Государственное образовательное учреждение
высшего профессионального образования
«МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ЛЕСА»

А. В. Чернышов

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ ИЗ СОСТАВА ОС UNIX И ИХ ПРИМЕНЕНИЕ В ПОВСЕДНЕВНОЙ ПРАКТИКЕ

УЧЕБНОЕ ПОСОБИЕ

Для студентов направления 230100

Издание второе, исправленное

Москва
Издательство Московского государственного университета леса
2010

УДК 004.4.02'2
Ч-49

Разработано в соответствии с требованиями Государственного образовательного стандарта высшего профессионального образования по направлению 230100.

Одобрено и рекомендовано к изданию редакционно-издательским советом университета.

Чернышов А. В.

Ч-49 Инструментальные средства программирования из состава ОС UNIX и их применение в повседневной практике: Учебное пособие для студентов направления 230100: Издание второе, исправленное. — М.: ГОУ ВПО МГУЛ, 2010. — 192 с.: ил.

ISBN

Учебное пособие посвящено вопросам практического использования стандартных инструментальных средств ОС UNIX для разработки прикладных и системных программ. Для инженеров–программистов, аспирантов и студентов.

УДК 004.4.02'2

ISBN

Введение

Приступив к чтению курса «Системное программное обеспечение» для студентов направления «Информатика и вычислительная техника», автор с удивлением обнаружил, что доступная для большинства студентов литература по этому курсу крайне недостаточна по глубине рассматриваемых вопросов в практическом плане. Если построению ассемблеров, редакторов связей и загрузчиков в ней уделяется достаточно большое внимание, то, например, построение компиляторов с языков программирования высокого уровня рассматривается в основном с чисто теоретических позиций. Причём сложность и некоторая запутанность, а также неизбежная в условиях дефицита времени поверхностность изложения этого материала способны лишь навсегда отбить у студентов желание разбираться в этом вопросе. Такой подход, кроме всего прочего, не позволяет организовать сколько-нибудь серьёзные лабораторные работы по этому разделу.

Известно, что на специальности «Прикладная математика» читаются специальные курсы, посвящённые теории грамматик, а также методам выполнения лексического и синтаксического разбора исходных строк программ. Для закрепления знаний студенты даже выполняют лабораторные работы, в которых требуется построить компилятор подмножества какого-либо языка заданными методами разбора. Однако складывается впечатление, что, защитив лабораторные работы, бывшие студенты на практике прибегают к использованию полученных знаний крайне редко и неохотно. Слишком уж сложны в реализации и отладке эти алгоритмы.

Сложилась парадоксальная практика. При необходимости разработать программу, настраиваемую командами, записанными в файл, нотацию команд стараются упростить до предела. Часто это упрощение приводит к абсолютной нечитабельности, что в свою очередь способно порождать ошибки при эксплуатации программы. Процедура же ввода и разбора данных всё равно получается громоздкой и незащищённой от различных ошибок как программных, так и ошибок в исходных данных.

Между тем, ещё в середине 70-х годов XX века в рамках ОС UNIX были разработаны инструментальные средства программиста, позволяющие разрабатывать программы лексического и синтаксического разбора, имея только общие представления о теории грамматик. Это генератор программ лексического разбора LEX и генератор программ синтаксического разбора YACC. Последний имеет ещё одно название — компилятор компиляторов. Пользуясь этими средствами, можно без особого труда строить достаточно сложные входные языки управления работой и настройки программ. Их можно

с успехом применять и для построения трансляторов, в том числе и с непроцедурных языков программирования.

Возможности изучения и, главное, применения этих средств сдерживаются двумя обстоятельствами. Во-первых, эти средства зачастую не включаются фирмами-разработчиками в распространённые коммерческие системы программирования.

Во-вторых, крайне редко удаётся увидеть в литературе подробное описание приёмов и методов создания программ с помощью рассматриваемых инструментальных средств. Чаще встречается описание общих подходов, сопровождаемое либо примитивным, либо заумным примером. Ни то, ни другое не способствует прояснению вопроса о правильном использовании этих средств. Но даже такие источники информации в наших библиотеках — большая редкость.

В настоящем учебном пособии автор предпринял попытку собрать воедино разбросанную по разным книгам справочную и учебную информацию о практическом использовании YACC и LEX. При этом пособие не претендует на теоретическую или техническую полноту. Из курса теории грамматик даётся только самый необходимый минимум понятий. Не приводится также исчерпывающая документация на рассматриваемое программное обеспечение. Однако общий объём материала должен обеспечить студентам возможность выполнения не только практических и лабораторных работ, но и возможных курсовых проектов по дисциплине «Системное программное обеспечение». Автор надеется, что будущие инженеры при этом получат правильное представление о возможностях инструментальных средств программирования и будут в дальнейшем с успехом применять их для решения не только учебных задач.

Все приводимые в настоящем учебном пособии примеры программ были проверены на практике и должны работать. Не все примеры являются оптимальными с точки зрения быстродействия или занимаемой памяти. Автор стремился прежде всего к примерам программ, которые были бы с минимумом комментариев понятны студентам, не владеющим языком Си в совершенстве.

Всё же предполагается, что читатели уже имеют начальное представление о языке программирования Си. Поэтому собственно язык в настоящем учебном пособии не рассматривается. Знание же языка Си++ для изучения и практического использования материала книги не требуется.

Во избежание путаницы с первоисточниками, изложение теоретического материала ведётся применительно к работе на ОС UNIX. Это прежде всего касается имён файлов, генерируемых в результате

работы LEX и YACC. Однако с точки зрения автора это не должно вызвать серьёзных проблем при работе с описываемыми инструментальными средствами под управлением других ОС.

В настоящем учебном пособии принята следующая система обозначений. Прописными буквами обозначаются инструментальные средства в целом, например YACC. Специальным шрифтом приводятся названия конкретных программ и файлов, например: `yacc`, `syntax.c`. Обозначение файлов с соответствующими спецификациями даётся обычным шрифтом, например уасс-программа.

Поскольку наиболее доступной для практического использования студентами из совместимых с ОС UNIX в настоящее время является ОС Linux, автор все приводимые в учебном пособии программы проверял именно в среде этой ОС. Эта же система была избрана автором и для организации студенческих лабораторных работ.

Материал учебного пособия размещён по главам следующим образом. Глава 1 посвящена введению в теорию грамматик. В ней приводятся определения основных терминов, знание которых необходимо для понимания последующих глав.

Главы 2 и 3 посвящены инструментальным средствам программирования, соответственно YACC и LEX. Они призваны служить для читателей введением в работу с этими генераторами, а также играть роль справочных руководств по этим средствам. При первом чтении книги нет необходимости изучать их подробно. Лучше возвращаться к ним для разъяснения вопросов, возникающих при чтении последующих глав.

Последующие главы посвящены созданию простого и полезного вычислительного средства (калькулятора выражений) и постепенному наращиванию его возможностей. К концу учебного пособия рассматриваемый калькулятор превращается в интерпретатор языка для вычисления сложных выражений.

Главы 4 и 5 посвящены созданию простейшего калькулятора с помощью YACC и некоторым простым его доработкам, а также организации обработки ошибок.

В главе 6 рассматривается ещё одно инструментальное средство, предназначенное для автоматизации различных работ в системе, MAKE. В частности, рассматривается вопрос об автоматизации процедуры трансляции и сборки рабочей программы, состоящей из нескольких файлов. Глава не претендует на роль справочного руководства по MAKE. Некоторые дополнительные вопросы использования MAKE рассматриваются в последующих главах по мере возникновения необходимости.

В главе 7 к разработке калькулятора подключается генератор LEX. Генерируемый им лексический анализатор заменяет использовавшийся до этого, написанный вручную на языке Си.

Глава 8 посвящена введению в калькулятор возможности работать с именами переменных произвольной длины, а также предопределёнными константами и встроенными математическими функциями.

В главе 9 выполняется переход от непосредственного вычисления выражений в процессе разбора к трансляции на простую машину с последующим выполнением полученной машинной программы.

Глава 10 посвящена дополнению языка калькулятора управляющими структурами.

В главе 11 рассматриваются направления дальнейшего развития возможностей калькулятора и его способов реализации.

В приложениях читатели найдут дополнительную информацию, предназначенную для желающих опробовать описанные средства на практике.

Приложение 1 содержит полные тексты двух разработанных в учебном пособии программ: `calc4a` и `calc6`.

В приложении 2 приведён краткий справочник по языку калькулятора `calc6`.

В приложении 3 даются задачи на самостоятельную доработку калькулятора. Большинство задач было взято автором из [1], но есть и оригинальные. Задачи разбиты на группы, соответствующие главам основной части пособия, после прочтения которых имеет смысл приступать к их решению. Среди задач есть такие, решение которых тривиально и требует от программиста лишь усидчивости. Однако для многих из приведённых задач нет однозначных решений. Студенты должны самостоятельно выбрать и обосновать способы решения таких задач.

Приложение 4 даёт краткую историческую справку об изучаемых инструментальных средствах программиста.

В приложении 5 приводится сводная информация о командах трансляции и сборки программ в ОС UNIX и ОС Linux.

В приложении 6 даны адреса в сети Internet, по которым можно получить доступ к современным системам программирования для различных ОС, содержащих представленные в настоящем пособии инструментальные средства программиста.

Автор выражает благодарность своим студентам предыдущих выпусков, указавших ему на ошибки в первом издании настоящего учебного пособия.

Глава 1
Основы теории грамматик

1.1. Основные понятия

Введём несколько терминов, которыми будем в дальнейшем постоянно пользоваться.

Алфавит — это любое множество символов. Понятие символа не определяется. Заметим, однако, сразу, что все символы мы будем подразделять на *терминальные* и *нетерминальные*.

Цепочка символов — некоторая последовательность символов. Например, последовательность символов 1, 2, 3 образует цепочку 123.

Зададимся множеством символов E и обозначим через E^* множество всех возможных цепочек из элементов множества E.

Язык — это подмножество E^*.

Примеры языков: русский, язык арифметических выражений, язык программирования C. Язык можно задать различными способами: перечислив все возможные цепочки; написав программу-распознаватель, которая получает на вход цепочку символов и выдаёт ответ «да», если цепочка принадлежит языку, и «нет» в противном случае; с помощью механизма порождения — грамматики.

Задание грамматики определяется как задание четвёрки, состоящей из:

– множества терминальных символов;

– множества нетерминальных символов;

– множества правил вывода (правил подстановки);

– начального символа.

Терминальными символами или просто **терминалами** называются символы, составляющие словарь языка.

Нетерминальными символами (**нетерминалами**) называют символы грамматики, которые в соответствии с правилами подстановки могут быть заменены на некоторые цепочки символов.

Каждое **правило вывода** грамматики задаёт некоторую **подстановку**, которую можно применить для получения одной цепочки символов из другой. При этом левая часть правила определяет некоторый нетерминал*, а правая — саму эту подстановку.

Последовательность подстановок, с помощью которой из начального символа грамматики получается некоторая цепочка, называется **выводом** этой цепочки.

---

* Это справедливо не для всех классов грамматик.

Таким образом, **начальный символ** — это некоторый специальным образом зафиксированный нетерминал, из которого осуществляют вывод всех цепочек языка.

Дадим теперь определение языка с точки зрения задания его грамматикой.

Язык — это совокупность всех цепочек *терминальных* символов, которые можно вывести из *начального символа*, пользуясь правилами подстановки.

В дальнейшем в настоящей главе будем все терминальные символы обозначать строчными латинскими буквами, например: a, b. Все нетерминальные символы — прописными латинскими буквами или словами, записанными в угловые скобки, например: W, `<формула>`. Цепочки символов (включающие, возможно, и терминальные, и нетерминальные символы) будем обозначать греческими буквами, например χ.

Для записи правил вывода будем использовать форму, близкую к широко распространённой форме Бэкуса–Наура (БНФ).

Покажем на простом примере использование введённых терминов.

Возьмём сильно упрощённый язык арифметических выражений, имеющий словарь, состоящий из следующих символов:

$$\{a, b, (,), +, -, *, /\}.$$

Здесь a и b имеют смысл операндов выражения, а смысл остальных символов очевиден. Рассмотрим грамматику языка, заданную следующими правилами вывода:

$$\begin{aligned}
S &: \ W + S \\
S &: \ W - S \\
S &: \ W \\
W &: \ W * W \\
W &: \ W/W \\
W &: \ (S) \\
W &: \ a \\
W &: \ b
\end{aligned}$$

Символы, записанные выше как символы словаря — это терминальные символы. Символы S и W — нетерминальные символы. Нетерминал S зафиксирован как начальный символ грамматики. Пользуясь правилами вывода, можно, например, следующим способом осуществить вывод цепочки:

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

из начального символа

$$
\begin{aligned}
&S\\
&W - S\\
&(S) - S\\
&(W + S) - S\\
&(a + S) - S\\
&(a + W) - S\\
&(a + W * W) - S\\
&(a + b * W) - S\\
&(a + b * b) - S\\
&(a + b * b) - W\\
&(a + b * b) - W/W\\
&(a + b * b) - a/W\\
&(a + b * b) - a/b
\end{aligned}
$$

Заметим, что приведённые выше правила вывода в нашей грамматике можно было записать в более компактной форме, используя символ | как знак «или»:

$$
\begin{aligned}
&S:\ W + S \mid W - S \mid W\\
&W:\ W * W \mid W/W \mid (S) \mid a \mid b
\end{aligned}
$$

Обратите внимание, что грамматика не определяет смысла выражения (так называемой **семантики**). Хотя мы и условились, что «смысл символов очевиден», ничто не мешает рассматривать, например, символ $*$ не как знак численного умножения, а как знак поразрядного логического умножения (побитного И).

1.2. Классификация грамматик

Первая (созданная в 1950-х годах) классификация грамматик по сложности соответствующих программ-распознавателей называется иерархией Хомского. В ней выделены 4 класса грамматик (в порядке возрастания сложности):

а) регулярные (или автоматные). Правила имеют вид

$$A:\ \chi B \quad \text{или} \quad A:\ \chi,$$

где χ — цепочка терминалов или пустая цепочка;

б) контекстно свободные (или КС). Правила имеют вид

$$A: \psi,$$

где ψ — цепочка из терминалов и нетерминалов.

К этому классу относится рассмотренный в предыдущем параграфе язык арифметических выражений;

в) контекстно зависимые (неукорачивающие). Правила имеют вид:

$$\zeta: \xi,$$

где ζ и ξ — цепочки из терминалов и нетерминалов, ζ содержит нетерминал, длина ζ меньше или равна длине ξ;

г) без ограничений.

Следующие вложения почти очевидны, если не допускать наличия правил с пустой правой частью в КС-грамматиках:

$$\text{а} \in \text{б} \in \text{в} \ \in \text{г}.$$

Конечно, для языка, определённого регулярной грамматикой, всегда можно написать контекстно свободную грамматику (и даже грамматику без ограничений!). Тем не менее, все вложения строгие: в каждом классе существуют языки, которые нельзя задать грамматиками более простого класса.

1.3. Синтаксический анализ

Главной задачей синтаксического анализа является установление принадлежности символов входного текста языку, определяемому грамматикой. Для решения этой задачи синтаксическому анализатору необходимо восстановить вывод входной цепочки из начального символа или установить невозможность такого вывода. Методы анализа обычно делят на **нисходящие** и **восходящие**, отличающиеся друг от друга порядком, в котором распознаются правила в выводе.

Если для одной и той же исходной цепочки символов одним и тем же методом разбора можно найти несколько различных выводов (различных последовательностей замен) этой цепочки из начального символа, то такая грамматика называется **неоднозначной**. Иначе грамматика **однозначная**. Наиболее удобно представлять последовательность вывода в виде так называемого **дерева разбора**.

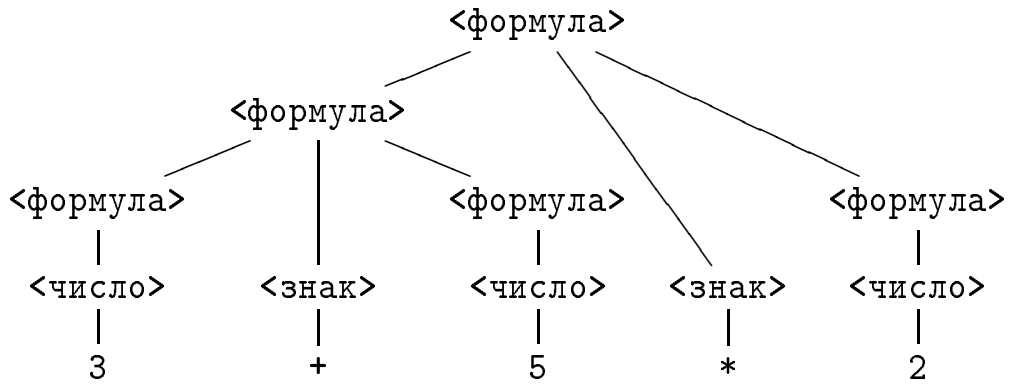

Рис. 1.1. Дерево разбора

Рассмотрим пример. Пусть задана грамматика:

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

Выражение $3+5*2$ есть формула, поскольку для него можно построить дерево разбора (рис. 1.1).

Однако это не единственно возможное дерево разбора. Второе дерево разбора, удовлетворяющее той же грамматике, приведено на рис. 1.2. Следовательно, рассмотренная грамматика неоднозначна.

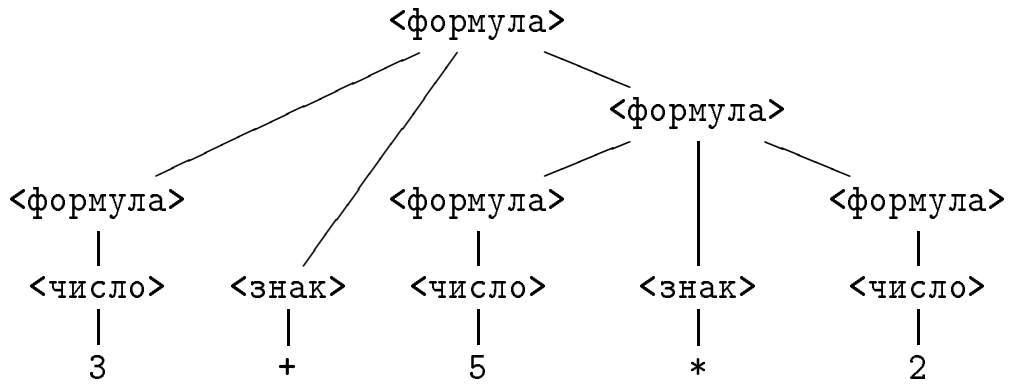

Рис. 1.2. Альтернативное дерево разбора

Однако тот же самый язык можно описать однозначной грамматикой, заменив первое правило на пару:

```
<формула>: <терм> | <терм><знак><формула>
<терм>: (<формула>) | <число>
```

Дерево разбора для этой грамматики и нашего выражения в качестве упражнения постройте самостоятельно.

В приведённых примерах никак, кроме скобок, нельзя задать приоритет операций. В первом случае порядок вычислений просто неоднозначен и, следовательно, результат непредсказуем. Во втором случае вычисление любого выражения без скобок выполняется

всегда справа налево, что для нашего примера лишь случайно совпадает с принятым порядком выполнения операций (сначала умножение, а потом сложение). Изменив грамматику, мы можем учесть приоритет операций:

```
<формула>: <терм> | <формула> + <терм>
<терм>: <элемент> | <терм> * <элемент>
<элемент>: (<формула>) | <число>
```

В приведённой грамматике цепочки в левой и правой частях правила начинаются с одного нетерминала. Такая грамматика называется **леворекурсивной**. При реализации такой грамматики программно «в лоб» возникает проблема бесконечной рекурсии. В данном случае проблема легко снимается устранением левой рекурсии:

```
<формула>: <терм> | <терм><пм><формула>
<терм>: <элемент> | <элемент><уд><терм>
<пм>: + | -
<уд>: * | /
```

Одновременно мы добавили в нашу грамматику операции вычитания и деления. Однако в праворекурсивной грамматике появляется новая проблема. Преобразовав грамматику, мы изменили порядок свёртки операций. Если традиционно операции одного приоритета выполняются слева направо (говорят, что операции **левоассоциативны**), то в нашей грамматике все операции определены как **правоассоциативные**. Например, дерево разбора для выражения $5 - 3 - 2$ показано на рис. 1.3.

