+
```

В данном случае есть несколько правил, не позволяющих использовать метод рекурсивного спуска «в лоб». Ниже будет описан некий модифицированный алгоритм рекурсивного спуска с возвратами, позволяющий, при неудаче применения одной из альтернатив, вернуться к исходному состоянию и продолжить применение других правил. Построим дерево разбора для описанного выше предложения языка (рис. 10).

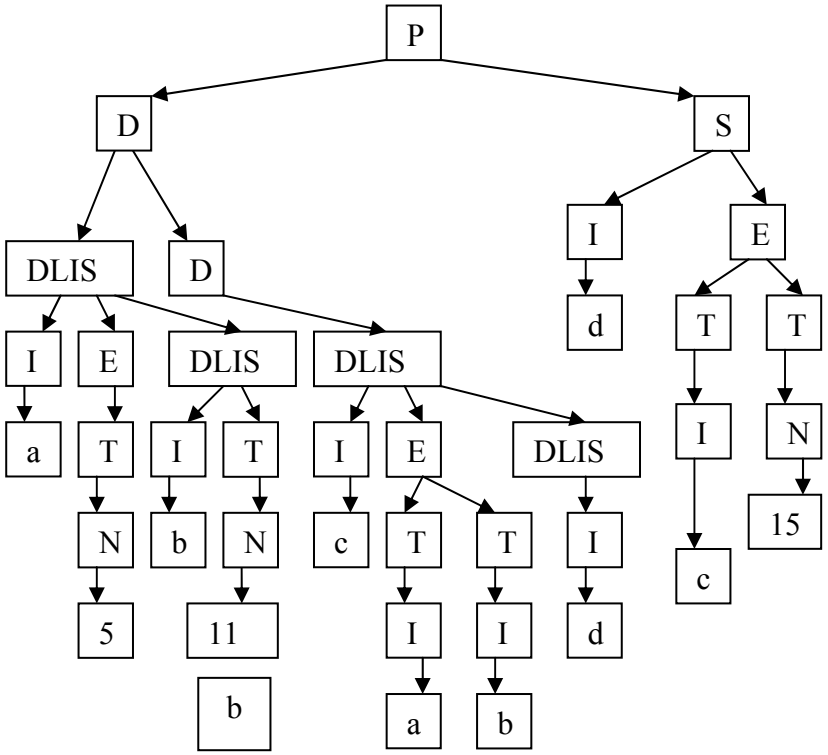


Рис. 10

3.4. Реализация синтаксического анализатора

За основу лексического анализатора, являющегося «поставщиком» лексем для синтаксического анализатора, возьмем класс, реализованный в предыдущем разделе. Видоизменим его так, чтобы он распознавал следующие классы лексем:

```
typedef enum TokenType { //Типы лексем
IDENT, // Идентификатор
NUM, //Число
PLUS, //Знак +
MINUS, // Знак -
MUL, // Знак *
DIV, // Знак /
```

```
    LBR, // Знак {
    RBR, // Знак }
    VAR, // Ключевое слово var
    EQUAL, //Знак =
    COMMA, // Знак ,
    DCOMMA // Знак ;
    };
```

Изменения затронут только реализацию, а интерфейс останется тем же. Кроме того, для реализации отката в синтаксическом анализаторе нам понадобится реализовать еще два метода:

```
    //Класс лексического анализатора
    class Lex {
    public:
      Lex(const char *fname); //Конструктор
      ~Lex();     //Деструктор
      bool GetToken(Token &token); //Функция получения очередной лексемы
      void SaveStatement(); // Сохранить состояние
      void  RollBack();    //Откатить  к  сохраненному состоянию

    private:
      ifstream  _file;  //Файловый  поток  для  чтения символов входной цепочки
      pos_type _position; // Указатель на сохраненный фрагмент потока
    };

    void Lex::SaveStatement()
    {
      _position = _file.tellg();
    }

    void Lex::RollBack()
    {
      _file.seekg(_position);
    }

    class Syntax {
    public:
      Syntax(const char *);
      bool Parse();
```

```
  private:
    Lex lex;

    bool On_P();
    bool On_D();
    bool On_DLIST();
    bool On_S();
    bool On_E();
    bool On_T();
    bool On_I();
    bool On_N();
  };

  Syntax::Syntax(const char *fname) : lex(fname)
  {
  }

  bool Syntax::Parse()
  {
    return On_P();
  }

  //Для правила P → D ⇒ S ⇒ ⊥

  bool Syntax::On_P()
  {
    Token t;
    if(!On_D()) return false;
    if(!On_S()) return false;
    if(lex.GetToken(t)) return false; // Если еще
что-то осталось, значит ошибка

    return true;
  }

  //Для правила D → ”var” ⇒ DLIST ⇒ “;” [⇒ D]

  bool Syntax::On_D()
  {
    Token t;
    if(!lex.GetToken(t)) return false;
    if(t._type != VAR) return false;
```

```
  if(!On_DLIST()) return false;
  if(!lex.GetToken(t)) return false;
  if(t._type != DCOMMA) return false;
  lex.SaveStatement();
  if(!On_D()) lex.RollBack();

  return true;
}

//Для правила DLIST → I ⇒ [“=”⇒ E] [⇒ “,” ⇒ DLIST]


bool Syntax::On_DLIST()
{
  Token t;
  if(!On_I()) return false;
  lex.SaveStatement();
  if(!lex.GetToken(t)) return false;
  if(t._type == EQUAL)
  {
    if(!On_E()) return false;
  }
  else
  {
    lex.RollBack();
  }
  lex.SaveStatement();
  if(!lex.GetToken(t)) return false;
  if(t._type == COMMA)
  {
    if(!On_DLIST()) return false;
  }
  else
  {
    lex.RollBack();
  }

  return true;
}

//Для правила S → I ⇒ “=”⇒ E ⇒ “;” [⇒ S]

bool Syntax::On_S()
```

```
    {
      Token t;
      if(!On_I()) return false;
      if(!lex.GetToken(t)) return false;
      if(t._type != EQUAL) return false;
      if(!On_E()) return false;
      if(!lex.GetToken(t)) return false;
      if(t._type != DCOMMA) return false;

      lex.SaveStatement();
      if(!On_S()) lex.RollBack();

      return true;
    }

    //Для правила E → T ⇒ {"+-*/"⇒ T}*


    bool Syntax::On_E()
    {
      Token t;
      if(!On_T()) return false;
      while(1)
      {
        lex.SaveStatement();
        if(!lex.GetToken(t)) return false;
        if(!(t._type == PLUS || t._type == MINUS ||
t._type == MUL || t._type == DIV))
        {
          lex.RollBack();
          return true;
        }
        if(!On_T()) return false;
      }
    }

    //Для правил T → "(" ⇒ T ⇒ ")",T → I | N


    bool Syntax::On_T()
    {
      Token t;
      lex.SaveStatement();
```

```
  if(!lex.GetToken(t)) return false;
  if(t._type == LBR)
  {
    if(!On_T()) return false;
    if(!lex.GetToken(t)) return false;
    if(t._type != RBR) return false;
  }
  else
  {
    lex.RollBack();
    lex.SaveStatement();

    if(On_I()) return true;
    else
    {
      lex.RollBack();
      if(!On_N()) return false;
    }
  }
  return true;
}

//Для правила I → (a | b | c | d | e | f)+


bool Syntax::On_I()
{
  Token t;
  if(!lex.GetToken(t)) return false;
  if(t._type != IDENT) return false;

  return true;
}

//Для правила N → (0 | 1 | 2 | 3| 4 | 5 | 6 | 7 | 8 | 9)+

bool Syntax::On_N()
{
  Token t;
  if(!lex.GetToken(t)) return false;
  if(t._type != NUM) return false;