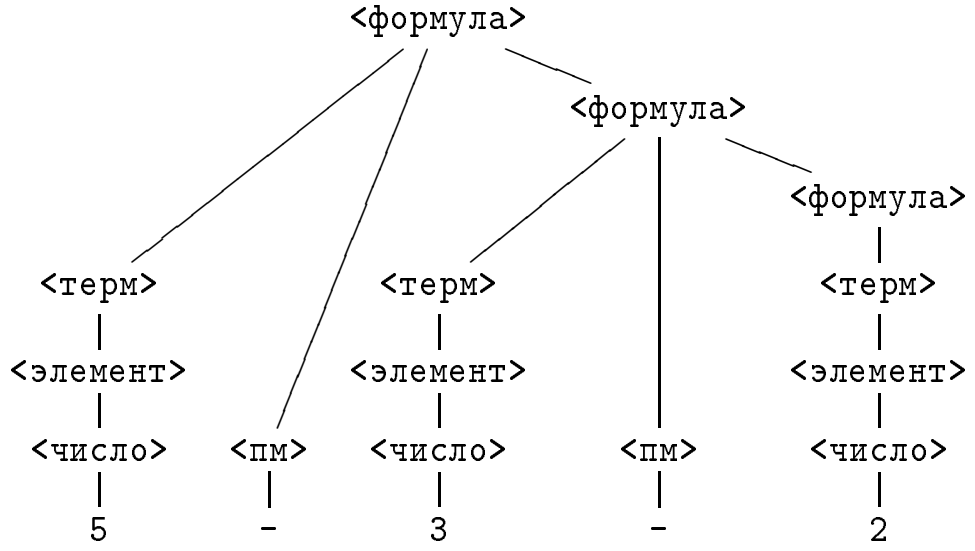

Рис. 1.3. Праворекурсивное дерево разбора

Строго говоря, в БНФ нет средств, позволяющих менять ассоциативность операций, не меняя рекурсивность грамматики. Однако в тех инструментальных средствах программиста, которые мы в дальнейшем рассмотрим, такие средства предусмотрены, поэтому на практике описанная проблема не возникает.

Кроме привычной формы записи арифметических формул, называемой **инфиксной**, когда знак операции записывается между операндами, в мире вычислительной техники широко распространена **постфиксная**, или **польская**, форма записи, в которой знак операции записывается после операндов. Например:

$2 + 3$	$2\ 3\ +$
$2 * 3 + 4$	$2\ 3\ *\ 4\ +$
$2 * (3 + 4)$	$3\ 4\ +\ 2\ *$

При такой форме записи скобки не требуются, а значение формулы очень легко вычислить с помощью стека чисел. Именно таким образом обычно происходит вычисление выражений в вычислительных машинах.

В сущности, инфиксная и постфиксная формы записи арифметических выражений представляют собой два языка, описывающих одну и ту же предметную область. Перевод с одного языка на другой называется **трансляцией**. В частности, перевод выражения с инфиксной в постфиксную форму записи — трансляция. Если сперва выполняется полный перевод выражения, а затем его вычисление на основании записи на новом языке, то такая трансляция называется **компиляцией**. Если же выражение вычисляется в процессе разбора без перевода, то это **интерпретация**.

1.4. Лексический анализ

Рассматривая в п. 1.3 синтаксический анализ, мы считали, что терминальные символы исходных цепочек нам известны как бы заранее. Мы пользовались этим допущением потому, что все рассмотренные нами примеры в качестве терминальных символов имели единичные символы ASCII.

На практике редко используют такой подход. Чаще терминалами являются такие объекты исходного текста как идентификаторы, целые и вещественные числа, ключевые слова и т. п. Распознавание этих объектов во входном потоке транслятора — задача лексического анализатора.

Лексический анализ включает в себя сканирование входного потока и распознавание в нём лексем, составляющих предложения исходного текста.

Лексема — это неделимый элемент распознаваемого языка, соответствующий понятию *терминальный символ* в грамматике. Если лексема представляет собой один символ ASCII, то её часто называют **литералом**. Например, в грамматиках, рассмотренных в п. 1.3, литералами являются символы + и −.

Такие объекты, как идентификаторы и целые числа, могут считаться лексемами, но могут также распознаваться и как грамматические конструкции, «собираемые» из литералов. Например:

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

В этом случае лексический анализатор должен распознавать только литералы. Далее в процессе синтаксического анализа будет установлено, что некоторая цепочка цифр образует объект `<число>`.

Однако такой подход значительно усложняет синтаксический анализатор. Кроме того, при таком подходе гораздо сложнее учитывать такие ограничения, как, например, максимальная длина идентификатора. Ещё одним отрицательным свойством такого подхода является сложность введения новых ключевых слов, поскольку это опять же требует значительной переработки синтаксического анализатора.

Выбор правильного уровня «разделения полномочий» между лексическим и синтаксическим анализаторами не всегда является простой задачей. Однако в случае разработки транслятора для большинства существующих языков программирования можно рекомендовать подход, при котором в качестве лексических единиц распознаются ключевые слова, идентификаторы, числовые константы, символьные строки; комментарии лексическим анализатором просто игнорируются, а остальные символы считаются литералами. Названные лексические единицы и литералы передаются затем на вход синтаксического анализатора как терминальные символы.

Распознанные лексемы для повышения эффективности дальнейшей обработки кодируются некоторыми уникальными кодами, например, целыми числами. Кодировать литералы нет необходимости — для числовой идентификации вполне достаточно использовать их ASCII код.

В случае распознавания ключевого слова кодирования целым числом вполне достаточно для дальнейшей обработки. В случае же идентификатора дополнительно требуется знать его имя, а в случае числа — его значение. Эта информация, наряду с кодом лексемы, также должна быть передана синтаксическому анализатору.

Можно представить себе транслятор, который сначала выполняет полный лексический разбор входного текста, а затем по распознанным лексемам проводит синтаксический анализ. Однако чаще

применяют более эффективный метод, при котором лексический анализатор вызывается по мере необходимости самим синтаксическим анализатором, когда ему требуется получить на вход очередную лексему. Распознав лексему во входном потоке и передав её синтаксическому анализатору, лексический анализатор завершает свою работу до следующего вызова. Такой подход позволяет отслеживать состояние синтаксического анализатора и при необходимости корректировать правила лексического разбора.

1.5. Применение лексических и синтаксических анализаторов

Модули лексического и синтаксического анализаторов могут применяться для создания самых разнообразных программ: потокового редактора текстов; языка пакетной обработки заданий; интерпретатора команд, задаваемых с консоли; систем вёрстки и многих других. Но самым важным их приложением является, безусловно, работа в трансляторах с языков высокого уровня.

Лексический и синтаксический анализаторы являются весьма сложными частями любого транслятора. К счастью, теория их построения достаточно хорошо разработана. Это как раз те части транслятора, построение которых хорошо поддаётся формализации, что позволяет автоматизировать этот процесс и переключить внимание программиста на действительно творческую работу по созданию семантической (смысловой) части транслятора.

Инструментальные средства программиста, позволяющие автоматизировать построение модулей лексического и синтаксического анализаторов, в мировой практике получили обобщающее название *компиляторов компиляторов**. Среди множества подобных разработок наибольшую известность и практическую ценность приобрели: LEX — генератор программ лексического разбора и YACC — генератор программ синтаксического разбора. Эти инструментальные средства созданы для ОС UNIX. Впоследствии с их помощью на UNIX решены многие прикладные задачи. На сегодняшний день они являются неотъемлемой частью инструментария программиста в UNIX. Благодаря тому, что их исходные тексты написаны на Си, они легко переносятся в другие ОС.

В дальнейших главах учебного пособия даются необходимые сведения по использованию LEX и YACC и приводятся пошаговые примеры разработки сложных программ с их использованием.

---

* Иногда так называют только генератор программ синтаксического разбора.

Надо заметить, что в последнее время широкое распространение получили реализации GNU**, называющиеся соответственно FLEX и BISON. Обладая большей мощностью, эти реализации в основе своей повторяют «классические» LEX и YACC.

** Независимая группа разработчиков программного обеспечения, разрабатывающая и распространяющая свои разработки в соответствии с так называемой Лицензией GNU. GNU — это аббревиатура-рекурсия, буквально раскрываемая как «GNUs Not Unix».

Глава 2
Генератор программ синтаксического разбора YACC

2.1. Общие сведения

Из инструментальных средств построения программ для лексического и синтаксического анализа мы рассмотрим прежде всего генератор программ синтаксического разбора YACC как наиболее часто применяемый на практике. Дело в том, что во многих случаях лексический анализатор оказывается довольно прост, и его реализация «вручную» (без использования LEX) оказывается намного эффективнее.

Генератор программ синтаксического разбора YACC является стандартной частью системы программирования в ОС UNIX. Он использовался для разработки компиляторов с языков Паскаль, Ratfor, АПЛ, Си, а также для менее традиционных приложений, включающих, например, язык описания математических формул EQN для системы подготовки документов.

Генератор YACC преобразует контекстно свободную грамматику, заданную в файле спецификаций, в программу на Си, выполняющую так называемый LALR(1)* алгоритм разбора, использующий метод разбора снизу вверх. Этот алгоритм применим к подавляющему большинству существующих языков программирования.

Заданная грамматика может быть неоднозначной. Потенциально возможно два типа неоднозначности — это конфликты сдвиг/свёртка и свёртка/свёртка. Разрешение конфликтов рассмотрим в дальнейшем, однако сразу необходимо заметить, что существуют грамматики, для которых конфликт типа сдвиг/свёртка является нормальным явлением и правильно разрешается генератором YACC по умолчанию (выбирается сдвиг). Конфликт же типа свёртка/свёртка скорее всего говорит не о неоднозначности, а о серьёзной ошибке в задании грамматики. Такого типа конфликтов необходимо избегать.

---

* Эта аббревиатура означает, что мы просматриваем цепочку символов слева направо, выполняя свёртку правых символов и заглядывая при свёртке не более чем на одну лексему вперёд.

2.2. Получение рабочей программы

Файл спецификаций, имеющий по традиции расширение `.y`, будем называть yacc-программой. Условимся, что во всех наших примерах, если не оговорено что-либо другое, этот файл называется `file.y`. В файле с yacc-программой записываются только грамматические зависимости лексем. Смысл самих лексем не конкретизируется.

Трансляция файла с yacc-программой выполняется командой

```
yacc file.y
```

В результате трансляции получается файл `y.tab.c`, содержащий функцию `yyparse()`, которая как раз и выполняет всю процедуру синтаксического анализа. Функция `yyparse()` вызывает функцию `yylex()` для получения каждой новой лексемы и функцию `yyerror()` в случае обнаружения синтаксической ошибки во входном потоке. Функции `yylex()` и `yyerror()`, а также функция `main()`, из которой происходит вызов самой функции `yyparse()`, не создаются генератором `yacc`, и могут быть взяты из стандартной библиотеки UNIX, но чаще всего описываются самим программистом. В реализации YACC для ОС Linux стандартная библиотека содержит только функции `yyerror()` и `main()`. В силу примитивности функций, предоставляемых стандартной библиотекой, рекомендуется, чтобы все эти три функции были обязательно описаны пользователем.

Полученный файл `y.tab.c` транслируется далее в объектный модуль с использованием стандартного транслятора с языка Си. Сборка рабочей программы выполняется стандартной программой редактирования связей с подключением модулей, содержащих рассмотренные выше функции.

Команда `yacc` может иметь два флага.

Если задан флаг `-v`, строится файл `y.output`, который содержит описание таблиц разбора и конфликтов, порождаемых неоднозначной грамматикой.

Если задан флаг `-d`, строится файл `y.tab.h`, содержащий операторы `#define`, определяющие имена лексем. Это позволяет использовать имена лексем не только в файле `y.tab.c`. В частности, можно использовать имена лексем в лексическом анализаторе, описанном в другом `.c` файле.

2.3. Формат файла спецификаций

Файл, содержащий yacc-программу, имеет следующую структуру:

```
    раздел деклараций
%%
    раздел правил
%%
    раздел функций
```

Раздел функций может быть опущен. Тогда вторая пара %% не нужна.

С другой стороны, для простой программы в разделе функций могут быть заданы все необходимые для работы программы функции — `main()`, `yylex()` и `yyerror()`. В этом случае для получения рабочей программы достаточно дать две команды:

```
yacc file.y
cc y.tab.c
```

При этом мы получим рабочую программу в файле с именем `a.out`.

Функции в разделе функций записываются обычным образом и при трансляции программой `yacc` просто копируются в файл `y.tab.c`. Поэтому этот раздел файла спецификаций YACC специально рассматривать мы не будем.

Отметим также сразу, что в отличие от БНФ в yacc-программе принято (хотя совсем и не обязательно) записывать имена терминальных символов прописными буквами, а имена нетерминалов — строчными. Особенностью многих реализаций является также необходимость записывать терминалы и нетерминалы только английскими буквами.

2.3.1. Раздел деклараций yacc-программы

Раздел деклараций yacc-программы включает строки-директивы и строки исходного текста. Строки-директивы начинаются символом %, стоящим в первой позиции строки. Строки исходного текста помещаются внутрь пары строк-директив `%{` и `%}` и без изменений копируются в выходной файл `y.tab.c`. Строки исходного текста могут содержать директивы препроцессора компилятора Си и описания переменных. Область действия этих переменных будет распространяться на все разделы программы, размещённой в файле `y.tab.c`.

Строки-директивы используются для декларации имён лексем, их приоритетов и ассоциативности, а также имени начального нетерминала и некоторых других возможностей.

Декларация имён лексем осуществляется директивой `%token` и имеет вид

```
%token список_имён_лексем
```

Строка может быть повторена несколько раз. Перечисленные в данных строках имена используются для представления терминальных символов. Любое другое имя, используемое в грамматике, но не объявленное с помощью `%token`, считается нетерминальным символом.

Директивами `%token` должны быть описаны все используемые имена лексем, кроме литералов. Декларация имён ни в коей мере не обеспечивает распознавания лексем, которое должно проводиться в лексическом анализаторе.

Пример декларации имён лексем:

```
%token IDENT CONST IF THEN ELSE
```

В приведённом примере описывается пять имён лексем. Первые два имени соответствуют распознаванию лексем «идентификатор» и «константа». Три последние демонстрируют тот факт, что если ключевые слова распознаваемого входного потока распознаются как лексемы лексическим анализатором, то имена лексем могут совпадать с этими ключевыми словами. Имена лексем не должны лишь совпадать с зарезервированными словами языка Си.

Приоритеты и ассоциативность лексем задаются в секции деклараций посредством директив `%left`, `%right` и `%nonassoc`:

```
%left список_лексем
%right список_лексем
%nonassoc список_лексем
```

Каждая директива приписывает всем перечисленным в её списке лексемам одинаковый приоритет. Строки записываются сверху вниз в порядке увеличения приоритета или силы связывания. Устанавливаемый директивами приоритет не имеет численного выражения. Директива `%left` задаёт для лексем и литералов, перечисленных в этой строке, левую ассоциативность, директива `%right` — правую ассоциативность, а директива `%nonassoc` означает, что операции не ассоциированы между собой (то есть если при анализе потребуется разрешение ассоциативности, будет сгенерирована ошибка). Если какие-либо лексемы перечислены в указанных строках, их можно отдельно не объявлять с помощью директивы `%token`. Если унарная операция обозначается той же лексемой, что и бинарная (например, унарный минус и операция вычитания), то необходимо согласование их приоритетов. Такое согласование производится с помощью директивы `%prec` непосредственно в синтаксическом правиле в разделе правил (см. ниже).

Пример задания приоритетов и ассоциативности лексем:

```
%token OR AND XOR NOT
%right '='
%left NOT
%left OR XOR
%left AND
%left '+' '-'
%left '*' '/'
```

В примере самый низкий приоритет имеет лексема (литерал) `'='`, самый высокий — лексемы `'*'` и `'/'`. Информация о приоритетах и ассоциативности используется генератором `yacc` для разрешения конфликтов грамматического разбора. Использование лексемы само по себе не требует задания её приоритета или ассоциативности.

При первом появлении лексемы или литерала в секции деклараций за каждым из них может следовать натуральное число, рассматриваемое как **номер типа** лексемы (код лексемы). По умолчанию номера типов всех лексем определяются следующим образом:

- для литерала номером типа лексемы считается числовое значение символа, рассматриваемое как однобайтовое целое число;
- лексемы, обозначенные именами, в соответствии с очерёдностью их объявления получают последовательные номера, начиная с 257;
- специальная лексема `error`, зарезервированная для обработки ошибок, получает номер типа 256.

Переопределением номеров типов лексем увлекаться не стоит. Как правило, достаточно использовать алгоритм назначения номеров по умолчанию. Переопределив при необходимости ряд номеров типов лексем, программист должен проверить их уникальность.

Для каждого имени лексемы, независимо от того, переопределён ли её номер пользователем, `yacc` генерирует в выходном файле строки вида

```
#define имя_лексемы номер_типа
```

Например, директива

```
%token LIT 258
```

породит оператор

```
#define LIT 258
```

Декларация имени начального нетерминала выполняется директивой `%start`:

```
%start имя_начального_нетерминала
```

Если директива не указана, то действует правило по умолчанию, при котором в качестве начального символа грамматики выбирается

нетерминал, указанный в левой части самого первого правила грамматики секции правил yacc-программы.

По умолчанию стек значений синтаксического анализатора (значений, возвращаемых лексическим анализатором) содержит целые числа. Это соглашение можно изменить. Прежде всего, можно просто изменить тип всех значений, сохраняемых в стеке. Например, для хранения в стеке только вещественных чисел можно применить следующую последовательность директив:

```
%{
#define YYSTYPE double
%}
```

Если же необходимо хранить в стеке объекты различных типов, то тип стека может быть задан как объединение значений различных типов. Это выполняется с помощью директивы `%union`, имеющей следующий формат:

```
%union{
  тело объединения на Си
}
```

В этом случае при выполнении синтаксического разбора YACC будет осуществлять проверку типов в семантической части синтаксических правил. Для осуществления возможности проверки типов задание соответствия «тип — лексема» должно быть выполнено явно.

Имена элементов объединения, описывающего возможные типы объектов, хранимых в стеке, связываются с именами нетерминалов с помощью директивы `%type`. В этой директиве не должны встречаться литералы или номера лексем. Для связи элементов объединения с именами терминалов используется уже знакомая нам директива `%token`. Синтаксис директив в этом случае проще всего продемонстрировать на примере:

```
%union{
  double val;   /* фактическое числовое значение */
  int index;    /* индекс в массиве переменных */
}
%token <val> NUMBER
%token <index> VALUE
%type <val> formula
```

В примере `NUMBER` и `VALUE` — некоторые терминальные символы, а `formula` — какой-то нетерминал, используемый в семантической части синтаксических правил по его значению.

2.3.2. Раздел правил yacc-программы

Этот раздел содержит контекстно свободную грамматику, по которой будет вести разбор порождённый YACC синтаксический анализатор. Секция состоит из одного или нескольких синтаксических правил, имеющих форму, близкую к БНФ*:

```
имя_нетерминала : определение ;
```

Здесь символы : и ; — зарезервированные символы YACC. Определение — последовательность из литералов, имён лексем и нетерминальных символов, за которой может следовать семантическое действие. В процессе разбора такая последовательность заменяется на `имя_нетерминала`, указанное в левой части правила. Такая процедура получила название **свёртка**.