  return true;
}
```

Полученный синтаксический анализатор всего лишь отвечает на вопрос, принадлежит ли входная цепочка символов исходному языку, т.е. является ли правильным предложением. Для компилятора или даже транслятора этой функции мало. Синтаксический анализатор должен переводить программу в некоторое внутреннее представление и обеспечивать вызов семантических процедур и процедур генерации кода.

4. СЕМАНТИЧЕСКИЙ АНАЛИЗ И ГЕНЕРАЦИЯ КОДА

Как уже было сказано, целью семантического анализа является проверка текста входной программы на наличие семантических ошибок. Для нашего языка семантические правила будут выглядеть примерно так:

1) перед использованием переменная должна быть объявлена;

2) если переменная используется по значению, то она должна быть проинициализирована;

3) допускаются неинициализированные переменные только как левая часть оператора присваивания.

Как правило, в современных компиляторах нет отдельной фазы семантического анализа. Так называемые семантические процедуры (процедуры проверки семантических ошибок) встраиваются в другие фазы компиляции, например в фазы генерации кода и синтаксического анализа. Так как в качестве основного принципа конструирования простейшего компилятора мы выбрали синтаксически управляемую трансляцию, то основной фазой будет являться именно синтаксический анализ. Следовательно, именно его мы будем дополнять семантическими процедурами, а также процедурами генерации кода, формирующими текст выходной программы на языке ассемблера. В данном пособии мы не будем рассматривать вопросы создания непосредственно машинного кода по нескольким причинам:

1) машинный код слишком зависим от архитектуры целевой вычислительной системы;

2) машинный код сложен для восприятия и понимания, так как требует специальных знаний по аппаратной реализации целевой вычислительной системы;

3) задача перевода ассемблерного кода в машинный достаточно тривиальна и не относится к задачам компиляции программ на языках высокого уровня. Во многих компиляторах задачу перевода ассемблерного текста в машинный код решают сторонние ассемблеры.

Итак, нам предстоит модернизировать синтаксический анализатор так, чтобы он не просто отвечал на вопрос, является ли входная программа правильной с точки зрения синтаксиса, но пытался перевести ее эквивалентную программу на выходном языке, в качестве которого мы будем рассматривать язык ассемблера.

Для начала нам придется рассмотреть некоторые теоретические аспекты перевода текста программы во внутреннее представление и способа вычисления арифметических выражений. Вместе с инструкциями они составляют основу любого языка программирования, в том числе и нашего.

4.1. Генерация внутреннего представления программ

Результатом работы синтаксического анализатора должно быть некоторое внутреннее представление исходной цепочки лексем, которое отражает ее синтаксическую структуру.

Основные свойства языка внутреннего представления программ:

1) он позволяет фиксировать синтаксическую структуру исходной программы;

2) текст на нем можно автоматически генерировать во время синтаксического анализа;

3) его конструкции должны относительно просто транслироваться в объектный код либо достаточно эффективно интерпретироваться.

Существует несколько общепринятых способов внутреннего представления программ:

- постфиксная запись,
- префиксная запись,
- многоадресный код с явно именуемыми результатами,
- многоадресный код с неявно именуемыми результатами,
- связные списочные структуры, представляющие синтаксическое дерево.

В основе каждого из этих способов лежит некоторый метод представления синтаксического дерева. Чаще всего синтаксическим деревом называют дерево вывода исходной цепочки, в котором удалены вершины, соответствующие цепным правилам вида $A \rightarrow B$, где $A, B \subset VN$. Выберем в качестве языка для представления промежуточной программы постфиксную запись (ее часто называют ПОЛИЗ – польская инверсная запись). В ПОЛИЗе операнды выписаны слева направо в порядке их использования. Знаки операций стоят таким образом, что знаку операции непосредственно предшествуют ее операнды. Например, обычной (инфиксной) записи выражения

a*(b+c)-(d-e)/f

соответствует постфиксная запись:

abc+*de-f/-.

Более формально постфиксную запись выражений можно определить таким образом:

- если E является единственным операндом, то ПОЛИЗ выражения E – это этот операнд;
- ПОЛИЗом выражения E1 θ E2, где θ – знак бинарной операции, E1 и E2 операнды для θ, является запись E1’ E2’ θ , где E1’ и E2’ – ПОЛИЗ выражений E1 и E2 соответственно;
- ПОЛИЗом выражения θ E, где θ – знак унарной операции, а E – операнд θ, является запись E’ θ, где E’ – ПОЛИЗ выражения E;
- ПОЛИЗом выражения (E) является ПОЛИЗ выражения E.

Запись выражения в такой форме очень удобна для последующей интерпретации (т.е. вычисления значения этого выражения) с помощью стека: выражение просматривается один раз слева направо, при этом:

- если очередной элемент ПОЛИЗа – операнд, то его значение заносится в стек;
- если очередной элемент ПОЛИЗа – операция, то на "верхушке" стека сейчас находятся ее операнды (это следует из определения ПОЛИЗа и предшествующих действий алгоритма); они извлекаются из стека, над ними выполняется операция, результат снова заносится в стек;
- когда выражение, записанное в ПОЛИЗе, прочитано, в стеке останется один элемент – значение всего выражения.

Может оказаться так, что знак бинарной операции по написанию совпадает со знаком унарной; например, знак "-" в большинстве языков программирования означает и бинарную операцию вычитания, и унарную операцию изменения знака. В этом случае во время интерпретации операции "-" возникнет неоднозначность: сколько операндов надо извлекать из стека и какую операцию выполнять. Устранить неоднозначность можно, по крайней мере, двумя способами:

1) заменить унарную операцию бинарной, т.е. считать, что "-a" означает "0-a";

2) ввести специальный знак для обозначения унарной операции; например, "-a" заменить на "&a". Важно отметить, что это изменение касается только внутреннего представления программы и не требует изменения входного языка.

Теперь необходимо разработать ПОЛИЗ для операторов входного языка. Каждый оператор языка программирования может быть представлен как n-местная операция с семантикой, соответствующей семантике этого оператора. Оператор присваивания

I := E

в ПОЛИЗе будет записан как

I E :=

где ":=" – это двухместная операция, а I и E – ее операнды; I означает, что операндом операции ":=" является адрес переменной I, а не ее значение.

Оператор перехода в терминах ПОЛИЗа означает, что процесс интерпретации надо продолжить с того элемента ПОЛИЗа, который указан как операнд операции перехода.

Чтобы можно было ссылаться на элементы ПОЛИЗа, будем считать, что все они перенумерованы, начиная с 1 (допустим, занесены в последовательные элементы одномерного массива).

Пусть ПОЛИЗ оператора, помеченного меткой L, начинается с номера *p*, тогда оператор перехода goto L в ПОЛИЗе можно записать как

p !

где ! – операция выбора элемента ПОЛИЗа, номер которого равен *p*.

Немного сложнее окажется запись в ПОЛИЗе условных операторов и операторов цикла. Введем вспомогательную операцию – условный переход "по лжи" с семантикой

if (not B) then goto L

Это двухместная операция в операндами B и L. Обозначим ее !F, тогда в ПОЛИЗе она будет записана как

B p F!

где *p* – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой L. Семантика условного оператора

if B then S1 else S2

с использованием введенной операции может быть описана как

if (not B) then goto L2; S1; goto L3; L2: S2; L3: ...

Тогда ПОЛИЗ условного оператора будет таким:

B p2 !F S1 p3 ! S2 ... ,

где *pi* - номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой Li, $i = 2,3$.

Семантика оператора цикла while B do S может быть описана так:

L0: if (not B) then goto L1; S; goto L0; L1: ... .

Тогда ПОЛИЗ оператора цикла while

B p1 !F S p0 ! ... ,

где *pi* – номер элемента, с которого начинается ПОЛИЗ оператора, помеченного меткой Li, $i = 0,1$.

Постфиксная польская запись операторов обладает всеми свойствами, характерными для постфиксной польской записи выражений, поэтому алгоритм интерпретации выражений пригоден для интерпретации всей программы, записанной на ПОЛИЗе (нужно только расширить набор операций; кроме того, выполнение некоторых из них не будет давать результата, записываемого в стек).

Постфиксная польская запись может использоваться не только для интерпретации промежуточной программы, но и для генерации по ней объектной программы. Для этого в алгоритме интерпретации вместо выполнения операции нужно генерировать соответствующие команды объектной программы.

4.2. Пример таблицы идентификаторов

Для хранения имен и значений идентификаторов удобно использовать глобальную таблицу идентификаторов программы. В нашем языке нет функций, поэтому таблица будет содержать

только имена и текущие значения переменных, а также дополнительные сведения о них. Для начала определим элемент этой таблицы – структуру, аккумулирующую основные сведения о переменной:

```
    struct variable
    {
      variable()
      {
         _id = ++counter;      // Идентификатор переменной генерируется автоматически
      }

      variable(string name, int value, bool initialized)
      {
         _id = ++counter; // Идентификатор переменной генерируется автоматически
         _name = name;
         _value = value;
         _initialized = initialized;
      }

      variable(const variable &obj) // Конструктор копирования во избежание генерации нового id при копировании объекта
      {
         _id = obj._id;
         _name = obj._name;
         _value = obj._value;
         _initialized = obj._initialized;
      }

      int _id; // Уникальный идентификатор
      string _name;   // Имя переменной
      int _value;       //Значение, в нашем языке - это целое число
      bool  _initialized;   //Проинициализирована ли переменная

      private:
      static int counter; // Счетчик для идентификатора
    };
    int variable::counter = 1; // Статическую переменную необходимо инициализировать
```

Теперь составим таблицу идентификаторов:

```
class table
{
  public:

      void Push(variable& var) // Функция вставки
новой переменной
      {
         _items.push_back(var);
      }

      bool GetByID(int id, variable& var) // Полу-
чение значения переменной по идентификатору
      {
         for(int i = 0; i < _items.size(); i++)
        {
           if(_items[i]._id == id)
         {
             var = _items[i];
             return true;
         }
        }
         return false;
      }

      bool GetByName(const string &name, variable&
var) //Получение значения переменной по имени
      {
         for(int i = 0; i < _items.size(); i++)
        {
           if(_items[i]._name == name)
         {
             var = _items[i];
             return true;
         }
        }
         return false;
      }

   // Для перебора значений в принципе нам будет
достаточно следующего функционала

      typedef vector<variable>::iterator iterator;
      iterator begin()
      {
         return _items.begin();
      }
```

```
        iterator end()
        {
           return _items.end();
        }

    protected:

        vector<variable>  _items;  //Таблица  -  это
просто массив структур типа "переменная"
        /*
```

В данном случае можно было бы реализовать более оптимальный с точки зрения скорости поиска способ хранения элементов, например, в виде хеш-таблицы с ключом по полю id или по полю name, однако наш пример учебный и мы ограничимся простым массивом с поиском методом простого перебора.

```
      */

    };
```

4.3. Вычисление арифметических выражений

Наиболее удобной структурой данных для перевода программы во внутреннее представление (ПОЛИЗ) и генерации кода является дерево синтаксического разбора. Кроме того, написанный нами синтаксический анализатор состоит из рекурсивных процедур, очень удобных для построения дерева, также имеющего рекурсивную структуру. К сожалению, в STL отсутствует контейнерный класс для реализации дерева, поэтому нам придется реализовать его самим.

Для начала определим структуру узла дерева с учетом того, что это должен быть контейнер и дерево не обязательно бинарное:

```
    template<class T> // Шаблонный класс узел дерева
(рекурсивно определяемая структура данных)
    class node
    {
    public:

      typedef  node<T>*  node_ptr;    //Указатель  на
узел
      T _value; // Данные
```

```
    list<node_ptr> _next; //Список указателей на
другие узлы

    ~node() //Деструктор
    {
      list<T>::iterator it;
      for(it = _next.begin(); it != _next.end();
it++)
      {
        if(*it != NULL)
        {
          delete *it;
        }
      }
    }

    // Функция для получения постфиксной (обратной
польской) записи для узла дерева реализуется как
обход ЛПК
    void reverse(list<T> &res) //Для сохранения
результата между рекурсивными вызовами используется
передача списка по ссылке. На выходе список содер-
жит элементы в порядке, соответствующем обходу ЛПК
    {
      list<T>::iterator it;
      for(it = _next.begin(); it != _next.end();
it++)
      {
        if(*it != NULL)
        {
          it->reverse(res);
        }
      }
      res.push_back(_value);
    }
  };
```

Теперь опишем контейнерный класс самого дерева:

```
template<class T> // Шаблонный класс дерева
class tree
{
public:
  node<T>* root; // Указатель на корень
```

```
    ~tree() // Деструктор
    {
      delete root;
    }

    list<T> Reverse() // Функция ЛПК(Левое, Правое, Корень) для дерева.
  // Такой обход дает обратную польскую запись
    {
      list<T> res;
      root->reverse(res);
      return res;
    }
  };
```

Попробуем написать код, позволяющий вычислять арифметические выражения при помощи обратной польской записи и стека. Вычисление выражений на этапе компиляции в нашем языке будет необходимо только для инициализации переменных. В реальных языках программирования оно также может быть использовано для константных вычислений в операторах присваивания, а также в других случаях.

Использовать синтаксическое дерево разбора для построения польской записи не получится, так как оно предназначено для других целей, например для генерации кода.

Поэтому для каждого вычислимого (а именно таким должно быть выражение для инициализации переменной) компилятор должен будет построить особое арифметическое дерево. Исходными данными для этого будет уже разобранный лексическим анализатором список лексем. Алгоритм построения дерева:

1. Рекурсивная процедура получает на вход корень некоего дерева и список лексем.
2. Если список состоит из одного элемента, то этот элемент помещается в корень дерева. Дальнейший рекурсивный вызов не производится.
3. Если список содержит множество элементов, а первый и последний элементы – парные скобки, то скобки извлекаются из списка, процедура вызывается снова.
4. Если первый и последний элементы – не парные скобки, то в списке осуществляется поиск операции с наименьшим при-

оритетом. Приоритет операций в порядке возрастания: («+», «-», «*», «/»). Операции внутри скобок игнорируются.

5. Найденная операция с наименьшим приоритетом помещается в корень дерева. Создаются левое и правое поддеревья. Найденная операция извлекается из списка, и он разбивается на два. Процедура вызывается рекурсивно для левого и правого поддеревьев (для левой и правой частей списка соответственно).

Для реализации этого алгоритма нам понадобится несколько модифицировать описание функций On_E(), On_T(), On_I() и On_N(), добавив реализацию On_E(list<Token> &_list), On_T(list<Token> &_list), On_I(list<Token> &_list), On_N(list<Token> &_list). В этом случае при рекурсивном вызове не только производится синтаксический разбор, но и происходит формирование списка лексем, необходимого для построения дерева арифметического разбора. Передача списка между вызовами функций по ссылке позволяет не только сэкономить время и память (нет необходимости вызывать конструктор копирования), но и формировать список на этапе нескольких вызовов.

Модифицированные функции с параметрами выглядят так:

```
bool Syntax::On_E(list<Token> &_list)
{
  Token t;
  if(!On_T(_list)) return false;
  while(1)
  {
    lex.SaveStatement();
    if(!lex.GetToken(t)) return false;
    if(!(t._type == PLUS || t._type == MINUS ||
t._type == MUL || t._type == DIV))
    {
      lex.RollBack();
      return true;
    }

    _list.push_back(t);

    if(!On_T(_list)) return false;
  }
}
```

```
bool Syntax::On_T(list<Token> &_list)
{
  Token t;
  lex.SaveStatement();
  if(!lex.GetToken(t)) return false;
  if(t._type == LBR)
  {
    _list.push_back(t);
    if(!On_T(_list)) return false;
    if(!lex.GetToken(t)) return false;
    if(t._type != RBR) return false;
    _list.push_back(t);
  }
  else
  {
    lex.RollBack();
    lex.SaveStatement();

    if(On_I(_list)) return true;
    else
    {
      lex.RollBack();
      if(!On_N(_list)) return false;
    }
  }
  return true;
}


bool Syntax::On_I(list<Token> &_list)
{
  Token t;
  if(!lex.GetToken(t)) return false;
  if(t._type != IDENT) return false;

  _list.push_back(t);

  return true;
}


bool Syntax::On_N(list<Token> &_list)
{
  Token t;
```

```
    if(!lex.GetToken(t)) return false;
    if(t._type != NUM) return false;