Правила могут записываться в произвольном порядке. Однако, если имя нетерминала, являющегося начальным символом грамматики, не задано явно в разделе деклараций, то необходимо, чтобы правило, определяющее этот нетерминал, было задано первым.

Все нетерминалы, используемые в правилах, должны быть определены. То есть имя каждого из них должно появиться в левой части хотя бы одного правила. Для примера покажем, как можно было бы задать правило для заголовка функции в языке Си:

```
function : NAME '(' arglist ')' ;
```

Здесь правило состоит из четырёх элементов — двух литералов (символы (и), взятые в одиночные кавычки), лексемы `NAME`, очевидно, соответствующей имени функции, и нетерминала `arglist`, соответствующего списку аргументов.

Допустимо задание нескольких правил, определяющих один нетерминал, т. е. правил с одинаковой левой частью. Например, правила, определяющие виды операторов в некотором языке, могут выглядеть следующим образом:

```
statement : assign_stat ;
statement : if_then_stat ;
statement : goto_stat ;
```

---

* В приводимых в параграфе примерах часто в качестве имён нетерминалов для бóльшей наглядности используются слова, содержащие русские буквы. Это не противоречит принципам работы YACC, но реальный транслятор `yacc` имена, составленные не из цифр и английских букв, использовать не разрешает.

Правила с общей левой частью можно задавать в сокращённой записи без повторения левой части, используя для разделения альтернативных определений символ |:

```
statement : assign_stat
    | if_then_stat
    | goto_stat ;
```

Семантическое действие может быть связано с любым правилом. **Действие** — это набор операторов языка Си, которые будут выполняться при каждом распознавании правила. Действие не является обязательным элементом правила. Оно заключается в фигурные скобки и помещается вслед за определением:

```
имя_нетерминала : определение {действие} ;
```

На операторы, входящие в действие, не накладывается никаких ограничений. В частности, в действиях могут содержаться вызовы любых функций на языке Си. Отдельные операторы могут быть помечены, к ним возможен переход из других действий.

Существует возможность задания действий, которые будут выполняться по мере распознавания отдельных фрагментов правила. Действие в этом случае помещается после одного из элементов правой части так, чтобы положение действия соответствовало моменту разбора, в который данному действию будет передано управление. Например, в правиле

```
if_then_stat : IF '(' expression ')' { действие_1 }
    THEN statement ';' { действие_2 } ;
```

действия заданы с таким расчётом, чтобы `действие_1` выполнялось при распознавании правой круглой скобки, а `действие_2` — по окончании разбора всего правила.

В действиях можно использовать так называемые **позиционные переменные**. Эти переменные записываются как `$1`, `$2`, ..., где `$i` соответствует значению i-го элемента правой части правила. Элементы правой части правила нумеруются слева направо без различий для лексем и нетерминалов. Например, для правила

```
function : NAME '(' arglist ')' ;
```

имеем следующие отношения позиционных переменных:

```
$1  Name
$2  '('
$3  arglist
$4  ')'
```

Значение определяемого правилом нетерминала должно присваиваться переменной `$$`. Например, правило и действие для математической операции могут быть записаны так:

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

Если в действии явно не задано присвоение переменной $$, а также если действие в правиле отсутствует, значением нетерминала после свёртки становится по умолчанию значение первого элемента правой части правила, т. е. неявно выполняется правило

```
$$=$1;
```

Если включён контроль типов, то при несовпадении типа позиционной переменной и типа лексемы в правиле выдаётся сообщение об ошибке.

Если в правой части правила используются действия, выполняемые в процессе распознавания правила, то каждое такое действие (группа, заключённая в фигурные скобки) соответствует своей позиционной переменной, значение которой считается пустым.

Существует два специфических вида правил, на которые полезно обратить внимание — это пустое и рекурсивное. Пустое правило имеет вид

```
имя_нетерминала : ;
```

или

```
имя_нетерминала : { действие } ;
```

Сочетание пустого правила с другим, определяющим тот же нетерминал, является одним из способов указать на необязательность во входном потоке соответствующей конструкции.

Правила часто описывают некоторую конструкцию рекурсивно, т. е. правая часть может включать определяемый нетерминал. Различают леворекурсивные правила:

```
имя_нетерминала : имя_нетерминала повторяющийся_фрагмент ;
```

и праворекурсивные правила:

```
имя_нетерминала : повторяющийся_фрагмент имя_нетерминала ;
```

Генератор YACC допускает оба вида рекурсивных правил, однако при использовании правил с правой рекурсией объём анализатора увеличивается, так как существует опасность переполнения внутреннего стека анализатора во время разбора.

Рекурсивные правила необходимы при задании последовательностей и списков. Например:

```
последовательность: элемент
    | последовательность элемент ;
```

или

```
список : элемент
    | список ',' элемент ;
```

Нетерминальные символы, связанные с последовательностями или списками разнородных элементов, могут описываться произвольным

числом рекурсивных и нерекурсивных правил. Например, определение идентификатора в программе может быть задано следующим правилом:

```
идентификатор : БУКВА
    | '$'
    | идентификатор БУКВА
    | идентификатор ЦИФРА
    | идентификатор '_' ;
```

Следует обратить внимание на то, что при задании рекурсивных правил для определяемого нетерминала должно быть задано хотя бы одно правило без рекурсии. Так, при задании списков или последовательностей, которые могут быть пустыми (отсутствовать), в качестве нерекурсивного правила используется пустое:

```
последовательность :
    | последовательность элемент ;
```

Сочетание пустых и рекурсивных правил является удобным способом представления грамматик и ведёт к большей их общности. Однако некорректное использование пустых правил может вызвать конфликтные ситуации из-за неоднозначности выбора нетерминала.

Для унарных операций в общем случае должен быть указан их приоритет. В том случае, если унарная и бинарная операции используют одно и то же обозначение, для изменения приоритета в одном отдельном правиле используется ключевое слово `%prec`. Это слово ставится непосредственно после тела правила, и за ним записывается имя лексемы или литерал, к приоритету которого надо приравнять приоритет данного правила. Например, для задания явного различия приоритетов унарного минуса и операции вычитания могут быть использованы следующие операторы:

```
%token NUMBER
%left '+' '-'
%left UNARYMINUS
%%
expr : NUMBER { $$=$1; }
    | expr '-' expr { $$=$1-$3; }
    | '-' expr %prec UNARYMINUS { $$=-$2; } ;
```

В данном случае символ унарного минуса имеет тот же приоритет, что и `UNARYMINUS`. А поскольку `UNARYMINUS` в разделе деклараций описан с наивысшим приоритетом, то и операция унарного минуса имеет наивысший приоритет.

2.4. Неоднозначность

Как уже было замечено, контекстно свободная грамматика, на базе которой составляется уасс-программа, может быть неоднозначной. Генератор YACC распознаёт два типа неоднозначностей (конфликтов) — типа сдвиг/свёртка (shift/reduce) и типа свёртка/свёртка (reduce/reduce).

По умолчанию для разрешения конфликтов синтаксического разбора используются следующие правила:

- в случае конфликта сдвиг/свёртка выполняется сдвиг (то есть в результате будет свёрнута самая длинная из опознанных конструкций), что для большинства грамматик, описывающих современные языки программирования, оказывается правильным решением;
- в случае конфликта свёртка/свёртка для свёртки используется то правило, которое записано первым при описании грамматики.

Кроме того, для разрешения неоднозначностей используется информация об ассоциативности и приоритете. Конфликт сдвиг/свёртка всегда разрешается в пользу действия (сдвиг или свёртка) с более высоким приоритетом. Больший приоритет правила вызывает свёртку по нему, больший приоритет лексемы — сдвиг. Однако задание приоритетов не ведёт к устранению конфликтов и не делает грамматику однозначной. Если приоритет одинаков, используется ассоциативность. Левая ассоциативность означает свёртку, правая — сдвиг, отсутствие ассоциативности означает ошибку.

Приоритет правила автоматически определяется приоритетом последней лексемы в нём. Если в разделе деклараций для этой лексемы не задан приоритет или если правая часть правила вообще не содержит лексем, то приоритет правила не определён. Этот принцип можно отменить явным заданием приоритета правила с помощью директивы `%prec`, как описано выше.

2.5. Обработка ошибок

Стандартной реакцией синтаксического анализатора на ошибку является выдача сообщения `Syntax error` и прекращение разбора. Эту реакцию можно изменить, например, сделав сообщение об ошибке более информативным, задав собственную процедуру `yyerror()`. Однако желательно, чтобы анализатор в этом случае продолжал просмотр входного потока. Для этого программист должен ввести в грамматику дополнительные правила, указывающие, в каких конструкциях синтаксические ошибки являются допустимыми.

Одновременно эти правила определяют путь дальнейшего разбора для ошибочных ситуаций.

Для указания точек допустимых ошибок используется зарезервированное имя лексемы `error`. Пример:

```
a : b c d
    | b c error ;
```

В данном случае предполагается, что после того, как будут распознаны элементы `b` и `c`, может не последовать элемент `d`, что будет распознано как ошибка. Правила, содержащие лексему `error`, выполняются так же, как все остальные правила. В частности, с ними могут быть связаны действия. Опустив внутренние подробности обработки анализатором лексемы `error`, покажем, как можно вернуть анализатор в состояние, позволяющее продолжить синтаксический анализ входного потока.

Общая форма правила с действиями по восстановлению имеет вид

```
оператор : error {
    resynch();
    yyclearin;
    yyerrok;
    } ;
```

Здесь `resynch()` — это заданная пользователем функция, которая просматривает входной поток до начала очередного правильного оператора. `yyclearin` и `yyerrok` — это специальные операторы YACC, первый из которых стирает из входного потока лексему, вызвавшую ошибку, а второй гасит состояние ошибки анализатора.

2.6. Связь с лексическим анализатором

Для получения очередной лексемы анализатор синтаксического разбора вызывает функцию `yylex()`. Функция не имеет аргументов и должна вернуть в качестве кода возврата номер очередной лексемы. Если лексема, кроме номера, имеет значение, то оно должно быть помещено лексическим анализатором в переменную `yylval`. Тип переменной `yylval` соответствует типу внутреннего стека синтаксического анализатора. Если стек анализатора описан в разделе деклараций с помощью директивы `%union`, то и переменная `yylval` является объединением. Например, если предположить, что тип стека синтаксического анализатора задан следующим образом:

```
%union{
  double val;   /* фактическое числовое значение */
  int index;    /* индекс в массиве переменных */
}
```

```
%token <val> NUMBER
%token <index> VALUE
%type <val> formula
```
то где-то в функции лексического анализатора (`yylex()`) операторы, соответствующие распознаванию во входном потоке числа, могут выглядеть следующим образом:
```
    scanf("%f",&(yylval.val));
    return NUMBER;
```
Здесь в переменную `yylval` помещается значение распознанной лексемы, а её тип `NUMBER` передаётся синтаксическому анализатору как код возврата функции `yylex()`.

2.7. Стандартные функции

Простейшая функция `yylex()` имеет вид
```
yylex()
{
    return getchar();
}
```
Однако при таком её определении вся работа по распознаванию элементов и конструкций языка ложится на синтаксический анализатор. Как уже было показано, такой подход значительно усложняет синтаксический анализатор, делает его тяжеловесным и трудномодифицируемым.

На практике желательно, чтобы функция `yylex()` распознавала типы лексем, такие, как ключевые слова, числа, идентификаторы. Поэтому, как правило, эта функция разрабатывается параллельно с созданием синтаксического анализатора.

Функции `main()` и `yyerror()` для окончательного варианта программы также желательно разработать самостоятельно. Однако для целей отладки собственно синтаксического анализатора вполне могут подойти простейшие варианты:
```
#include <stdio.h>

main()
{
    yyparse();
}

yyerror(str)
char *str;
{
    fprintf(stderr,"%s",str);
}
```

Глава 3
Генератор программ лексического разбора LEX

3.1. Общие сведения

Приступая к описанию генератора LEX, сразу оговоримся, что в настоящей главе приведены сведения из документации по LEX, которая, как показывает личный опыт автора, не всегда оказывается точной. Вернее сказать, если по отдельности все описанные здесь конструкции работают вполне нормально, то, будучи объединёнными в lex-программу, они иногда начинают конфликтовать, в результате чего получаемый в итоге лексический анализатор не работает. Проблема, видимо, заключается в том, что существуют различные более или менее удачные реализации LEX, которые, по идее, должны работать с единым входным языком. Однако в силу того, что LEX используется на практике значительно реже, чем YACC, его базовый алгоритм оказался меньше отработан. В связи с этим автор рекомендует разрабатывать лексический анализатор по шагам, добавляя необходимые правила и шаблоны постепенно, и следить при этом за работоспособностью получаемого анализатора. Если после добавления очередного правила анализатор перестал работать, значит, данное правило нужно просто записать каким-нибудь другим способом, что обычно возможно.

Читателю рекомендуется воспринимать приводимые в этой главе примеры исключительно как иллюстрации к описанию языка. Проверка работоспособности этих примеров не проводилась. Проверенные примеры приводятся в последующих главах настоящего учебного пособия.

Генератор LEX по файлу спецификаций, который в дальнейшем будем называть lex-программой, строит детерминированный конечный автомат для распознавания во входном потоке цепочек, соответствующих регулярным выражениям, заданным в lex-программе. Автомат читает входной поток из файла `yyin` (обычно это стандартный ввод) и при необходимости выводит цепочки в файл `yyout` (обычно это стандартный вывод).

В файле спецификаций также могут быть заданы действия, которые необходимо выполнить при распознавании цепочек, соответствующих регулярным выражениям.

Генератор LEX может быть использован для создания программ анализа и обработки текста как самостоятельно, так и в паре с генератором синтаксического разбора YACC.

3.2. Получение рабочей программы

Файл с lex-программой обычно имеет расширение `.l`. Будем считать, что для всех наших примеров, если не оговорено другое, этот файл имеет имя `file.l`. Файл содержит список регулярных выражений, в соответствии с которыми должны быть распознаны цепочки символов во входном потоке, и действия, которые должны быть выполнены при распознавании этих цепочек. Действия записываются в виде фрагментов на языке Си.

Трансляция файла с lex-программой выполняется командой

```
lex file.l
```

В результате трансляции получается файл `lex.yy.c`, содержащий программу лексического анализатора на языке Си. Программа приводится в действие вызовом функции `yylex()`.

Функция `yylex()` в процессе работы может вызывать несколько функций с предопределёнными именами. Это `yywrap()`, `input()`, `output()`, `unput()`, `yymore()`, `yyless()`. Все эти функции по умолчанию определены, но пользователь может их при необходимости переопределить. Смысл этих функций описан в последующих параграфах.

Функция `main()` не создаётся генератором LEX. Она должна быть написана пользователем. Так же как и в случае с генератором YACC, в UNIX существует стандартная библиотека LEX, из которой эта функция может быть взята. Но стандартная функция `main()` покрывает только определённые и самые примитивные случаи использования LEX, поэтому обычно функцию `main()` необходимо разработать самостоятельно.

Полученный файл `lex.yy.c` транслируется далее в объектный модуль с использованием стандартного транслятора с языка Си. Сборка рабочей программы выполняется стандартной программой редактирования связей с подключением модулей, содержащих необходимые функции.

Команда `lex` может иметь следующие флаги:

- `-t` вывод результата в стандартный вывод;
- `-v` вывод краткой характеристики размеров построенного анализатора;
- `-n` запрет вывода размеров анализатора (режим по умолчанию);
- `-f` отмена упаковки выходных таблиц. В этом случае `lex` работает быстрее, но может строить только небольшие программы.

3.3. Формат файла спецификаций

Файл, содержащий lex-программу, имеет следующую структуру:

```
    раздел деклараций
%%
    раздел правил
%%
    раздел функций
```

Раздел функций может быть опущен. Тогда вторая пара %% не нужна.

С другой стороны, для простой программы в разделе функций могут быть заданы и переопределены все необходимые для работы программы функции, в частности `main()`. В этом случае для получения рабочей программы достаточно дать две команды:

```
lex file.l
cc lex.yy.c
```

При этом мы получим рабочую программу в файле с именем `a.out`.

Функции в разделе функций записываются обычным образом и при трансляции программой `lex` просто копируются в файл `lex.yy.c`. Поэтому этот раздел файла спецификаций LEX специально рассматривать мы не будем.

3.3.1. Регулярные выражения

Для задания лексем, которые необходимо выделить из входного потока, в LEX используется механизм, широко применяемый во всей ОС UNIX. Это так называемые регулярные выражения*.

Регулярные выражения, или **шаблоны**, используются для задания множества символьных строк. Говорят, что элементы этого множества соответствуют регулярному выражению.

Регулярное выражение записывается как строка символов и метасимволов.

Метасимволы, называемые иногда **операторами**, — это символы, имеющие в шаблоне специальное значение. Перечислим их все (самый первый — «точка»):

```
. \ ^ ? * + | $ / %
[] {} () <>
```

* На самом деле LEX поддерживает работу с более мощным языком задания регулярных выражений, чем тот, который принят в остальных частях ОС UNIX.

Символы, не являющиеся метасимволами, определяют сами себя. При описании регулярных выражений под символом подразумевается любой символ, отличный от символа перевода строки.

Если необходимо отменить специальное значение метасимвола, то перед ним нужно указать обратную наклонную черту или указать его в двойных кавычках. Например:

`abc+` — символ `+` — оператор;
`abc\+` — символ `+` — просто символ;
`abc"+"` — символ `+` — просто символ;

Вообще, любой символ, заданный в двойных кавычках, — это всегда просто символ. Символ пробела в выражении, если он не находится внутри квадратных скобок, также необходимо заключать в двойные кавычки.

Далее перечислим метасимволы и их значения:

`.` любой символ, кроме символа «перевод строки»;

`\n` символ «перевод строки»;

`\t` символ табуляции;

`\b` возврат курсора на один шаг назад;

`\код` указание символа восьмеричным кодом;

`\c` символ `c`, где `c` — произвольный символ, кроме цифры и скобок (к самому символу `\` это экранирование также применимо);

`^` символ обозначает начало строки, если он стоит в начале регулярного выражения;

`$` символ обозначает конец строки, если он стоит в конце регулярного выражения;

`[...]` один символ из числа перечисленных в скобках. Например, `[abc]` означает «символ `a`, или символ `b`, или символ `c`». Символы могут быть заданы диапазонами с использованием символа «`-`». Например, `[0-9]` задаёт все цифры, а `[+-0-9]` — все цифры и знаки `+` и `-`. Если символ `^` является первым из перечисленных в скобках, то регулярному выражению соответствует любой символ, не входящий в число перечисленных в скобках (то есть в скобках задаются запрещённые символы). Внутри скобок символы `.`, `*`, `[` и `\` теряют свой специальный смысл;

`ab/cd` `ab` учитывается только тогда, когда за ним следует `cd`;

`ab|cd` или `ab`, или `cd`;

Метасимволы–повторители:

`*` любое (в том числе ни одного) число вхождений символа или группы символов, стоящего перед `*`;

`?` одно или ни одного вхождения символа или группы символов, стоящего перед `?`;

`+` одно или больше вхождений символа или группы символов, стоящего перед `+`;

`x{n,m}` здесь `n` и `m` — натуральные числа, причём `m>n`. Означает «от `n` до `m` вхождений `x`»;

Специальные символы:

`<s>r` регулярное выражение `r` будет распознано, только если программа находится в состоянии `s`;

`{имя}` вместо `{имя}` будет подставлено определение имени (строки подстановки) из раздела деклараций lex-программы. Например:

```
БУКВА    [A-ZA-Яa-za-я]
ЦИФРА    [0-9]
ИДЕНТИФ  _?{БУКВА}({БУКВА}|{ЦИФРА})*
```

В примере объект ИДЕНТИФ (идентификатор) определяется как необязательный символ подчёркивания, за которым следует буква, за которой любое число букв и/или цифр в произвольном порядке.

Ещё несколько примеров:

`^ABC` Выбрать цепочку символов `ABC`, если она начинает строку.

`xyz$` Выбрать цепочку `xyz`, если она заканчивает строку.

`x{2,7}` От двух до семи вхождений `x`.

`x{2,}` Два и более вхождений `x`.

`[0-9]+` Цепочка цифр ненулевой длины.

`[0-9]?` Необязательная цифра.

3.3.2. Раздел деклараций lex-программы

В разделе деклараций могут быть заданы следующие элементы: определения строк подстановки, фрагменты программы пользователя, начальные условия, изменения размеров внутренних массивов и таблицы наборов символов. Очень важно, с какой позиции записываются строки.

Определением строки подстановки считается любая строка, начинающаяся с первой позиции и не заключённая в операторы `%{` и `%}`. Формат определения:

```
имя значение
```

где имя — любая последовательность из букв и цифр, начинающаяся с буквы. В качестве разделителя между именем и значением используется один или более пробелов или табуляций. Например, строка

```
ЦИФРА    [0-9]
```

определяет для имени {ЦИФРА} подстановку [0-9]. Из этого общего правила есть несколько исключений, описанных далее.

Фрагменты программ пользователя задаются в форме

```
%{
строки фрагмента с любой позиции
%}
```

Все строки фрагмента программы пользователя, заданные таким образом, будут являться внешними для любой функции программы lex.yy.c.

Начальные условия задаются в форме

```
%START метка1 метка2 ...
```

Если начальные условия определены, то эта строка должна быть первой в lex-программе. Начальные условия определяют список состояний, в которые может переходить лексический анализатор при разборе входного потока.

Таблица символов задаётся в виде

```
%T
целое_число строка_символов
     ...
     ...
%T
```

и предназначена для осуществления внутренней перекодировки символов при обработке. От более подробных объяснений воздержимся, поскольку на практике эта возможность LEX обычно не используется.

Генератор lex имеет внутренние таблицы, размеры которых ограничены. При построении программы лексического анализа может произойти переполнение любой из этих таблиц. Программист имеет возможность изменить размеры этих таблиц (сокращая одни и увеличивая другие) так, чтобы они не переполнялись. Это осуществляется с помощью команд изменения внутренних массивов, имеющих вид:

```
%x число
```

где число — новый размер массива; x — одна из букв: p — позиции, n — состояния, e — узлы дерева, a — упакованные переходы, k — упакованные классы символов, o — массив выходных элементов. Текущие размеры таблиц и степень их заполнения можно узнать, используя при трансляции lex-программы ключ -v.

В разделе деклараций могут быть указаны комментарии, которые записываются, как в языке Си, но не с первой позиции строки.

3.3.3. Раздел правил lex-программы

В этом разделе записываются правила разбора входного потока. **Правила** — это таблица, в которой левая колонка содержит регулярное выражение (шаблон), а правая — действие (фрагмент программы на языке Си, либо оператор LEX), выполняемое при распознавании регулярного выражения. Регулярное выражение отделяется от действия одним или более пробелом или табуляцией. Действие может быть записано на нескольких строках и в этом случае должно быть заключено в фигурные скобки.

Правила могут быть простыми и помеченными. Простое правило имеет вид

```
шаблон действие
```

Помеченное правило имеет вид

```
<метка>шаблон действие
```

или

```
<метка1,метка2,...>шаблон действие
```

Все используемые метки должны быть описаны директивой `%START` в разделе деклараций lex-программы.

Простые правила выполняются всегда. Помеченные — только после активизации их оператором `BEGIN`. Оператор `BEGIN метка` просто расширяет список активных правил, активизируя помеченные меткой `метка`. Если используется несколько меток, то после указанного оператора независимо от предыдущего состояния активными окажутся только обычные правила и правила, помеченные меткой `метка`. Оператор `BEGIN 0` оставляет активными только простые правила.

В качестве первого правила lex-программы может быть задано правило с действием без шаблона, которое будет выполняться при каждом вызове функции `yylex()`, до перехода к разбору входного потока. В частности, в этом правиле может быть задан и оператор `BEGIN`.

Действие в правиле lex-программы выполняется, если правило активно и распознана лексема, соответствующая шаблону этого правила. Если, однако, комбинация символов входного потока не соответствует ни одному шаблону, то она будет просто напечатана на выходе. Можно даже сказать, что действие — это то, что делается вместо копирования входного потока символов на выход.

В качестве действия в правиле может быть задан пустой оператор «;». В таком случае строки, соответствующие данному регулярному выражению, будут пропускаться. Например, правило

```
[ \t\n]      ;
```

запрещает вывод всех символов пробела, табуляции и перевода строки. А правило

```
.|\n         ;
```

записанное в конце раздела правил, подавляет вывод всех цепочек символов, не соответствующих ни одному правилу.

Для нескольких шаблонов может быть указано одно действие. С этой целью применяют оператор «|». Например, записанное выше правило может быть переписано как

```
" " |
\t  |
\n  ;
```

Очень часто используется операция вывода распознанного текста. Для этого используется оператор `ECHO`. Например, в следующем правиле выводятся все слова, состоящие из прописных английских букв:

```
[A-Z]+  ECHO;
```

Входной текст, распознанный по шаблону, помещается в предопределённую строку `yytext`, а его длина — в переменную `yyleng`. Эти имена доступны для действий на языке Си. Например, предыдущее правило можно переписать с действием на языке Си:

```
[A-Z]+  printf("%s",yytext);
```

При распознавании цепочек правила проверяются в порядке записи сверху вниз. При этом LEX делает попытку выделить из входного потока наиболее длинную цепочку. Если несколько правил соответствуют одной и той же цепочке, применяется правило, которое записано первым.

Как правило, для каждой входной цепочки срабатывает только одно правило. Иногда желательно после срабатывания правила продолжить сопоставление для текущей цепочки символов. Эта возможность обеспечивается оператором `REJECT`, который буквально означает «выбрать следующую альтернативу». Например, предположим, что мы хотим подсчитать в тексте все вхождения цепочек «он» и «она». Правила для подсчёта могли бы выглядеть следующим образом:

```
она f++;
он  m++;
.   |
\n  ;
```

Так как «она» включает в себя «он», анализатор не будет распознавать те вхождения «он», которые включены в «она». Предположим, что нас это не устраивает. Модифицированные правила, устраняющие описанный выше недостаток, выглядят так:

```
она {f++; REJECT;}
он  {m++; REJECT;}
.   |
\n  ;
```

Правила с метками удобно использовать для описания лексического анализатора, имеющего несколько разных состояний (т. е. разных конечных автоматов) с явным переключением с одного на другое. Например, следующие правила обеспечивают удаление комментариев из программы на языке Си (`out` — вне комментария; `in` — внутри):

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

Комментарии в разделе правил могут быть записаны только внутри блоков действий.

3.3.4. Предопределённые функции

Лексический анализатор, построенный с помощью LEX, определяет ряд специальных функций. Эти функции условно можно разделить на два типа: используемые анализатором и используемые программистом.

К используемым анализатором функциям относятся следующие: `yywrap()`, `input()`, `output()` и `unput()`. Эти функции вызываются самим анализатором и могут быть переопределены программистом для решения своих нестандартных задач.

Функции, используемые программистом, — это `yymore()` и `yyless()`. Эти функции могут быть использованы программистом при описании действий в правилах, расширяя возможности анализа входного потока.

Далее опишем назначение функций и дадим формат их вызова.

`input()` — читает и возвращает символ из входного потока символов.

`output(c)` — выводит в выходной поток символ `c`.

`unput(c)` — возвращает символ `c` обратно во входной поток для повторного чтения.

`yywrap()` — возвращает `1` по завершении входного потока символов, иначе возвращает `0`. Эта функция вызывается анализатором автоматически по обнаружении признака конца файла во входном потоке и является единственным способом завершить его работу, если он используется как самостоятельная программа. Определённая генератором `lex` функция `yywrap()` всегда возвращает `1`, что служит сигналом к завершению работы. Если необходимо начать ввод данных из другого источника и продолжить работу, программист должен переопределить функцию `yywrap()` таким образом, чтобы она открыла новый входной поток и вернула `0`, что будет сигналом к продолжению работы. Эта функция также удобна, когда необходимо выполнить какие-либо действия по завершении входного потока символов.

`yymore()` — добавляет в `yytext` следующую распознанную цепочку символов. В противоположность стандартной ситуации, когда содержимое `yytext` сбрасывается перед началом распознавания следующей цепочки, функция `yymore()` заставляет анализатор добавить новую распознанную цепочку символов к старой (удлинить распознанную строку).

`yyless(n)` — выделяет `n` символов из `yytext`. Остальные символы из `yytext` возвращаются во входной поток для повторного просмотра.

Если лексический анализатор используется в паре с синтаксическим анализатором и вызывается им для распознавания только очередной лексемы, то его работа прекращается оператором `return` всякий раз, когда очередная лексема распознана.

3.4. Стандартные функции

Для работы лексического анализатора как законченной программы требуется всего одна функция `main()`, стандартный вариант которой имеет вид:

```
#include <stdio.h>

main()
{
    yylex();
}
```

В этом варианте лексический анализатор будет получать входной поток из стандартного ввода и печатать результат на стандартный вывод. Для получения ввода из файла и записи результата работы

в файл функция `main` может переоткрыть `yyin` и `yyout` на заданные файлы. Кроме того, в функции `main()` может быть выполнена инициализация необходимых переменных.

3.5. Примеры простейших программ

Традиционный пример входной программы для LEX — подсчёт числа слов и строк в файле:

```
%{
    int words, lines;
%}
%%
[^, :\.;\n]* { ++words; }
\n           { ++lines; }
.         ;
%%
yywrap() {
  printf("%d слов, %d строк\n", words, lines );
  return(1);
}
main() { words = lines = 0; yylex(); }
```

Ещё пример. Программа для LEX, которая печатает все слова с переносами из входного потока:

```
NODELIM [^, :\.;\n-]
%%
{NODELIM}+"-\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 срабатывать для каждого символа (а не каждого слова).

Наконец, приведём полный текст lex-программы удаления комментариев из программы на языке Си:

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

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

Глава 4
Создание простого калькулятора

В этой главе мы займёмся созданием простого калькулятора, на примере которого изучим основные приёмы практического использования YACC. Мы будем разрабатывать калькулятор, вычисляющий выражение, введённое в строку. Такой подход принципиально отличается от используемого в настоящее время различными разработчиками подобных программ от кустарей-одиночек до крупнейших фирм, при котором основное внимание уделяется созданию шикарного кнопочного поля, позволяющего (так и быть) вручную выполнять отдельные действия над числами. Зачастую даже сам ввод чисел превращается в игру на нарисованном кнопочном поле.

Мы уделим основное внимание вычислительным возможностям калькулятора. Наш калькулятор будет иметь непривлекательный интерфейс командной строки, зато он будет проводить вычисление целых выражений в полностью автоматическом режиме. Начав с простого калькулятора, мы расширим его в последующих главах до возможностей инструментального вычислительного средства широкого профиля.

Наш простейший калькулятор будет работать с числами в десятичной записи и выполнять четыре арифметические операции: `+`, `-`, `*`, `/`, а кроме того, он будет допускать выражения со скобками произвольной глубины вложенности.

4.1. Задание грамматики

Для нашего простейшего калькулятора разделы деклараций и правил уасс-программы, определяющие грамматику входного языка, будут выглядеть так:

```
%{
#define YYSTYPE double
%}

%token DATA
%left '+' '-'
%left '*' '/'
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok wyrag '\n' {printf("\t%g\n",$2);}
  ;
```

```
wyrag : DATA          { $$=$1; }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag { $$=$1/$3; }
  | '(' wyrag ')'   { $$=$2; }
  ;
%%
```

В этом описании раздел деклараций содержит следующую информацию. Поскольку мы предполагаем, что наш калькулятор будет работать с вещественными числами, мы переопределяем стек YACC директивой `#define YYSTYPE double`. Мы будем использовать имя `DATA` для лексем, являющихся числами. Мы также будем использовать литералы `+`, `-`, `*` и `/` в качестве знаков операций. Причём приоритет у `*` и `/` будет выше, чем у `+` и `-`. Все литералы задают левоассоциативные операции, т. е. выражение `5-3-1` будет вычисляться как `(5-3)-1`.

В разделе правил описан собственно синтаксис входного языка калькулятора. Мы видим, что список последовательных выражений (нетерминал `spisok`) определяется как набор строк, оканчивающихся символом (литералом) перевода строки (`\n`). Пустые строки (состоящие только из одного символа перевода строки) также допустимы. Если в строке задано выражение, то его значение должно быть напечатано (точнее, отправлено в файл стандартного вывода).

Выражение (нетерминал `wyrag`) определяется как число (`DATA`), или как пара нетерминалов `wyrag`, объединённых знаком операции, или как нетерминал `wyrag` в скобках.

Действия, описанные в правилах, выполняют вычисление выражения в процессе разбора. Возможен другой подход, когда по результатам разбора записывается программа вычислений на промежуточном языке, которая выполняется после разбора всего выражения. Этот, второй, вариант внутреннего построения калькулятора мы рассмотрим в дальнейших главах.

4.2. Создание лексического анализатора

Для нашего простейшего калькулятора лексический анализатор достаточно прост. Он должен пропускать все символы пробелов и табуляций, а также распознавать цепочки символов, представляющие числа, и переводить их во внутренний формат компьютера. Такой простейший анализатор не стоит программировать на LEX — он получится слишком громоздким. Несложная функция, написанная на языке Си, решит поставленную задачу с минимумом издержек:

```
#include <stdio.h>
#include <ctype.h>

yylex()
{
    int c;

    do{
        c=getchar();
    }while(c==' ' || c=='\t');

    if(c==EOF)
        return 0;
    if(c=='.' || isdigit(c)){
        ungetc(c,stdin);
        scanf("%lf",&yylval);
        return DATA;
    }

    return c;
}
```

Функция имеет имя `yylex()`, так как именно функцию с таким именем вызывает синтаксический анализатор, сгенерированный YACC, для получения очередной лексемы.

Прежде всего функция пропускает все символы пробелов и табуляций.

Если обнаружен конец файла, то функция возвращает код `0`, что является сигналом о конце входного потока для синтаксического анализатора.

Числа распознаются по начальной цифре или десятичной точке. Определение конца числа, как и его значения (перевод во внутреннее представление) возлагается на стандартную функцию языка Си `scanf()`. Значение распознанного числа передаётся синтаксическому анализатору через переменную с предопределённым именем `yylval`, а тип лексемы `DATA` возвращается синтаксическому анализатору через оператор `return`.

Любой другой символ просто передаётся синтаксическому анализатору как литерал. Если этот символ не принадлежит к множеству допустимых грамматикой, то синтаксический анализатор выдаст сообщение об ошибке синтаксиса.

4.3. Описание стандартных функций

Как уже было сказано, для простейших программ, к числу которых можно отнести и наш калькулятор, для окончательной сборки программы можно воспользоваться стандартной библиотекой YACC. Однако эта библиотека не всегда доступна. К тому же для того, чтобы впоследствии расширить функциональные возможности калькулятора, лучше сразу позаботиться об исходных текстах всех его частей. Поэтому для завершения работы над калькулятором нам необходимо задать две стандартные функции. Вот их текст:

```
#include <stdio.h>

main()
{
    yyparse();
}

yyerror(s)
char *s;
{
    fprintf(stderr,"%s\n",s);
}
```

В нашем простейшем случае все три функции (`main()`, `yyerror()` и `yylex()`) проще всего разместить непосредственно в разделе функций yacc-программы. Размещать функции можно в произвольном порядке.

4.4. Трансляция и пробный запуск

Запишем нашу yacc-программу в файл с именем `calc1.y`. Трансляция и сборка рабочей программы в ОС UNIX могут быть выполнены командами:

```
yacc calc1.y
cc -o calc1 y.tab.c
```

В результате получается исполняемая программа с именем `calc1`.

Запуск калькулятора на выполнение осуществляется командой

```
./calc1
```

После запуска можно сразу вводить в строку выражение. Как только будет нажата клавиша `|Enter|`, выражение будет вычислено и его результат напечатан на экране. Если же в выражении будут обнаружены синтаксические ошибки, калькулятор выдаст об этом сообщение и завершит работу:

```
4*3*2
    24
(1+2) * (3+4)
    21
1/2
    0.5
355/113
    3.14159
-3-4
syntax error
```

4.5. Возможности созданного калькулятора

Даже такой созданный нами примитивный калькулятор обладает рядом положительных возможностей, отсутствующих у его «экранных» собратьев.

Прежде всего, это возможность записывать целые выражения и не следить за порядком вычислений.

Вторая возможность следует из того, что наш калькулятор получает входной поток из стандартного потока ввода. Это даёт возможность передать на вход калькулятора файл с заранее записанными выражениями. Выражения будут вычислены и результаты напечатаны на экране. Команда будет выглядеть так:

```
calc1 <infile
```

где `infile` — файл с заранее записанными выражениями, которые надо вычислить.

Наконец, поскольку печать на экран выполняется через стандартный поток вывода, можно перенаправить его в файл на диске и сохранить результаты вычислений. Например:

```
calc1 >ofile
```

В этом случае результаты вычислений сохраняются в файле `ofile` и могут быть в дальнейшем использованы для анализа результатов.

Легко представить себе пакетный режим использования калькулятора:

```
calc1 <infile >ofile
```

Где можно так его использовать? Предположим, что Вы хотите получить таблицу значений некоторой функции. Вы можете записать в файл `infile` исходное выражение для вычисления первой точки, затем средствами текстового редактора быстро размножить строку с этим выражением и изменить в полученных строках только то значение, по которому нужно табулировать функцию. Далее Вам нужно выполнить приведённую выше команду — и таблица значений у Вас в файле `ofile`.

Эту же задачу можно решить ещё проще, используя наш калькулятор в качестве фильтра:

```
программа-генератор_выражений | calc1 | программа-табулятор
```

Для этого необходимо, вместо создания файла с выражениями вручную, написать программу, генерирующую эти выражения, на любом доступном языке (perl, awk или даже shell). Генератор должен выдавать выражения в стандартный поток вывода. На любом из указанных языков размер программы не превысит пяти строк.

Задачей программы-табулятора будет формирование итоговой таблицы. В простейшем случае вместо такой программы можно, как и в предыдущем случае, перенаправить результаты в файл на диске.

В качестве конкретного примера предположим, что нам надо получить квадраты первых 100 натуральных чисел. В качестве языка для создания программы-генератора выражений будем использовать язык оболочки (shell). Программа генерации выражений для shell будет иметь вид:

```
#!/bin/sh
for i in `seq 0 100`
do
    echo $i \* $i
done
```

Запишем эту программу в файл с именем `g.sh` и дадим команду

```
sh g.sh | ./calc1
```

Простенько и со вкусом!

4.6. Полный текст простейшего калькулятора

В заключение приведём полный текст файла `calc1.y`, содержащего программу простейшего калькулятора. В последующих главах для экономии места в пособии полные тексты разрабатываемых программ приводить не будем. Вместо этого в приложении 1 приведём полные тексты наиболее интересных из разработанных программ.