    _list.push_back(t);

    return true;
  }
```

Теперь, когда эти функции формируют список лексем, составляющих арифметическое выражение, можно реализовать функцию, осуществляющую построение дерева.

Для этого необходимо модифицировать класс лексемы и ввести туда операцию сравнения:

```
  class Token {
  public:
    string _value; // Строковое значение
    TokenType _type; //Тип лексемы

    // Расчет приоритета операции
    unsigned int Priority() const
    {
      switch(_type)
      {
        case PLUS:
          return 0;
        case MINUS:
          return 1;
        case MUL:
          return 2;
        case DIV:
          return 3;
        default:
          return 100;
      }
    }

    // Оператор сравнения. Будет необходим нам для
поиска операции с наименьшим приоритетом

    int operator<(const Token &obj)
    {
      return (Priority() < obj.Priority());
```

```
    }

};
```

Функция для построения дерева арифметического разбора:

```
bool Syntax::ExprToTree(list<Token> &_list, node<Token> &_root)
{
  unsigned int size = _list.size(); //Здесь можно выиграть скорость, избежав многочисленного вызова функции
  if(size == 0) return false;  // Если пустой список, то ошибка
  if(size == 1) // Если в списке всего один элемент, то помещаем его в корень
  {
    _root._value = _list.back();
    return true; // Рекурсия останавливается
  }

  if(_list.back()._type == RBR && _list.front()._type == LBR) // Нужно снять скобки
  {
    // Снимаем скобки
    _list.pop_back();
    _list.pop_front();

    return ExprToTree(_list, _root);
  }
  // Нужно помнить, что операции внутри скобок нас не интересуют, поэтому искать минимальный элемент придется вручную

  list<Token>::iterator it = _list.begin(), min_it = _list.begin();
  for(; it != _list.end(); it ++)
  {
    if(it->_type == LBR)
    {
      // Ищем соответствующую закрывающую скобку
      it++;
      int brkt_count = 1;
```

```
          while(brkt_count != 0)
          {
            if(it->_type == LBR) brkt_count++;
            if(it->_type == RBR) brkt_count--;
            it++;
          }
        }
        if(*it < *min_it) min_it = it;
      }

      // Создаем два новых списка, разбивая исходный
на две части
      list<Token> _list1(_list.begin(), it--);
      list<Token> _list2(++++it, _list.end());

      _root._value = *it; // В корень помещаем най-
денную операцию
      node<Token> *ptr1 = new node<Token>, *ptr2 =
new node<Token>; // Два указателя на новые поддере-
вья
      _root._next.push_back(ptr1);
      _root._next.push_back(ptr2);

      // Вызываем рекурсию для левой и правой частей

      if(!ExprToTree(_list1, *ptr1)) return false;
      if(!ExprToTree(_list2, *ptr2)) return false;

      return true;

    }
```

Далее необходимо описать функцию для расчета арифметического выражения при помощи стека. Так как эта функция может выявлять семантические ошибки (использование неинициализированных переменных), то сначала нужно определить для них класс-исключение:

```
    //Класс-исключение для семантических ошибок
    class SemanticErr {
    public:
      inline SemanticErr(string str)
      {
```

```
      cout << "Семантическая ошибка: " << str << endl;
    }
  };
```

Функция расчета арифметического выражения:

```
  extern table _table; // Глобальная таблица идентификаторов

  unsigned int Syntax::CalculateExpr(list<Token> &_poliz)
  {
    stack<unsigned int> _stack;
    unsigned int r1,r2;
    variable _var;
    list<Token>::iterator it = _poliz.end();

    for(; it != _poliz.end(); it++)
    {
      if(it->_type == IDENT) // Идентификатор
      {
        // Сначала нужно осуществить поиск в глобальной таблице идентификаторов
        if(!_table.GetByName(it->_value, _var)) throw SemanticErr("Нет объявления переменной " + it->_value);
        if(!_var._initialized) throw SemanticErr("Нет инициализации переменной " + it->_value);
        //Если все успешно (переменная объявлена и инициализирована), то кладем ее значение в стек:
        _stack.push(_var._value);
        continue; // На следующий шаг
      }

      if(it->_type == NUM)
      {
        //Число всегда кладем в стек
        _stack.push(atoi(it->_value.c_str())); // Здесь необходимо преобразование в число
      }

      if(it->_type == PLUS || it->_type == MINUS || it->_type == MUL || it->_type == DIV) // Операция
```

```
        {
          // Все операции у нас двуместные, поэтому
набор действий  однотипный: вынимаем оба аргумента
из стека, производим операцию и кладем результат в
стек

         if(_stack.size() < 2) throw SemanticErr ("Недос-
таточное число аргументов у оператора " + it->_value);

          r2 = _stack.top();
          _stack.pop();
          r1 = _stack.top();
           _stack.pop();

          switch(it->_type)
          {
            case PLUS:
              r1 = r1 + r2;
              break;

            case MINUS:
              r1 = r1 - r2;
              break;

            case MUL:
              r1 = r1*r2;
              break;

            case DIV:
              if(r2 == 0) throw SemanticErr("Ошибка
деления на ноль");
              r1 = r1 / r2;
          }