```
%{
#define YYSTYPE double
%}

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

```
spisok :    /* пусто */
  | spisok '\n'
  | spisok wyrag '\n' {printf("\t%g\n",$2);}
  ;
wyrag : DATA        { $$=$1; }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag { $$=$1/$3; }
  | '(' wyrag ')'   { $$=$2; }
  ;
%%

#include <stdio.h>
#include <ctype.h>

main()
{
    yyparse();
}

yyerror(s)
char *s;
{
    fprintf(stderr,"%s\n",s);
}

yylex()
{
    int c;
    do{
        c=getchar();
    }while(c==' ' || c=='\t');
    if(c==EOF)
        return 0;
    if(c=='.' || isdigit(c)){
        ungetc(c,stdin);
        scanf("%lf",&yylval);
        return DATA;
    }
    return c;
}
```

Глава 5
Простейшие доработки калькулятора

Выше уже была высказана мысль, что инструментальные средства программиста облегчают разработку сложных языковых программ. Подтвердим эту мысль на практике. В этой главе мы проведём некоторые доработки нашего простейшего калькулятора. Прежде всего добавим операции унарного минуса и возведения в степень. Затем уделим внимание обработке ошибок. Наконец, введём возможность пользоваться ячейками памяти.

5.1. Операции унарного минуса и возведения в степень

Для решения поставленной задачи не потребуются какие-либо специальные разработки на языке Си. Операция унарного минуса является стандартной в синтаксисе Си, а операция возведения в степень может быть выполнена с помощью функции `pow()`, входящей в стандартную математическую библиотеку Си. Поэтому введение операций унарного минуса и возведения в степень потребует добавления всего нескольких строк в текст yacc-программы.

Раздел деклараций будет выглядеть так:

```
%{
#include <math.h>

#define YYSTYPE double
%}

%token DATA
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%
```

Мы предполагаем здесь, что операция возведения в степень (`^`) имеет приоритет выше, чем операции умножения и деления, но меньше, чем операция унарного минуса (которая имеет наивысший приоритет). Причём для операции возведения в степень задана левая ассоциативность, т. е. выражение вида `a^b^c` будет вычислено как `(a^b)^c` или, записывая в принятой математической нотации, $(a^b)^c$.

Обратите внимание, что `UNARYMINUS` задан как некоторый терминальный символ, который реально никогда не будет возвращён лексическим анализатором. Он нужен лишь для задания положения в иерархии приоритетов операций.

В разделе правил добавляется два новых правила для нетерминала wyrag:

```
wyrag : DATA           { $$=$1; }
   ...
   | wyrag '^' wyrag     { $$=pow($1,$3); }
   | '-' wyrag %prec UNARYMINUS { $$=-$2; }
   ;
```

В каком именно порядке добавлять правила — совершенно безразлично. Оператор %prec показывает, что приоритет унарного минуса должен быть таким же, как у UNARYMINUS. Приоритет операции вычитания устанавливается обычным образом. Для возведения в степень используется функция pow(), прототип которой описан в файле math.h. Поэтому в раздел деклараций включена строка

```
#include <math.h>
```

Каких-либо других исправлений на данном этапе расширения возможностей калькулятора не требуется. Наша функция лексического анализатора написана таким образом, что пропускает через себя на вход синтаксического анализатора все символы, не являющиеся числами, (в том числе и те, которые не являются частью грамматики). Поэтому символ ^, введённый теперь в грамматику, она также пропустит. В стандартных функциях также не требуется никаких изменений.

Новый вариант программы запишем в файл с именем calc1a.y. При трансляции и сборке программы надо указать компилятору Си необходимость использовать математическую библиотеку:

```
yacc calc1a.y
cc -o calc1a -lm y.tab.c
```

Запустим программу (командой ./calc1a) и проверим работу новых функций:

```
-3-2
    -5
5^2
    25
2^(1/2)
    1.41421
```

5.2. Обработка ошибок

Как Вы уже могли видеть во время проверки работы калькулятора, его реакция на ошибки весьма проста — краткое сообщение по-английски и завершение работы. Если при этом Вы обрабатываете файл с предварительно записанными в него выражениями, то понять, какое именно выражение оказалось ошибочным, достаточно сложно.

Мы рассмотрим здесь некоторые меры, с помощью которых можно, во-первых, локализовать место появления ошибки для последующего анализа, а во-вторых, парировать возникшие ошибки с тем, чтобы продолжить работу без повторного запуска калькулятора. Заметим, что возможность продолжения работы без перезапуска будет важна для последующих версий калькулятора, которые получат возможность сохранения результатов вычислений в памяти.

Мы рассмотрим обработку трёх типов ошибок: синтаксическая (ошибка в записи выражения), деление на ноль и переполнение вещественного.

5.2.1. Общие принципы обработки ошибок

Прежде всего условимся о форме сообщения об ошибке. Для нас важно, чтобы сообщения об ошибках были осмысленны и понятны пользователю. Поэтому желательно, чтобы пользователь получал их на родном языке, в нашем случае на русском. Кроме того, желательно, чтобы сообщения были лаконичными. Если пользователю что-либо непонятно в тексте сообщения, он может обратиться к документации, где смысл сообщения разъяснён более подробно. Однако искать разъяснение сообщения в документации гораздо удобнее по цифровым кодам (которые могут быть упорядочены), чем по тексту. Из этого, в частности, следует необходимость в нумерации всех типов сообщений об ошибках. Наконец, не все текстовые сообщения, выдаваемые калькулятором, могут быть сообщениями об ошибках. Поэтому текст сообщения об ошибке должен начинаться со слова «Ошибка».

При локализации места ошибки очень полезной оказывается информация о номере строки, в которой эта ошибка обнаружена. В случае нашего калькулятора этой информации будет вполне достаточно.

Правилом хорошего тона в программировании принято считать такое построение программы, при котором все сообщения об ошибках выдаются одной функцией, обеспечивающей единство формы сообщения и последующих действий программы по восстановлению. Тексты сообщений при этом размещаются в массиве строк, из которых они выбираются по коду сообщения, задаваемому в месте

обнаружения ошибки. Коды сообщений при этом могут выдаваться вместе с текстом сообщений и быть задокументированы.

Такой подход к разработке программы позволяет при необходимости легко перевести сообщения об ошибках на любой разговорный язык, не прибегая к сканированию текста программы в поисках таких сообщений.

Наша функция выдачи сообщения об ошибке может выглядеть следующим образом:

```
char *ms_calc_error[]={
/* 0 */ "Синтаксическая ошибка",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного"
};

calc_error(num)
int num;
{
    fprintf(stderr,"\nОшибка! Строка %u\n(%02d) %s\n",
                        nline,num,ms_calc_error[num]);
    ... /* операции по восстановлению */
}
```

Функция `calc_error()` получает на вход код сообщения об ошибке. Номер текущей строки содержится в глобальной переменной `nline`. Возможные коды ошибок (в виде комментариев) и соответствующие им сообщения сведены в массив строк `ms_calc_error`. При необходимости в этот массив могут быть добавлены и другие сообщения об ошибках со своими кодами. Работа функции заключается в выдаче сообщения об ошибке в стандартный поток диагностики и выполнении некоторых действий по восстановлению, согласованных с остальными частями программы калькулятора. Сообщение об ошибке содержит код ошибки, текстовое сообщение и номер строки, в которой ошибка обнаружена. Например, при обнаружении синтаксической ошибки сообщение может иметь вид

```
Ошибка! Строка 5
(00) Синтаксическая ошибка
```

Описанную функцию выдачи сообщения об ошибке должны вызывать все функции, причастные к обработке ошибок.

Подсчёт числа строк будем выполнять в функции лексического анализатора. Для этого потребуется всего три строки кода:

```
unsigned int nline=1;  /* начальное значение счётчика */
...
yylex()
```

```
{
    int c;
    ...
    if(c=='\n')
        nline++;
    ...
}
```

Наилучшими действиями по восстановлению работы калькулятора в нашем случае будет перевод синтаксического анализатора в начальное состояние. Это достаточно просто выполнить с помощью пары функций `setjmp()` и `longjmp()`. Их точный синтаксис можно посмотреть в руководстве по библиотечным функциям Си. Мы приведём здесь только готовый пример их использования:

```
#include <setjmp.h>

jmp_buf jmp_env;

main()
{
    if(setjmp(jmp_env))
        ;
    yyparse();
}

...

calc_error(num)
int num;
{
    ...
    longjmp(jmp_env,1);
}
```

Функция `setjmp()` возвращает 0 при первом вызове и не 0, если в эту точку осуществлён переход (в данном случае по ошибке) функцией `longjmp()`. В последнем случае программист может предусмотреть некоторые действия, что и показано оператором `if()`. В нашем примере никакие дополнительные действия не выполняются.

5.2.2. Организация обработки конкретных ошибок

При обработке ошибок будем использовать коды ошибок, приведённые в примерах, рассмотренных в предыдущем пункте.

Проще всего обработать ошибку деления на ноль. Эта обработка задаётся в действии к правилу, описывающему деление:

```
wyrag : DATA           { $$=$1; }
   ...
   | wyrag '/' wyrag     {
                  if($3==0.)
                      calc_error(1);
                  $$=$1/$3;
             }
```

Для обработки переполнения вещественного воспользуемся системой программных сигналов. Нам необходимо перехватить сигнал «переполнение вещественного», который возникает при переполнении вещественного числа. Для этого разработаем функцию `fpeerr()`, которая должна будет обрабатывать этот сигнал, и установим её в функции `main()` как функцию обработки заданного сигнала.

Заметим, что в системе программирования Си ОС UNIX и в системе программирования gcc ОС Linux правила записи конструкции перехвата сигнала несколько отличаются.

Для ОС UNIX имеем следующую запись:

```
#include <signal.h>
int fpeerr();

main()
{
    signal(SIGFPE,fpeerr);
    yyparse();
}

fpeerr()
{
    calc_error(2);
}
```

Для ОС Linux запись будет другая:

```
#include <signal.h>
void fpeerr();

main()
{
```

```
    signal(SIGFPE,fpeerr);
    yyparse();
}

void fpeerr()
{
    calc_error(2);
}
```

Обработка синтаксических ошибок оказывается не таким простым делом, каким может показаться на первый взгляд. Простая правка функции `yyerror()` типа

```
yyerror()
{
    calc_error(0);
}
```

приведёт к тому, что мы будем получать сообщения о синтаксической ошибке на каждый неверно введённый в строку символ*. Более правильным решением является сброс оставшейся входной строки и перевод анализатора в нормальное состояние. К сожалению, для осуществления этой задачи придётся несколько отойти от принципа единства функции выдачи сообщений и повторить процедуру выдачи сообщения в функции `yyerror()`:

```
yyerror()
{
    fprintf(stderr,"\
    \nОшибка! Строка %u\n(00) Синтаксическая ошибка\n",
                        nline);
}
```

Обратите внимание, что здесь нет функции `longjmp()`. Эта функция не нужна, т. к. вся процедура восстановления записывается в разделе

---

* Строго говоря, функция `yyerror()` является функцией, вызываемой синтаксическим анализатором в случае любой своей ошибки, а не только синтаксической. Правильный формат её вызова — `yyerror(s)`, где `s` является указателем на строку сообщения об ошибке (по-английски). Однако анализ текста анализатора показывает, что возможно всего две ошибки — «синтаксическая ошибка» и «переполнение стека анализатора». Вторая ошибка настолько маловероятна, что для большинства прикладных программ нет причин в разработке специализированной процедуры восстановления. Поэтому, пользуясь особенностями внутренней реализации механизма передачи параметров функций Си, а также учитывая, что мы выдадим своё сообщение на русском языке, параметр `s` можно в аргументах функции `yyerror()` не указывать.

синтаксических правил yacc-программы. Для этого введём специальное правило с ключевым словом `error` и зададим для него действия по восстановлению:

```
spisok :      /* пусто */
  ...
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
```

Функция `resynch()` обеспечивает пропуск всех символов до начала новой строки. Вот её текст:

```
resynch()
{
    int c;

    do{
        c=getchar();
    }while(c!='\n');

    ungetc(c,stdin);
}
```

Произведя необходимые изменения, запишем новую yacc-программу в файл `calc1b.y` и выполним её трансляцию. После запуска готовой программы проверим её реакцию на предусмотренные ошибки:

```
5+2
    7
qwerty

Ошибка! Строка 2
(00) Синтаксическая ошибка
4/0

Ошибка! Строка 4
(01) Деление на 0
1e1000*1e1000

Ошибка! Строка 5
(02) Переполнение вещественного
```

Надо заметить, что наш счётчик строк не всегда выдаёт достоверную информацию. Это связано с особенностями построения лексического анализатора и его взаимодействия с синтаксическим анализатором. Для устранения этого недостатка функция лексического анализатора должна быть доработана так, чтобы из стандартного потока ввода в промежуточный буфер читалась сразу строка, а затем уже из строки осуществлялись выборка и анализ символов.

5.3. Работа с регистровыми переменными

Добавим в наш калькулятор ещё одну чрезвычайно полезную функцию — возможность сохранения в памяти и последующего использования результатов вычисления выражений. Будем считать, что у нас есть 26 ячеек памяти с именами от `a` до `z`. Будем называть такие ячейки памяти регистрами. Для работы с регистрами мы должны ввести в наш калькулятор операцию присваивания значения выражения регистру и операцию извлечения значения из регистра. Для удобства использования разрешим также операцию множественного присваивания, например:

```
x = y = z = 0
```

Несмотря на значительный шаг вперёд в функциональной мощности калькулятора, для реализации предполагаемых новых возможностей нам потребуются весьма небольшие изменения в разделах деклараций и правил yacc-программы и в функции лексического анализатора.

Прежде всего заметим, что для хранения значений регистровых переменных очень удобен обычный массив чисел. Однобуквенное имя регистра в этом случае легко преобразуется к индексу ячейки памяти в массиве.

Поскольку синтаксический анализатор теперь будет работать не только с вещественными числами, но и с целочисленными индексами массива регистров, нам придётся отказаться от описания типа стека анализатора как `double` и описать его как объединение `%union` вещественного и целого типов. Кроме того, поскольку в этом случае YACC будет выполнять контроль типов, нам придётся описать типы используемых в выражениях лексем и нетерминалов. Наконец, для описания типа регистровой переменной мы введём ещё один тип лексемы — `VAR`. Далее приведём разделы деклараций и правил yacc-программы с внесёнными изменениями:

```
%{
#include <math.h>

float mem[26];  /* массив регистров */
```

```
%}

%union{
double data;    /* числовое значение */
int index;  /* индекс в~массиве регистров */
}

%token <data> DATA
%token <index> VAR
%type <data> wyrag
%right '='
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok wyrag '\n'   { printf("\t%g\n",$2); }
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
wyrag : DATA        { $$=$1; }
  | VAR         { $$=mem[$1]; }
  | VAR '=' wyrag   { $$=mem[$1]=$3; }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag {
                if($3==0.)
                    calc_error(1);
                $$=$1/$3;
            }
  | '(' wyrag ')'   { $$=$2; }
  | wyrag '^' wyrag { $$=pow($1,$3); }
  | '-' wyrag %prec UNARYMINUS { $$=-$2; }
  ;
```

```
%%
```

В функцию лексического анализатора необходимо внести небольшие дополнения, связанные с необходимостью распознавать имена регистровых переменных и возвращать новый тип лексемы, а также изменить заполнение переменной `yylval`, поскольку стек синтаксического анализатора является теперь объединением:

```
yylex()
{
    int c;

    do{
        c=getchar();
    }while(c==' ' || c=='\t');

    if(c==EOF)
        return 0;
    if(c=='.' || isdigit(c)){
        ungetc(c,stdin);
        scanf("%lf",&yylval.data);
        return DATA;
    }

    if(islower(c)){
        yylval.index=c-'a';
        return VAR;
    }

    if(c=='\n')
        nline++;

    return c;
}
```

Запишем новую программу в файл `calc2.y`, выполним её трансляцию и проверим новые возможности:

```
a=4
    4
b=5
    5
a^b
    1024
```

Обратите внимание, что в нашей программе калькулятора начальные значения регистровых переменных не определены. В начальный момент работы они содержат произвольные значения, т. е. «мусор». Выход из этой ситуации может быть двояким. С одной стороны, мы можем принудительно задавать всем регистровым переменным какое-нибудь известное пользователю начальное значение, например ноль. С другой стороны, мы можем просто вести контроль за тем, каким регистрам пользователь уже присваивал значения, а каким нет, и выдавать сообщение об ошибке при попытке использовать в выражении значение регистра, которому ещё ничего не присвоено.

Глава 6
Утилита MAKE для автоматизации работ в системе

6.1. Деление программы на логические части

Наша программа постепенно увеличивается в размерах. Держать её исходный текст в одном файле становится всё более неудобно. Файл становится труднообозримым. При необходимости каких-либо изменений приходится долго искать нужное место в файле.

Эта проблема потенциально гораздо опаснее, чем может показаться на первый взгляд. Хранение всей программы в одном файле, во-первых, повышает опасность порчи её текста из-за случайных сбоев системы или ошибок программиста. Во-вторых, рост размеров программы может привести к невозможности редактировать её в простых текстовых редакторах, имеющих ограничения на размер редактируемого файла, а использование «больших» редакторов, обладающих соответственно и бóльшим временем загрузки, заметно снизит эффективность работы программиста при отладке программы. В-третьих, при любых незначительных исправлениях в исходном тексте будет требоваться перетрансляция всей программы, что опять же ведёт к неэффективному использованию рабочего времени программиста, да и машинного времени тоже. Можно назвать и ряд других причин, по которым хранение исходного текста всей программы в одном файле нежелательно.

Опытный программист стремится разделить текст своей программы на ряд файлов, содержащих законченные (самостоятельные) или по крайней мере логически связанные части программы. Это позволяет, во-первых, упростить поиск редактируемых фрагметов, во-вторых, транслировать только те части программы, в которых были сделаны изменения, в-третьих, значительно снижает риск потери текста программы (по крайней мере бóльшей его части) из-за сбоев системы.

В случае нашего калькулятора мы можем разделить его исходный текст на три логически законченных части: модуль синтаксического анализатора, модуль лексического анализатора и служебный модуль.

Модуль синтаксического анализатора должен содержать только разделы деклараций и правил исходной yacc-программы, то есть те её части, из которых и порождается собственно синтаксический анализатор. В дальнейшем именно этот модуль мы будем называть yacc-программой калькулятора. Запишем его в файл `syntax.y`:

```
%{
#include <math.h>

float mem[26];  /* массив регистров */
%}

%union{
double data;    /* числовое значение */
int index;  /* индекс в~массиве регистров */
}

%token <data> DATA
%token <index> VAR
%type <data> wyrag
%right '='
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok wyrag '\n'   { printf("\t%g\n",$2); }
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
wyrag : DATA        { $$=$1; }
  | VAR         { $$=mem[$1]; }
  | VAR '=' wyrag   { $$=mem[$1]=$3; }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag {
                if($3==0.)
                    calc_error(1);
                $$=$1/$3;
            }
```

```
  | '(' wyrag ')'    { $$=$2; }
  | wyrag '^' wyrag { $$=pow($1,$3); }
  | '-' wyrag %prec UNARYMINUS { $$=-$2; }
  ;
%%
```

Модуль лексического анализатора будет содержать функции `yylex()` и `resynch()`, обеспечивающие лексический анализ входного потока. Запишем его в файл `lexical.c`. Обратите внимание, что в файле присутствует директива `#include "y.tab.h"`, требующая подключить файл с описанием типов лексем и типа стека синтаксического анализатора, без которых связь между лексическим и синтаксическим анализаторами будет невозможна. Файл `y.tab.h` мы будем получать автоматически в процессе трансляции yacc-программы.

```
#include <stdio.h>
#include <ctype.h>
#include "y.tab.h"

extern unsigned int nline;

yylex()
{
    int c;

    do{
        c=getchar();
    }while(c==' ' || c=='\t');

    if(c==EOF)
        return 0;
    if(c=='.' || isdigit(c)){
        ungetc(c,stdin);
        scanf("%lf",&yylval.data);
        return DATA;
    }

    if(islower(c)){
        yylval.index=c-'a';
        return VAR;
    }

    if(c=='\n')
```

```
        nline++;

    return c;
}

resynch()
{
    int c;

    do{
        c=getchar();
    }while(c!='\n');

    ungetc(c,stdin);
}
```

Наконец, служебный модуль будет содержать все остальные функции, обеспечивающие работу калькулятора — головную функцию `main()`, а также функции, обеспечивающие обработку ошибок. Попутно условимся считать (исключительно в рамках данного учебного пособия), что имя итоговой рабочей программы калькулятора соответствует имени этого файла. На данный момент последняя версия нашего калькулятора называется `calc2a`. Поэтому и файл служебного модуля будет иметь имя `cacl2a.c`:

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

void fpeerr();

jmp_buf jmp_env;

unsigned int nline=1;   /* счётчик строк */

main()
{
    signal(SIGFPE,fpeerr);
    if(setjmp(jmp_env));
    yyparse();
}
```

```
void fpeerr()
{
    calc_error(2);
}

yyerror()
{
    fprintf(stderr,"\
    \nОшибка! Строка %u\n(00) Синтаксическая ошибка\n",
                        nline);
}

char *ms_calc_error[]={
/* 0 */ "",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного"
};

calc_error(num)
int num;
{
    fprintf(stderr,"\nОшибка! Строка %u\n(%02d) %s\n",
                        nline,num,ms_calc_error[num]);
    longjmp(jmp_env,1);
}
```

Обратите внимание, что, благодаря разделению текста программы на отдельные файлы, удаётся ограничить область действия глобальных переменных пределами только тех файлов, где они определены. Для переменной `nline`, подсчитывающей число входных строк, пришлось сделать дополнительное описание `extern` в файле `lexical.c`, чтобы сделать её доступной и в функции лексического анализатора, и в функциях печати сообщений об ошибках.

6.2. Проблемы ручной трансляции

Разделив исходный текст программы на несколько файлов, мы значительно облегчили себе задачу внесения изменений, но попутно создали несколько проблем. Как теперь выполнять трансляцию и сборку программы? Рассмотрим возникающие проблемы на примере работы в ОС UNIX.

Для получения рабочей программы калькулятора из исходного текста, записанного в один файл, нам надо было дать две команды:

```
yacc calc2a.y
cc -o calc2a y.tab.c
```

Если бы при отладке программы нам надоело постоянно вводить эти две команды, то можно было бы записать их в командный файл и запускать его на выполнение всякий раз, когда нам потребовалось бы получить рабочую программу из её исходного текста.

Если же исходный текст программы записан в нескольких файлах, то число команд в процедуре трансляции заметно увеличивается. В нашем случае (три файла) необходимо выполнить следующие команды:

```
yacc -d syntax.y
cc -c y.tab.c
cc -c lexical.c
cc -c calc2a.c
cc -o calc2a calc2a.o lexical.o y.tab.o
```

Здесь файлы `.o` — это объектные файлы, полученные в результате трансляции соответствующих файлов `.c`. Рабочая программа собирается редактором связей именно из объектных файлов.

В этом примере есть одна неточность. На самом деле, все команды `cc` можно заменить на одну, поскольку команда `cc` это допускает:

```
cc -o calc2a calc2a.c lexical.c y.tab.c
```

В итоге получим, на первый взгляд, те же две команды:

```
yacc -d syntax.y
cc -o calc2a calc2a.c lexical.c y.tab.c
```

Но оказывается, что всё не так просто. Предположим, что в процессе отладки калькулятора, когда рабочая программа уже несколько раз была получена, мы внесли изменения в файл `calc2a.c`. Поскольку в другие файлы исправления не вносились, то мы можем значительно сократить время получения новой рабочей программы. Для этого мы должны оттранслировать только файл `calc2a.c`, а для остальных файлов использовать готовые объектные модули:

```
cc -c calc2a.c
```

```
cc -o calc2a calc2a.o lexical.o y.tab.o
```
или, сокращая запись,
```
cc -o calc2a calc2a.c lexical.o y.tab.o
```

На этом пути нас, однако, подстерегает одна проблема. Если мы внесли изменения не в `.c`, а в `.h` или в `.y` файл, то нам может потребоваться перетрансляция всех `.c` файлов, в которых используется данный файл `.h` или файл `y.tab.h`, порождаемый командой `yacc -d`. Например, если мы внесём изменения в файл `syntax.y`, то нам необходимо будет перетранслировать как сам этот файл, так и файл `lexical.c`, причём в определённом порядке — сначала `syntax.y`, потом `lexical.c`, поскольку иначе файл `lexical.c` будет оттранслирован до генерации новой версии файла `y.tab.h`.

Мы пришли к понятию **зависимости** файлов друг от друга. Дерево зависимостей файлов для нашей программы калькулятора приведено на рис. 6.1. Здесь исходные файлы программы подчёркнуты. Трансляцию необходимо выполнять в порядке строго снизу вверх.

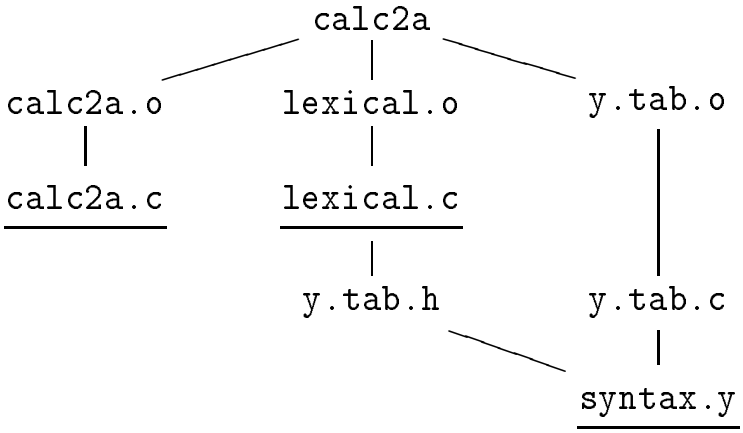

Рис. 6.1. Дерево зависимостей файлов программы калькулятора

Для любой программы, исходный текст которой записан в нескольких файлах, довольно сложно удержать в памяти все зависимости. Зато легко допустить ошибку, забыв перетранслировать какой-либо файл. Выполнять же каждый раз полную перетрансляцию программы крайне невыгодно, особенно при большом числе исходных файлов. Тем более, что это не может гарантировать от ошибок, вызванных неправильной последовательностью трансляции.

Описанные проблемы позволяет решить утилита `make`, специально предназначенная для выполнения работ с зависимыми файлами.

6.3. Описание утилиты MAKE

Утилита `make` предоставляет уникальные возможности по управлению любыми видами работ (а не только трансляцией и сборкой программ) в операционной системе. Всюду, где имеется необходимость учитывать зависимости файлов и времена их создания (модификации), `make` оказывается незаменимым инструментом. Утилита реализует непроцедурный язык, который позволяет управлять группами командных строк системы. В основу такого управления положены зависимости между файлами с исходными данными и файлами, в которых содержатся результаты. При этом предполагается любой возможный список действий над исходными файлами: компиляция, макрообработка, редактирование, печать, упаковка, шифрование и т. п. Исходной информацией для утилиты является make-программа, представляющая список определений макропараметров и список правил. Каждое правило включает формулировку цели и список действий для командного интерпретатора (список команд системы для достижения цели). При выполнении make-программы утилита `make` использует информацию о связях между целями и результатами и передаёт на выполнение списки действий, которые в данный момент необходимо выполнить для получения заданного результата. Программист, владеющий средствами `make`, использует технологию разработки программного комплекса, независимо от его сложности, схематически показанную на рис. 6.2.

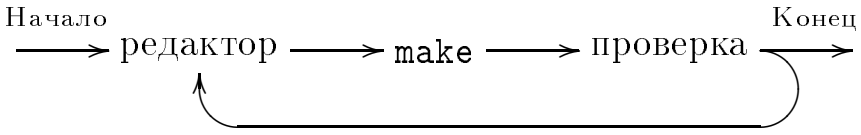

Рис. 6.2. Технология разработки программного комплекса при использовании утилиты `make`

Идея работы `make` заключается в сравнении времён создания зависимых файлов. Если файлы цели имеют время создания более раннее, чем файлы, от которых они зависят, значит, в последних были сделаны какие-то исправления, и требуется выполнить повторное построение файлов цели. Если же в процессе сравнения выяснится, что все файлы, от которых зависят файлы цели, старше файлов целей, то повторное построение файлов целей проводить не нужно.

Входной язык утилиты `make` ОС UNIX достаточно сложен для того, чтобы подробно рассматривать его в настоящем учебном

пособии. Мы рассмотрим только те основные его элементы, которые нам понадобятся для автоматизации трансляции и сборки нашей рабочей программы.

Программа на языке утилиты `make` называется make-программой и хранится в make-файле. В принципе, этот файл может иметь любое имя, но по умолчанию утилита всегда ищет файл с именем `makefile` в текущем подкаталоге. Если этот файл не найден, то по умолчанию предпринимается попытка найти файлы с некоторыми другими стандартными именами. Мы не будем здесь описывать полный алгоритм поиска make-файла, а просто будем считать, что наша make-программа всегда находится в файле `makefile` текущего подкаталога.

Основной частью make-программы являются правила. Каждое правило состоит из двух частей — строки описания зависимости файлов (как файлов цели от исходных файлов) и списка команд системы, которые необходимо выполнить, чтобы получить файлы целей из исходных файлов.

Строка зависимостей всегда начинается с первой позиции строки. Файлы целей отделяются в ней от исходных файлов двоеточием. Каждая строка списка команд записывается в отдельной строке и с отступом от начала строки хотя бы на один символ табуляции. Например, можно было бы так описать правило получения объектного модуля из исходного файла с программой на Си:

```
file.o : file.c
    cc -c file.c
```

За одной строкой зависимостей может следовать произвольное число строк команд. В том числе допускается записывать правила зависимостей вообще без команд.

Длинные строки можно записывать с продолжением на следующей строке. В этом случае строка, которая будет продолжена, должна оканчиваться символом \ (обратного слеша).

Возможно задание правил, содержащих только цель, и не содержащих исходных файлов.

Правила в make-файле записываются обычно в порядке «от целей к исходным файлам» сверху вниз. Хотя вообще порядок записи правил не влияет на работу make-программы.

Кроме задания правил с зависимостями и действиями для конкретных файлов (явных правил), могут быть заданы и неявные правила. В этом случае правила записываются с зависимостями, содержащими только расширения файлов (в обратном порядке). Зависимости сопровождаются списком команд системы, обеспечивающими получение файлов с расширениями целей из файлов с расширениями источников. Для обеспечения указания в командных строках

имён обрабатываемых файлов применяются встроенные макропараметры. Например, неявное правило трансляции любого файла `.c` может иметь вид:

```
.c.o :
    cc -c $<
```

где `$<` — встроенный макропараметр, означающий «имя исходного файла». Имея это правило, при описании зависимостей объектных файлов от файлов с их исходным текстом на Си мы можем не указывать командные строки — `make` будет пользоваться для получения объектных файлов неявным правилом.

В make-программе мы можем задавать и использовать макропараметры. Задание макропараметра выполняется записью вида

```
ИМЯ = значение
```

Обращение к заданному таким способом макропараметру выполняется оператором вида

```
$(ИМЯ)
```

Отметим, что существует стандартный набор предопределённых макропараметров, описывающих расположение в системе, имена и режимы работы по умолчанию основных системных компонентов. Значение этих макропараметров можно менять. Существует также возможность задания макрокоманд. Но здесь эта возможность рассмотрена не будет, поскольку для наших целей она не потребуется.

Утилита `make` ОС UNIX имеет набор встроенных неявных правил, значительно облегчающих создание make-программ в случае использования стандартных средств системы. Граф зависимостей целей, обеспечиваемый неявными правилами, приведён на рис. 6.3.

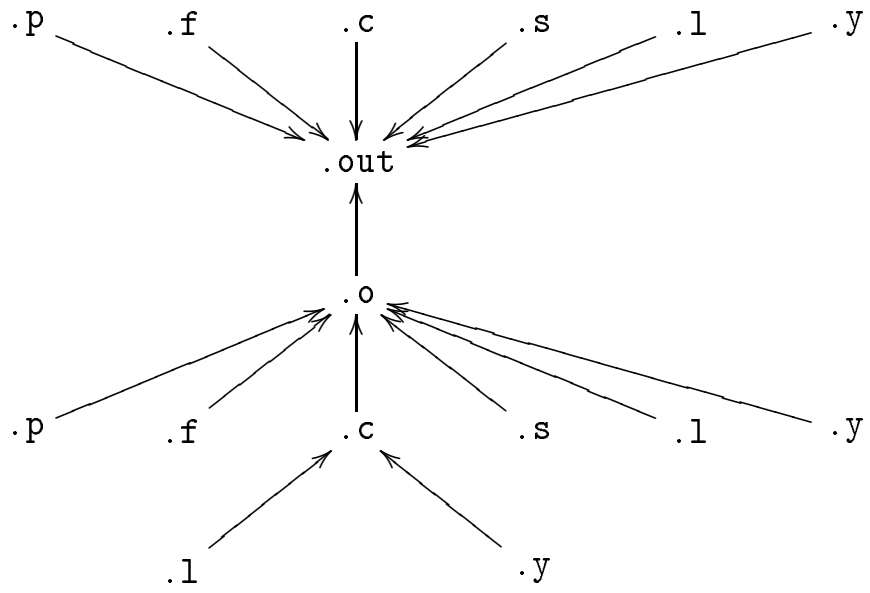

Рис. 6.3. Граф зависимостей целей `make`, обеспечиваемых неявными правилами по умолчанию

По поводу графа заметим, что при получении из файла `.y` файла `.c` или `.o` обеспечивается сохранение основной части имени файла. Например, для нашего случая из файла `syntax.y` будет получен файл `syntax.c` или `syntax.o` (в зависимости от заданной цели), а не `y.tab.c` или `y.tab.o`. Программист, однако, должен помнить, что если задан режим получения файла `y.tab.h`, то этот файл не переименовывается.

Запуск на выполнение make-программы в текущем подкаталоге выполняется командой

```
make
```

или

```
make цель
```

В первом случае целью работы будет считаться цель, записанная в make-файле самой первой. Во втором случае цель указана явно. Такая цель, указываемая явно, может не зависеть ни от каких файлов и использоваться для автоматизированного выполнения команд, не связанных непосредственно с трансляцией программы.

6.4. make-программа трансляции и сборки калькулятора

Для ОС UNIX make-программа предельно проста. В ней записываются только правила и зависимости, отличные от заданных по умолчанию:

```
PROG = calc2a
YFLAGS = -d

$(PROG) : $(PROG).o syntax.o lexical.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o

lexical.o : y.tab.h
```

Здесь макропараметр `YFLAGS` задаёт режим работы генератора `yacc`, при котором необходимо создавать файл `y.tab.h`. Показана также зависимость объектного файла `lexical.o` от файла `y.tab.h`.

Макропараметр `PROG` используется здесь для упрощения модификации make-программы под новые версии калькулятора, поскольку мы договорились, что имя рабочей программы совпадает с именем файла служебного модуля.

Обратите внимание, что в правилах указаны лишь самые необходимые зависимости и действия для сборки программы. Утилита `make` сама обнаружит необходимые исходные файлы с расширениями `.c` и `.y`.

Команда `cc` выполняет в данном случае вызов редактора связей с передачей ему всех необходимых для сборки программы библиотек объектных модулей. Ключ `-lm` указывает на необходимость дополнительного подключения математической библиотеки.

Запуск make-программы на выполнение осуществляется простой командой

```
make
```

В результате будет получена рабочая программа калькулятора.

6.5. Полезное дополнение make-программы

Как уже отмечалось, утилита `make` может помочь не только при получении рабочей программы из её исходного текста. Предположим, что программист в конце каждого рабочего дня архивирует результат работы, пользуясь утилитой сжатия `zip`. Для уменьшения размеров архива желательно включать в него только файлы с исходным текстом программы, удаляя предварительно все промежуточные. После создания архива полезно убрать с диска и все рабочие файлы.

Программист может добиться желаемой цели автоматически, если поместит в make-программу правило с независимой целью. Например, это правило может выглядеть так:

```
zip:
    -del *.bak
    -del *.o
    -del y.tab.*
    -zip -9 -u $(PROG) *.c *.h *.y makefile
    -del *.c
    -del *.h
    -del *.y
    -del makefile
```

Правило запускается в работу командой

```
make zip
```

После его работы в текущем подкаталоге остаются только файл архива и последний вариант рабочей программы.

Глава 7
Построение лексического анализатора с использованием LEX

7.1. Обоснование необходимости применения LEX

Наш лексический анализатор, написанный непосредственно на Си, достаточно прост, если не сказать примитивен. Расширяя возможности нашего калькулятора, мы должны были бы периодически наращивать возможности и лексического анализатора. Причём, в силу простоты последнего, эту процедуру ещё довольно долго можно выполнять вручную, не прибегая к сложным средствам автоматической генерации. При этом, во-первых, мы будем избавлены от необходимости иметь такие средства автоматической генерации, а, во-вторых, лексические анализаторы, построенные таким способом, отличаются малыми размерами и большой скоростью работы.

Однако на достаточно простых примерах можно показать, что такой подход не всегда оказывается успешным. Например, в нашем лексическом анализаторе выделение из входного потока подстроки, являющейся числом (тип лексемы `DATA`), возложен на стандартную функцию `scanf`. Эта функция старается прочитать из входного потока такое число, которое соответствует типу, заданному в её форматной строке. В данном случае мы предполагаем, что во входном потоке присутствуют только десятичные вещественные числа (формат `%lf`). А как быть, если мы захотим вводить числа в разных системах счисления? Часто для этого используют соглашения о постфиксной записи, при которой за числом, записанным не в десятичной системе счисления, следует признак системы счисления. Например, `100h` — число записано в шестнадцатеричной системе счисления (десятичное 256); `100o` — число записано в восьмеричной системе счисления (десятичное 64); `100b` — число записано в двоичной системе счисления (десятичное 4).

Такое разнообразие функция `scanf` самостоятельно разобрать уже не в состоянии. В данном случае лексический анализатор должен сам выделить подстроку, содержащую число, из входного потока символов, определить тип числа и вызвать нужную функцию преобразования подстроки в числовой внутренний формат.

Программа такого лексического анализатора, учитывающая все нюансы, особенно с возможностями записи вещественных (десятичных) чисел, достаточно сложна. Написанная вручную, без использования аппарата теории детерминированных и недетерминированных конечных автоматов, она наверняка будет содержать изъяны в логике и, следовательно, потребует больших затрат на тестирование и отладку. И вот здесь на помощь может прийти генератор программ лексического анализа LEX, который позволит описать

все допустимые лексические конструкции как набор простых регулярных выражений.

Для диалоговой системы, какой является наш калькулятор, небольшое увеличение времени работы лексического анализатора при переходе на LEX для пользователя практически незаметно. Зато заметно будут сокращены затраты на разработку и сопровождение при неизменно высоком качестве итоговой программы.

В настоящей главе мы сначала построим lex-программу, повторяющую уже имеющийся у нас лексический анализатор на Си, и обеспечим её взаимодействие с синтаксическим анализатором. Затем покажем, как скорректировать make-программу, чтобы учесть нововведения в файловой структуре исходных текстов программы. В заключение введём в наш калькулятор возможность читать числа, заданные в разных системах счисления, а также возможность писать текстовые комментарии.

7.2. Замечания относительно совместного использования YACC и LEX

Поскольку в данной главе мы будем в рамках одной программы использовать оба рассматриваемых в настоящем учебном пособии генератора анализаторов (и лексического, и синтаксического), полезно будет сделать несколько замечаний относительно правил их сопряжения.

Прежде всего отметим, что оба генератора создают анализаторы, в которых все внешние имена начинаются на комбинацию символов `yy`, причём для лексического и синтаксического анализаторов эти имена не перекрываются. Таким образом обеспечивается возможность их совместного использования без необходимости правки текста полученных анализаторов.

Лексический и синтаксический анализаторы «знают» друг о друге достаточно для того, чтобы взаимодействовать без вмешательства программиста. Единственное, что необходимо сделать программисту, — это передать лексическому анализатору информацию о типе стека синтаксического анализатора и номерах лексем, а также организовать lex-программу таким образом, чтобы после каждой распознанной лексемы управление передавалось синтаксическому анализатору. Иными словами, для каждой лексемы в lex-программе предусматривается оператор `return`.

Информация о типе стека и номерах лексем содержится в файле `y.tab.h`, который генерируется автоматически при трансляции yacc-программы. Таким образом, для сопряжения анализаторов нам необходимо предусмотреть в lex-программе оператор `#include "y.tab.h"`.

7.3. Простой лексический анализатор

В этом параграфе приведён текст lex-программы простого лексического анализатора, который в точности повторяет возможности лексического анализатора, написанного нами вручную.