          _stack.push(r1); // Результат записываем в
стек

        }
      }

     // Теперь, если все правильно, в стеке должно
остаться одно число - результат
```

```
    if(_stack.size()  !=  1)  throw  SemanticErr
("Ошибка вычисления арифметического выражения");
    return _stack.top();
  }
```

Теперь можно вносить изменения в функции синтаксического анализатора для параллельной генерации кода. Первое изменение коснется функции On_DLIST(), обрабатывающей объявления переменных. На нее будет возложена задача заполнить таблицу идентификаторов (переменных), а также вычислить арифметические операции при инициализации.

```
  // Для формирования таблицы идентификаторов нужно будет получать имена переменных, поэтому в случае успеха функция должна возвращать саму лексему

  bool Syntax::On_I(Token &_token)
  {
    Token t;
    if(!lex.GetToken(t)) return false;
    if(t._type != IDENT) return false;

    _token = t;

    return true;
  }

  bool Syntax::On_DLIST()
  {
    Token t;
    variable _var; // Переменная
    variable tmp;
    _var._initialized = false;
    _var._value = 0;

    if(!On_I(t)) return false;
    _var._name = t._value;

    //Здесь должна обрабатываться ошибка по повторному объявлению переменной

    if(_table.GetByName(t._value,    tmp))    throw
SemanticErr("Переменная  "  +  t._value  +  "  уже
объявлена");
```

```
      lex.SaveStatement();
      if(!lex.GetToken(t)) return false;
      if(t._type == EQUAL) // Переменная объявлена с
инициализацией
      {
        list<Token>  _list,  _poliz;  //  Список  для
строки инициализации и для ПОЛИЗА
        if(!On_E(_list))  return  false;  //  Получаем
список

        tree<Token> _tree;
        _tree.root = new node<Token>;
        if(!ExprToTree(_list,   _tree.root))   return
false; // Преобразуем выражение в дерево
        _poliz  =  _tree.Reverse();  // Получаем ПОЛИЗ
обходом дерева ЛПК
        unsigned int res = CalculateExpr(_ploiz);

        _var._initialized = true; // Теперь перемен-
ная инициализирована
        _var._value = res;    //И ей присвоено значение

      }
      else
      {
        lex.RollBack();
      }

      _table.Push(_var);

      lex.SaveStatement();
      if(!lex.GetToken(t)) return false;
      if(t._type == COMMA)
      {
        if(!On_DLIST()) return false;
      }
      else
      {
        lex.RollBack();
      }

      return true;
    }
```

4.4. Генерация кода

Для генерации кода нам понадобится отдельный объект, представляющий собой некий обособленный генератор кода, отвечающий за правильное формирование предложений на выходном языке (в нашем случае это будет язык ассемблера). Управление этим объектом будет осуществлять синтаксический анализатор. Определим класс для такого объекта:

```
class CodeGenerator {
public:

  void DeclareBlock(); // Формирование блока объявления переменных
  void AddLine(string &str); //Добавление строки в выходной текст

  void Out(const char *fname); // Вывод полученного результата в файл
  void Out(string &str); // Вывод в строку

protected:

  string _str; // Формируемый текст программы на выходном языке

};
```

Реализация методов выглядит так:

```
void CodeGenerator::DeclareBlock()
{
  //Эта функция должна на основании построенной таблицы идентификаторов сформировать блок объявления переменных на языке ассемблера

  table::iterator it = _table.begin();
  _str += ".DATA\n";

  // Пробегаем всю таблицу в цикле
  for(; it != _table.end(); it++)
  {
    // Для универсальности все переменные в результирующем коде будут начинаться с префикса var_ и занимать в памяти двойное машинное слово
```

```
        _str += "var_";
        _str += it->_name;
        _str += "\tdd\t";
        _str += it->_value;
        _str += "\n";
      }

      _str += "\n.CODE\n";

    }

    void CodeGenerator::AddLine(string &str)
    {
      _str += str;
      _str += "\n";
    }

    void CodeGenerator::Out(string &str)
    {
      str = ".MODEL SMALL\n";
      str += _str;
      str += "END";
    }

    void CodeGenerator::Out(const char *fname)
    {
      ofstream _file; //Файловый поток для вывода
символов выходной цепочки
      string out; // Стока
      _file.open(fname);
      if(!_file) throw CodeGenErr("Ошибка открытия
файла");
      Out(out); // Сначала формируем строку
      _file << out;   // Запись строки в файл
      _file.close();    //Закрываем поток
    }
```

В процессе генерации кода используется исключение:

```
    class CodeGenErr {
    public:
      inline CodeGenErr(string str)
```

```
      {
        cout << "Ошибка при генерации кода: " << str
<< endl;
      }
    };
```

В нашем простом языке, в принципе, все вычисления являются константными, поэтому они могут быть произведены на этапе компиляции, но тогда наш пример был бы слишком простым и вся генерация кода сводилась к вычислению арифметических выражений и объявлению переменных с инициализацией. Поэтому мы искусственно введем оператор присваивания как операцию, которая может быть произведена только на этапе выполнения. Так как оператор присваивания обрабатывается функцией On_S(), то именно ее код нам придется модифицировать:

```
    bool Syntax::On_S()
    {
      Token t;
      variable tmp;
      if(!On_I(t)) return false; // Здесь, кроме про-
стой проверки, нужно получить само значение лексемы

      //Далее нужно выполнить проверку на наличие
этой переменной в таблице идентификаторов, иначе -
сгенерировать исключение для семантической ошибки

      if(!_table.GetByName(t._value,   tmp))   throw
SemanticErr("В операторе присваивания используется
переменная, которая не объявлена");

      //Левая часть оператора присваивания запомнена в tmp

      if(!lex.GetToken(t)) return false;
      if(t._type != EQUAL) return false;

      // Правая часть должна быть вычислена при по-
мощи уже известного алгоритма

      list<Token> _list, _poliz; // Список для стро-
ки инициализации и для ПОЛИЗА
      if(!On_E(_list)) return false; // Получаем список
```

```
      tree<Token> _tree;
      _tree.root = new node<Token>;
      if(!ExprToTree(_list,   *_tree.root))   return
false; // Преобразуем выражение в дерево
      _poliz  =  _tree.Reverse();  //  Получаем  ПОЛИЗ
обходом дерева ЛПК
      unsigned int res = CalculateExpr(_poliz);

      // Строковый поток используется как средство
преобразования числа в строку, так как функция itoa
есть не во всех современных компиляторах
      stringstream strm;
      strm << res;

      // Левая и правая части получены, можно гене-
рировать соответствующий код

      gen.AddLine("MOV  var_" + tmp._name + ",  " +
strm.str());

      //Весьма тонкий момент: нужно не забыть испра-
вить значение в таблице идентификаторов

      table::iterator it = _table.begin();

      for(; it != _table.end(); it++)
      {
        if(it->_id == tmp._id)
        {
          it->_value = res;
          break;
        }
      }

      if(!lex.GetToken(t)) return false;
      if(t._type != DCOMMA) return false;

      lex.SaveStatement();
      if(!On_S()) lex.RollBack();

      return true;
    }
```

Изменения коснутся и кода функции On_P():

```
bool Syntax::On_P()
{
  Token t;
  if(!On_D()) return false;

  //Теперь таблица идентификаторов сформирована и можно выполнить генерацию кода, относящегося к объявлению переменных

  gen.DeclareBlock();

  if(!On_S()) return false;
  if(lex.GetToken(t)) return false; // Если еще что-то осталось - значит ошибка

  gen.Out("out.asm"); // Вывод полученного кода в файл.

  return true;
}
```

В результате построенный нами генератор кода для входной программы

```
var a=5, b=11;
var c=a+b, d;
d=c+15;
```

должен формировать выходную программу:

```
.MODEL SMALL
.DATA
var_a    dw    5
var_b    dw    11
var_c    dw    16
var_d    dw    0

.CODE
MOV var_d, 31
END
```

Нужно понимать, что рассмотренный нами пример исключительно учебный. Мы не преследовали цель создать компилятор для реального языка программирования. Однако построенный нами компилятор для простейшего языка служит хорошим примером реализации транслятора с произвольного языка, описанного грамматикой и неким набором семантических правил, в код на языке ассемблера. В нем присутствуют и фазы лексического и синтаксического анализа, и генерация кода, и вычисление константных выражений при помощи инверсной польской записи и стека. Рассмотренный пример может служить основой для построения более сложных компиляторов.

Библиографический список

1. *Ахо А.,* Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты. М.: Вильямс, 2001.
2. *Хантер Р.* Проектирование и конструирование компиляторов. М.: Финансы и статистика, 1984.
3. *Грис Д.* Конструирование компиляторов для вычислительных машин. М.: Мир, 1976.
4. *Страуструп Б.* Язык программирования C++. СПб.: Бином, 2001.

О Г Л А В Л Е Н И Е

Смирнова Наталья Николаевна, Тарасов Сергей Дмитриевич

Основы построения компиляторов

Редактор *Г.М. Звягина*
Корректор *Л.А. Петрова*
Подписано в печать 07.08.2007. Формат бумаги 60x84/16. Бумага документная.
Печать трафаретная. Усл. печ. л. 3,75. Тираж 150 экз. Заказ № 131
Балтийский государственный технический университет
Типография БГТУ
190005, С.-Петербург, 1-я Красноармейская ул., д.1