```
%{
#include <stdio.h>
#include "y.tab.h"

extern unsigned int nline;

long int idata;
%}
BUKVA        [a-z]
CIFRA        [0-9]
DCHISLO      {CIFRA}+
FCHISLOe     {CIFRA}*\.{CIFRA}*(e(\+|-)?{CIFRA}+)?
FCHISLOE     {CIFRA}*\.{CIFRA}*(E(\+|-)?{CIFRA}+)?
%%
\t|" "    {
       ;
  }
\n    {
  nline++;
  return *yytext;
  }
{BUKVA}   {
  yylval.index=*yytext-'a';
  return VAR;
  }
{DCHISLO}  {
  sscanf(yytext,"%ld",&idata);
  yylval.data=(double)idata;
  return DATA;
  }
{FCHISLOe} {
  sscanf(yytext,"%lg",&(yylval.data));
  return DATA;
  }
{FCHISLOE} {
  sscanf(yytext,"%lG",&(yylval.data));
  return DATA;
```

```
  }
.     {   /* другие символы */
  return *yytext;
  }
%%

yywrap()
{
    return 1;
}

resynch()
{
    int c;

    do{
        c=getchar();
    }while(c!='\n');

    ungetc(c,stdin);
}
```

Текст lex-программы записан в файл `lexical.l`.

В логику разбора входного потока здесь внесены незначительные изменения, касающиеся типов выделяемых лексем. Для дальнейших модификаций гораздо удобнее распознавать целые и вещественные числа как разные лексемы. Кроме того, для выполнения требований документации библиотеки Си вещественные числа, записываемые через разные основания степени (`e` и `E`), распознаются как разные лексемы и переводятся во внутренний формат через разные форматные строки (`%lg` и `%lG` соответственно). Но с точки зрения синтаксического анализатора логика работы нового лексического анализатора осталась прежней.

В разделе функций описаны две функции. Стандартная функция `yywrap()` просто возвращает `1`, сигнализируя о необходимости завершить работу. Функция `resynch()` скопирована из предыдущего модуля лексического анализатора. Она нужна, как Вы помните, исключительно для парирования ошибок в записи выражений, обнаруженных синтаксическим анализатором.

Если сравнить теперь размеры текстов лексического анализатора, написанного вручную, и его аналога, представленного lex-программой (а заодно посмотреть и текст итогового лексического

анализатора), то можно лишний раз убедиться, что для простых случаев, когда лексемы во входном потоке легко распознаются без применения сложных алгоритмов анализа, лексический анализатор, разработанный вручную, много эффективнее аналога, созданного с использованием генератора.

Новую версию программы калькулятора назовём `calc3`. Не забудьте переименовать файл с исходным текстом служебного модуля.

7.4. Коррекция make-программы

Для выполнения трансляции программы с новым лексическим анализатором нам необходимо внести изменения в make-программу. Однако, как это ни странно, для make-программы в ОС UNIX изменения не потребуются вообще*. В самом деле, утилита `make` прекрасно «разберётся» сама, из какого файла (`lexical.c` или `lexical.l`) должен быть получен файл `lexical.o`.

Впрочем, если в make-программе использовались дополнительные правила, типа правила получения архива исходных файлов, то в эти правила также придётся вносить изменения, чтобы учесть появление нового типа исходных файлов — типа lex-программы.

7.5. Использование новых возможностей лексического анализа

Теперь, когда мы перестроились на использование автоматически генерируемого лексического анализатора, покажем, как просто можно расширять список распознаваемых лексических единиц. Для этого добавим в наш калькулятор возможность вводить числа в двоичной, восьмеричной и шестнадцатеричной системах счисления, а также возможность задавать текстовые комментарии произвольной формы. Попутно будем отсекать на этапе лексического анализа все символы, не являющиеся законными лексемами.

Системы счисления будем определять по символу-признаку в конце записи числа. Потребуем, чтобы все числа начинались с цифры — это важно для записи шестнадцатеричных чисел. Например, запись `0ah` будет законным числом, а `ah` — это незаконная лексема.

Комментарии будем отмечать, как в языке Си, т. е. будем считать комментарием любую последовательность символов и строк, заключённую в группы `/*` и `*/`.

---

* Единственное, что придётся изменить — это имя программы с `calc2a` на `calc3`, поскольку мы условились таким способом различать версии программы.

Разделы деклараций и правил новой lex-программы приведены ниже.

```
%START PROG COMM
%{
#include <stdio.h>
#include "y.tab.h"

extern unsigned int nline;

long int idata;
%}
BUKVA       [a-z]
CIFRA       [0-9]
DCHISLO     {CIFRA}+
OCHISLO     [0-7]+[Oo]
SCHISLO     {CIFRA}({CIFRA}|[AaBbCcDdEeFf])*[Hh]
BCHISLO     [01]+[Bb]
FCHISLOe    {CIFRA}*\.{CIFRA}*(e(\+|-)?{CIFRA}+)?
FCHISLOE    {CIFRA}*\.{CIFRA}*(E(\+|-)?{CIFRA}+)?
%%
    {
    BEGIN PROG;
    }
<PROG>"+"|"-"|"*"|"/"|"("|")"|"^"|"="   {
    return *yytext;
    }
<PROG>\t|" "    {
        ;
    }
<PROG>\n    {
    nline++;
    return *yytext;
    }
<PROG>{BUKVA}   {
    yylval.index=*yytext-'a';
    return VAR;
    }
<PROG>{DCHISLO} {
    sscanf(yytext,"%ld",&idata);
    yylval.data=(double)idata;
    return DATA;
```

```
    }
<PROG>{OCHISLO} {
    sscanf(yytext,"%lo",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{SCHISLO} {
    sscanf(yytext,"%lx",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{BCHISLO} {
    getbin(&idata,yytext);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{FCHISLOe}    {
    sscanf(yytext,"%lg",&(yylval.data));
    return DATA;
    }
<PROG>{FCHISLOE}    {
    sscanf(yytext,"%lG",&(yylval.data));
    return DATA;
    }
<PROG>"/*"  {   /* Комментарий */
    BEGIN COMM;
    }
<COMM>"*/"  {
    BEGIN PROG;
    }
<COMM>.     {
    ;
    }
<COMM>\n    {
    nline++;
    }
<PROG>.     {   /* Нераспознанная лексема */
    calc_error(3,yytext);
    }
%%
```

Раздел функций lex-программы остаётся без изменений.

Новая lex-программа имеет два состояния: PROG и COMM. Состояние PROG является основным. В этом состоянии лексический анализатор распознаёт лексемы, вычисляет при необходимости их значения и передаёт результаты своей работы синтаксическому анализатору.

В состояние COMM анализатор входит при обнаружении комбинации символов /*. В этом состоянии анализатор игнорирует все символы, кроме символов возврата каретки (\n), который используется для подсчёта числа входных строк. Выход из состояния COMM обратно в состояние PROG осуществляется при обнаружении во входном потоке комбинации символов */. Обратите внимание, что при таком построении лексического анализатора вложенные комментарии недопустимы.

Для отсечения символов, не являющихся лексемами, нам пришлось перечислить все символы, которые мы считаем допустимыми во входном языке калькулятора. Любой другой символ на входе вызовет сообщение об ошибке.

Для преобразования во внутренний формат шестнадцатеричных и восьмеричных чисел используются стандартные строки формата функции sscanf. Для бинарных констант функцию преобразования придётся разработать самостоятельно. Вот один из вариантов её текста:

```
getbin(bin,str)
char *str;
long int *bin;
{
    long int rez;
    int i;

    rez=0;
    for(i=0;i<32 && *str;i++){
        rez<<=1;
        switch(*str){
        case '0' :
            break;
        case '1' :
            rez|=1;
            break;
        case 'b' :
        case 'B' :
            rez>>=1;
            *str='\0';
```

```
            goto endbin;
        default :
            calc_error(4,str);
            break;
        }
        str++;
    }
endbin:
    if(i==32 && *str!='\0')
        calc_error(5);

    *bin=rez;
}
```

Функция записывается в отдельный файл с именем `getbin.c`. Формат её вызова представлен в lex-программе.

Как уже, вероятно, заметил читатель, наши доработки потребовали введения новых кодов ошибок. Более того, для новых кодов ошибок доработан формат вызова функции `calc_error()`. Вторым аргументом в функцию передаётся строка, вызвавшая ошибку. Спешим обрадовать читателя — правка функции `calc_error()` выполнена таким образом, что никаких изменений в ранее записанных вызовах этой функции не потребуется. Между прочим, такое принципиально возможно только в языке Си. Вот текст исправленной функции:

```
char *ms_calc_error[]={
/* 0 */ "",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного",
/* 3 */ "Нераспознанная лексема %s",
/* 4 */ "Ошибка в~записи бинарной константы %s",
/* 5 */ "Бинарная константа больше 32 знаков"
};

calc_error(num,s)
int num;
char *s;
{
    fprintf(stderr,"\nОшибка! Строка %u\n(%02d) ",
                        nline,num);
    fprintf(stderr,ms_calc_error[num],s);
    fprintf(stderr,"\n");
```

```
    longjmp(jmp_env,1);
}
```
Как видно по тексту функции, теперь текст сообщения об ошибке сам является форматной строкой вывода. Поэтому, если в этой строке не содержатся символы шаблонов `%s`, то и второй аргумент функции `calc_error()` просто не используется.

Все остальные части программы остались неизменными. Не потребовалось ничего исправлять, в том числе и в синтаксическом анализаторе.

Нам осталось только добавить описание нового файла `getbin.c` в make-программу и снова изменить имя программы в соответствии с новой версией. Новую версию калькулятора назовём `calc3a`:

```
PROG = calc3a
YFLAGS = -d

$(PROG) : $(PROG).o syntax.o lexical.o getbin.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o getbin.o

lexical.o : y.tab.h
```

Выполним трансляцию программы (командой `make`) и проверим её работу:

```
./calc3a
2+3*5
    17
!#$

Ошибка! Строка 2
(03) Нераспознанная лексема !

Ошибка! Строка 2
(03) Нераспознанная лексема #

Ошибка! Строка 2
(03) Нераспознанная лексема $
```

При имеющихся правилах в lex-программе «посторонние» символы распознаются по-одному, поэтому мы и получаем сообщение об ошибке на каждый символ. Желательно, конечно, чтобы в подобном случае сообщение было только одно. Первая идея по исправлению ситуации состоит в том, чтобы поменять правило распознавания ошибочных символов на

```
<PROG>.+        {   /* Нераспознанная лексема */
    calc_error(3,yytext);
    }
```

Однако это приводит к прекращению правильного распознавания выражений. Причина этого заключается в том, что генерируемый анализатор стремится работать по так называемому «принципу наибольшей жадности», стремясь подобрать шаблон под самую длинную входную последовательность символов. А записанный нами шаблон обеспечивает это для почти любого набора входных символов. Поэтому более правильным является использовать уже созданную нами функцию `resynch()`, чтобы полностью пропустить остаток неправильной строки:

```
<PROG>.     {   /* Нераспознанная лексема */
    resynch();
    calc_error(3,yytext);
    }
```

Теперь при обнаружении во входном потоке «постороннего» символа будет выдано одно сообщение об ошибке (с указанием этого символа), и калькулятор перейдёт к новой строке, выбросив остаток ошибочно введённого выражения.

Глава 8
Использование имён произвольной длины

8.1. Постановка задачи

Регистровые переменные, введённые нами в главе 5, значительно облегчают выполнение вычислений сложных выражений. Однако в некоторых случаях их использование может оказаться неудобным. Во-первых, их всего 26. На практике же может возникнуть необходимость в запоминании большего количества чисел. Это прежде всего может быть связано с необходимостью использования общеупотребительных и специальных констант, которые неудобно каждый раз вводить вручную. Во-вторых, даже в 26 ячейках памяти, названных простыми буквами, легко «заблудиться» и забыть, в какой ячейке какая константа записана.

Гораздо удобнее работать с ячейками памяти, имеющими «человеческие» имена, т. е. имена произвольной длины. Ячейки с такими именами заводятся в память компьютера постепенно в процессе вычислений. Их может быть достаточно много — столько, сколько сможет разместиться в памяти компьютера. Поскольку право пользователя — дать ячейкам имена, соответствующие смыслу сохраняемых в них значений, то и проблемы с запоминанием где что лежит не будет.

Сразу оговоримся, что нет смысла разрабатывать язык, допускающий работу с бесконечно длинными именами. С такими именами работать будет практически невозможно. На практике достаточно опознавать вводимые пользователем имена по первым n символам, где n выбирается достаточно большим, чтобы не очень ограничивать пользователя в составлении подходящих для его задачи имён. При этом в принципе допускаются имена и большей длины, но их опознавание идёт только по этим первым n символам.

Кроме имён переменных, полезно иметь в калькуляторе набор встроенных наиболее употребительных констант и математических функций.

В данной главе мы сначала введём в наш калькулятор возможность работать с именами ячеек произвольной длины вместо ранее использовавшихся регистров. Затем добавим вызов наиболее используемых констант и математических функций.

8.2. Введение возможности работать с именами переменных произвольной длины

Прежде всего нам надо решить вопрос о способе хранения определяемых пользователем ячеек памяти. Проще всего это сделать с помощью связанного списка. Простой связанный список допускает поиск только линейным способом — последовательным просмотром всех элементов списка до первого совпадения с шаблоном поиска, но в нашем случае этого вполне достаточно. Новые имена проще всего добавлять к началу списка. Конец списка помечается специальным пустым указателем на следующий элемент. Удаление элементов из списка в нашем случае не предусматривается. Структура списка показана на рис. 8.1.

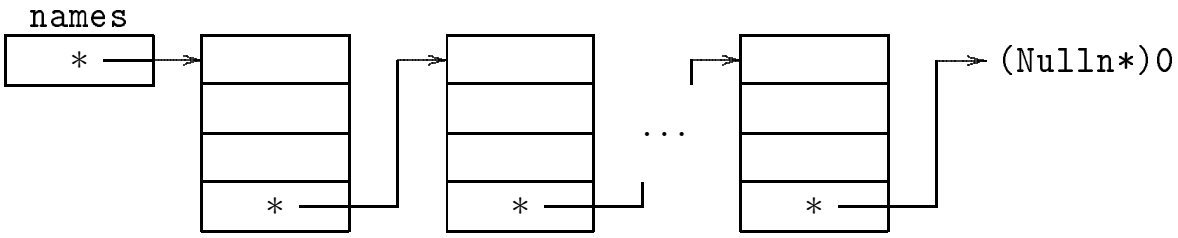

Рис. 8.1. Структура списка для хранения именованных ячеек памяти

8.2.1. Функции для работы с именами произвольной длины

Для систематизации работы со списком нам понадобятся две функции — функция добавления в список нового элемента (с выделением для него памяти) и функция поиска элемента по заданному имени. Такой подход позволит при необходимости поменять структуру списка и, например, алгоритм поиска, не затрагивая остальной программы.

С точки зрения файловой структуры программы, удобно поместить все функции, работающие с именами, в отдельный файл `names.c`. Однако некоторая информация из него может понадобиться в других файлах. Поэтому основные описания типов мы поместим в файл `names.h`.

Приведём содержимое файла `names.h`:

```
#define MAXLEN  20        /* максимальная длина имени */
#define Nulln   (Names*)0 /* значение пустого указателя */

typedef struct Ns{
char name[MAXLEN+1];    /* имя переменной */
double val;     /* значение переменной */
```

```
int def;    /* если =1, то имени уже присвоено значение */
struct Ns *sled;    /* указатель на следующую запись */
}Names;


Names* add_name();
Names* search_name();
```

В данном случае мы устанавливаем, что максимальная распознаваемая длина имени будет равна 20 символам. Этого вполне достаточно для практического применения. Элементом списка ячеек памяти будет являться объект типа `Names`, представляющий собой структуру, объединяющую имя ячейки, её значение, флаг определённости и указатель на следующий элемент списка. Флаг определённости `def` необходим для того, чтобы избежать использования в выражении переменной, которой ещё не присвоено значение.

В файле `names.h` мы также декларируем, что функции добавления нового элемента в список имён и поиска по списку возвращают указатель на найденный элемент (или пустой указатель, если элемент не найден).

Сами функции определены в файле `names.c`. Приводим здесь полный его текст:

```
#include <stdlib.h>
#include <string.h>
#include "names.h"

Names *names=Nulln; /* указатель на начало списка имён */

Names* malloc_name()
{
    Names *p;

    if((p=(Names*)malloc(sizeof(Names)))==NULL){
        calc_error(6);
    }

    return p;
}

Names* add_name(name)
char *name;
{
```

```
    Names *p;

    p=malloc_name();

    p->sled=names;
    names=p;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* ограничение длины имени */
    strcpy(p->name,name);
    p->def=0;       /* значение ещё не~присвоено */

    return p;
}

Names* search_name(name)
char *name;
{
    Names *p;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* ограничение длины имени */

    for(p=names;p!=Nulln;p=p->sled){
        if(strcmp(p->name,name)==0)
            break;  /* имя найдено */
    }

    return p;
}
```

Функция `malloc_name()` является служебной и нужна лишь для удобства программирования. Заметьте, что здесь введён новый код ошибки, который необходимо добавить в список ошибок служебного файла:

```
char *ms_calc_error[]={
/* 0 */ "",
    ...
/* 6 */ "Переполнение доступной оперативной памяти",
};
```

8.2.2. Изменение программы синтаксического анализатора

Для работы с новыми ячейками памяти необходимо изменить тип стека синтаксического анализатора. Попутно слегка изменим и сами синтаксические правила. Дело в том, что до настоящего времени калькулятор печатал на экране значение выражения всегда. Теперь мы добьёмся того, чтобы в случае присваивания переменной значение выражения не печаталось. Для этого мы добавим новый нетерминал `prisw`, соответствующий оператору присваивания.

Далее приводим полный текст нового файла `syntax.y`:

```
%{
#include <stdio.h>
#include <math.h>
#include "names.h"
%}

%union{
double data;    /* числовое значение */
Names *name;    /* указатель на имя */
}

%token <data> DATA
%token <name> VAR
%type <data> wyrag
%type <data> prisw
%right '='
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok prisw '\n'
  | spisok wyrag '\n'   { printf("\t%g\n",$2); }
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
```

```
prisw : VAR '=' wyrag   { $$=$1->val=$3;
              $1->def=1;
            }
  | VAR '=' prisw   { $$=$1->val=$3;
              $1->def=1;
            }
wyrag : DATA        { $$=$1; }
  | VAR         {
            if($1->def)
                $$=$1->val;
            else
                calc_error(7,$1->name);
            }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag {
            if($3==0.)
                calc_error(1);
            $$=$1/$3;
            }
  | '(' wyrag ')'   { $$=$2; }
  | wyrag '^' wyrag { $$=pow($1,$3); }
  | '-' wyrag %prec UNARYMINUS { $$=-$2; }
  ;
%%
```

Нетерминал `prisw` определён как одна из двух возможных последовательностей элементов грамматики. Для обеих последовательностей предусмотрены совершенно одинаковые действия, что на первый взгляд может показаться не очень правильным с точки зрения структуры программы. Однако попытки переопределить грамматику так, чтобы нетерминалу `prisw` соответствовало одно единственное действие, приводят к неоднозначности типа сдвиг/свёртка в самой грамматике. И хотя с неоднозначностями этого типа в грамматике в большинстве случаев можно мириться, лучше всё же, по возможности, их избегать.

Слегка изменено также правило обработки лексемы `VAR` в правиле для `wyrag`. Теперь перед использованием значения переменной проводится проверка на её определённость. Попутно добавлен ещё один код ошибки, который также необходимо внести в список ошибок в служебном файле. Общий список ошибок теперь выглядит так:

```
char *ms_calc_error[]={
/* 0 */ "",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного",
/* 3 */ "Нераспознанная лексема %s",
/* 4 */ "Ошибка в~записи бинарной константы %s",
/* 5 */ "Бинарная константа больше 32 знаков",
/* 6 */ "Переполнение доступной оперативной памяти",
/* 7 */ "Значение имени %s не~определено"
};
```

8.2.3. Изменение программы лексического анализатора

Наконец, необходимо позаботиться о распознавании имён произвольной длины во входном потоке. Изменения лексического анализатора, написанного на LEX, оказываются минимальными. Прежде всего необходимо определить в разделе деклараций шаблон для распознавания имени. Будем считать, что именем является любая последовательность букв и цифр, а также символов подчёркивания, начинающаяся с буквы:

```
    ...
BUKVA   [a-zA-Z]
CIFRA   [0-9]
IMJA    {BUKVA}({BUKVA}|{CIFRA}|"_")*
DCHISLO {CIFRA}+
    ...
```

Обратите внимание, что для использования в именах не только строчных, но и прописных букв расширено определение шаблона `BUKVA`. Между прочим, по заданному шаблону будут выделяться имена произвольной длины. В случае слишком длинных имён их укорачивание до 20 символов производится в функциях обработки имён файла `names.c`.

В разделе правил вместо правила для шаблона `{BUKVA` появляется правило для шаблона `{IMJA}`:

```
    ...
<PROG>\n        {
    nline++;
    return *yytext;
    }
<PROG>{IMJA}    {
    yylval.name=search_name(yytext);
    if(yylval.name==Nulln)
```

```
        yylval.name=add_name(yytext);
    return VAR;
    }
<PROG>{DCHISLO} {
    sscanf(yytext,"%ld",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
    ...
```

Обратите внимание, что лексический анализатор лишь производит поиск имени в таблице имён и, если имя не обнаружено, добавляет его в таблицу. Дело в том, что лексический анализатор «не знает», в каком контексте будет использована переменная с этим именем. Если переменной будет присваиваться новое значение, то такая операция допустима всегда. Если же значение переменной используется в выражении, то переменная должна быть уже определена (ей ранее должно быть присвоено значение). Такая проверка, как Вы уже видели, выполняется синтаксическим анализатором.

Наконец, для правильной работы лексического анализатора необходимо сообщить ему типы значений, возвращаемых функциями работы с именами. Для этого в начало lex-программы между директивами `%{` и `%}` надо добавить строку

```
#include "names.h"
```

Исходный текст новой программы готов к трансляции. Назовём новую программу `calc4`.

8.2.4. Изменение make-программы

В данном случае никаких принципиально новых инструментальных средств в программу добавлено не было. Всё, что нужно сделать — это описать зависимость итоговой программы и некоторых объектных модулей от файлов `names.c` и `names.h`. Для ОС UNIX make-программа усложнится незначительно:

```
PROG = calc4
YFLAGS = -d

$(PROG) : $(PROG).o syntax.o lexical.o getbin.o names.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o

lexical.o : y.tab.h names.h
syntax.o : names.h
```

После произведённых правок можно транслировать программу и проверять её работу.

8.3. Добавление предопределённых констант

Такие константы, как π или основание натурального логарифма e, удобно иметь введёнными в память калькулятора по умолчанию. Есть несколько способов доработки нашего калькулятора для этой возможности. Мы используем наиболее простой, хотя может быть и не самый эффективный с точки зрения использования оперативной памяти. Будем хранить предопределённые константы как обычные переменные, введённые нами в калькулятор в этой главе. Для простоты модификации списка констант организуем таблицу констант, в соответствии с которой и будем осуществлять начальную инициализацию переменных.

Всё, что нам потребуется — таблица констант и функция инициализации переменных, которая вызывается в самом начале работы программы. Логичнее всего записать их в файл `names.c`.

```
/* таблица констант */
struct Tn{
char name[MAXLEN+1];    /* имя константы */
double val;     /* значение */
}Tnames[]={
"Pi",3.14159,       /* число Пи */
"E",2.718282,       /* основание натурального логарифма */
"DEG",57.2958,      /* отношение радиана к~градусу */
"PHI",1.61803,      /* золотое сечение */
};

int len_Tnames=sizeof(Tnames)/sizeof(struct Tn);

init_names()
{
    Names *p;
    int lentab;
    int i;

    for(i=0;i<len_Tnames;i++){
        p=add_name(Tnames[i].name);
        p->val=Tnames[i].val;
        p->def=1;
    }
}
```

Обратите внимание на способ вычисления длины таблицы. Такая запись позволяет свободно изменять число элементов в таблице констант, не заботясь об изменении других элементов программы.

Для завершения изменений надо вставить вызов функции инициализации `init_names()` в функцию `main()` служебного модуля:

```
main()
{
    init_names();

    signal(SIGFPE,fpeerr);
    if(setjmp(jmp_env));
    yyparse();
}
```

Теперь можно оттранслировать программу и проверить работоспособность её новой возможности.

8.4. Добавление математических функций

Математические функции также можно рассматривать как имена ячеек памяти произвольной длины с той лишь разницей, что их значения вычисляются динамически в момент обращения и зависят от аргумента функции. Такой тип представления потребует введения в тип элемента списка имён ячеек специального флага, указывающего на тип ячейки памяти. Соответственно несколько изменится и работа с элементами списка.

Мы используем другой подход, при котором поиск имён математических функций будем осуществлять в специальной таблице функций. Этот подход потребует добавления в синтаксис нашего калькулятора нового терминального символа — `FUN` (вызов функции). Значением этого терминала будем считать индекс в таблице функций. Кроме таблицы функций, нам потребуются две функции обслуживания — функция поиска имени в таблице функций и функция реализации заданной в выражении математической функции:

```
#include <math.h>

/* таблица математических функций */
struct Tf{
char name[MAXLEN+1];     /* имя функции */
double (*fun)();     /* указатель на функцию */
}Tablf[]={
"sin",sin,
"cos",cos,
```

```
"tg",tan,
};

int len_Tablf=sizeof(Tablf)/sizeof(struct Tf);

search_fun(name)
char *name;
{
    int i;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* ограничение длины имени */

    for(i=0;i<len_Tablf;i++){
        if(strcmp(name,Tablf[i].name)==0)
            return i;   /* функция найдена */
    }

    return -1;  /* поиск неудачен */
}

double val_fun(index,arg)
int index;
double arg;
{
    return (*Tablf[index].fun)(arg);
}
```

Функции логично разместить в файле `names.c`. Поскольку функция `val_fun()` будет использоваться в других модулях, описание типа возвращаемого ею значения надо поместить в файл `names.h`:

```
double val_fun();
```

В таблице математических функций каждому имени функции сопоставлен указатель на соответствующую функцию. Как и в случае с таблицей предопределённых констант, эту таблицу можно безболезненно изменять, добавляя и удаляя записи. Функция реализации заданной математической функции `val_fun()` обеспечивает вызов функции по индексу в таблице.

Нам потребуются также небольшие изменения в yacc-программе и lex-программе.

В yacc-программе необходимо описать тип новой лексемы `FUN`, добавить тип индекса в стек анализатора, а также добавить правило для обработки функции:

```
    ...
%union{
double data;    /* числовое значение */
Names *name;    /* указатель на имя */
int index;  /* номер функции в~таблице функций */
}

%token <data> DATA
%token <name> VAR
%token <index> FUN
    ...
%%
    ...
wyrag : DATA        { $$=$1; }
  | FUN '(' wyrag ')'   { $$=val_fun($1,$3); }
    ...
```

В lex-программе необходимо обеспечить поиск имени в таблице функций:

```
<PROG>{IMJA}      {
    if((yylval.index=search_fun(yytext))!=-1)
        return FUN;
    yylval.name=search_name(yytext);
    if(yylval.name==Nulln)
        yylval.name=add_name(yytext);
    return VAR;
    }
```

Это все исправления. Назовём её `calc4a`. Поскольку мы не добавляли новые файлы, исправлений make-программы не требуется (кроме имени версии). Оттранслируем программу и проверим её работу:

```
Pi
    3.14159
sin(Pi)^2+cos(Pi)^2
    1
```

Глава 9
Компиляция на машину

9.1. Общие понятия

В этой главе мы преобразуем программу `calc4a` в программу `calc5` с теми же вычислительными возможностями, но имеющую принципиально иную внутреннюю организацию. Если в программе `calc4a` вычисление заданного выражения происходит в процессе разбора самого выражения и по окончании этого процесса сразу выдаётся результат, то в программе `calc5` в результате разбора входного выражения будет порождён код некоторой простой машины. При определении конца выражения порождённый код будет выполнен (проинтерпретирован), в результате чего выражение будет вычислено.

В качестве простой машины будет использована стековая машина. В этой машине операнды, встречающиеся в выражении, заносятся в стек. Операции выполняются над операндами, извлекаемыми из вершины стека, и результаты операций также заносятся в стек.

Работой со стеком управляет порождённая на этапе разбора исходного выражения программа. Например, при обработке выражения

```
x=2*y
```

будет порождена следующая программа:

```
Записать в~стек константу
... константа 2
Записать в~стек значение ячейки
... с~именем y
Перемножить два верхних элемента; результат заменяет их
Присвоить значение верхнего элемента стека ячейке
... с~именем x
Удалить верхний элемент из стека
СТОП
```

В приведённой программе каждая строка соответствует одной команде простой машины. Мы записали лишь смысл команд. Кодировка команд может носить произвольный характер и зависеть от конечных целей построения интерпретатора.

9.2. Принцип внутренней организации машины

Стековые машины обычно организуются с помощью простых интерпретаторов. Такая машина имеет стековую память для работы с операндами и программную память, содержащую коды операций.

Стековая память представляет собой простой массив, управляемый двумя командами: `push()` — положить слово данных в стек и `pop()` — извлечь слово данных из стека.

Программная память — это тоже массив, содержащий операции и операнды. Операции представляют собой машинные команды. Они могут быть закодированы различными способами. Мы будем считать, что каждая машинная команда — суть обращение к функции, реализующей эту команду. Команда может работать с операндами, записанными в программной памяти вслед за операцией, и/или с данными, находящимися в вершине стека.

9.3. Преобразование калькулятора

9.3.1. Изменение синтаксического анализатора

По сравнению с `calc4a` грамматика калькулятора практически не изменится. Но действия (в разделе правил) будут совершенно другие. В связи с этим немного изменится и раздел деклараций уасс-программы. Ниже приведён полный текст новой уасс-программы.

```
%{
#include <math.h>
#include "names.h"
#include "code.h"
%}

%union{
Names *name;     /* указатель на имя */
int index;  /* номер функции в~таблице функций */
}

%token <name> DATA
%token <name> VAR
%token <index> FUN
%right '='
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
```

```
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok prisw '\n'   {   code((Prog)pop);
                code(STOP);
                return 1;
            }
  | spisok wyrag '\n'   {   code(print);
                code(STOP);
                return 1;
            }
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
prisw : VAR '=' wyrag   { code(assign); code((Prog)$1); }
  | VAR '=' prisw   { code(assign); code((Prog)$1); }
  ;
wyrag : DATA        { code(pushconst); code((Prog)$1); }
  | VAR         { code(eval); code((Prog)$1); }
  | FUN '(' wyrag ')'   { code(fval); code((Prog)$1); }
  | wyrag '+' wyrag { code(add); }
  | wyrag '-' wyrag { code(sub); }
  | wyrag '*' wyrag { code(mul); }
  | wyrag '/' wyrag { code(calcdiv); }
  | '(' wyrag ')'   { ; }
  | wyrag '^' wyrag { code(power); }
  | '-' wyrag %prec UNARYMINUS { code(negate); }
  ;
%%
```

Как видно из текста yacc-программы, теперь каждое действие порождает набор машинных команд, записываемых в программную память функцией `code()`. Тип её аргумента — `Prog` — определён в файле `code.h` и представляет собой указатель на функцию, реализующую соответствующую команду.

Команды `assign` (присвоить ячейке памяти значение из вершины стека), `pushconst` (записать в стек константу), `eval` (записать

в стек значение ячейки памяти) и `fval` (вычислить значение встроенной функции по аргументу в вершине стека; результат заменяет аргумент в стеке) требуют дополнительный операнд — указатель на ячейку памяти, соответствующую именованной ячейке или константе, либо индекс в таблице встроенных функций.

Остальные команды работают со значениями, уже записанными в стек. Так, например, команда `add`, соответствующая операции `+`, выполняет сложение двух верхних элементов стека, удаляя их из стека и помещая в стек результат.

Необходимо заметить, что команда деления названа не `div`, а `calcdiv`, поскольку функция с именем `div()` имеется в стандартной библиотеке языка Си (выполняет вычисление целочисленного деления с получением остатка).

Обратите внимание, что больше нет необходимости в задании типа нетерминальных символов, поскольку нигде в правилах они не фигурируют в качестве параметров действий.

Синтаксический анализатор возвращает «не ноль» при обнаружении конца выражения. Если будет обнаружен конец файла, он вернёт «ноль». На этом основывается способ определения момента завершения работы программы.

Для использования нового синтаксического анализатора необходимо изменить его вызов в функции `main()` служебного модуля:

```
main()
{
    init_names();

    signal(SIGFPE,fpeerr);
    if(setjmp(jmp_env));
    for(code_init();yyparse();code_init()){
        execute();
    }
}
```

Здесь функция `code_init()` обеспечивает приведение стековой машины в исходное состояние, а функция `execute()` выполняет интерпретацию полученного в результате разбора программного кода.

9.3.2. Изменение лексического анализатора

Для лексического анализатора необходимо изменить только действия, связанные с распознаванием констант. Если раньше распознанная константа просто передавалась в стек синтаксического анализатора, то теперь необходимо разместить её где-нибудь в памяти и передать синтаксическому анализатору указатель на неё. Наиболее простым (хотя далеко не самым лучшим с точки зрения распределения памяти) решением является запись констант в таблицу имён с выделением для них ячеек памяти с пустыми именами.

Поскольку наш лексический анализатор распознаёт различные типы констант, то для унификации этого процесса удобно использовать вспомогательную функцию, которую надо записать в раздел функций lex-программы. Изменённые участки lex-программы приведены ниже:

```
    ...
%%
    ...
<PROG>{ДЧИСЛО}  {
    sscanf(yytext,"%ld",&idata);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{ОЧИСЛО}  {
    sscanf(yytext,"%lo",&idata);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{ШЧИСЛО}  {
    sscanf(yytext,"%lx",&idata);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{БЧИСЛО}  {
    getbin(&idata,yytext);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
```

```
<PROG>{ФЧИСЛОe} {
    sscanf(yytext,"%lg",&data);
    make_const(data);
    return DATA;
    }
<PROG>{ФЧИСЛОЕ} {
    sscanf(yytext,"%lG",&data);
    make_const(data);
    return DATA;
    }
    ...
%%
    ...
make_const(data)
double data;
{
    yylval.name=add_name("");
    (yylval.name)->val=data;
}
```

9.3.3. Создание стековой машины

Всю программу стековой машины мы поместим в файл `code.c`, а связанные с ним описания, необходимые в других файлах, — в файл `code.h`.

Стековая машина включает в себя описания областей памяти для стека и команд программы, функции работы с этими областями, функцию интерпретации, а также функции, обеспечивающие собственно выполнение машинных команд.

Приведём сначала текст файла `code.h`:

```
#define MAXSTACK      200     /* размер стека машины */
#define MAXPROG     1000    /* размер программы машины */

#define STOP    (Prog) 0

typedef double Stack;   /* тип данных стека */

typedef int (*Prog)();  /* тип данных программы */

add(), sub(), mul(), calcdiv(), power(), negate();
assign(), pushconst(), eval(), fval(), print();
Stack pop();
```

Этот файл достаточно прост, оставим его без комментариев.

Файл `code.c` имеет более 150 строк. Рассмотрим его по частям. Как обычно, файл начинается с определений, используемых в дальнейшем. В данном случае выделяются стековая и программная память, а также определяются указатели, обеспечивающие работу с ними:

```
#include <math.h>
#include <stdio.h>
#include "names.h"
#include "code.h"

Stack stack[MAXSTACK];  /* стек машины */

Stack *pstack;      /* указатель на вершину стека */

Prog prog[MAXPROG]; /* массив программы машины */

Prog *pprog;    /* указатель на первый свободный элемент */
Prog *pc;       /* счётчик команд */
```

Функция `code_init()` просто устанавливает начальные значения указателей:

```
code_init()
{
    pprog=prog;
    pstack=stack;
}
```

Функция `code()` помещает очередную операцию (или операнд) в программную память:

```
code(f)
Prog f;
{
    if(pprog>=&prog[MAXPROG])
        calc_error(10);
    *pprog++=f;
}
```

Функция интерпретации готовой программы фантастически проста — она просто выполняет вызовы функций-команд, указатели на которые записаны в программной памяти:

```
execute()
{
    for(pc=prog;*pc!=STOP;)
```

```
        (*pc++)();
}
```

Поскольку практически все машинные команды выполняют операции со стеком, нам потребуются две функции обслуживания стека — `push()` и `pop()`:

```
push(d)
double d;
{
    if(pstack>=&stack[MAXSTACK])
        calc_error(8);
    *pstack++=d;
}

Stack pop()
{
    if(pstack<=stack)
        calc_error(9);
    return *--pstack;
}
```

Сами функции машинных команд достаточно просты. Большинство из них может быть получено простым копированием и незначительной правкой текста одной функции.

```
add()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1+=d2;
    push(d1);
}

sub()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1-=d2;
    push(d1);
}
```

```
mul()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1*=d2;
    push(d1);
}

calcdiv()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d2==0.)
        calc_error(1);
    d1/=d2;
    push(d1);
}

power()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1=pow(d1,d2);
    push(d1);
}

negate()
{
    Stack d;

    d=pop();
    d=-d;
    push(d);
}
```

```
print()
{
    Stack d;

    d=pop();
    printf("\t%g\n",d);
}
```

Команды-функции, имеющие операнды в программной памяти, сами продвигают счётчик команд pc, чтобы обеспечить его правильное положение для выполнения следующей команды:

```
assign()
{
    Stack d;
    Names *pn;

    d=pop();
    pn=(Names*)(*pc++);
    pn->val=d;
    pn->def=1;
    push(d);
}
eval()
{
    Stack d;
    Names *pn;

    pn=(Names*)(*pc++);
    if(pn->def==0)
        calc_error(7,pn->name);

    d=pn->val;
    push(d);
}

pushconst()
{
    Names *pn;

    pn=(Names*)(*pc++);
    push((Stack)(pn->val));
```

```
}

fval()
{
    Stack d;

    d=pop();
    d=val_fun((int)(*pc++),d);
    push(d);
}
```

Обратите внимание на то, что добавились новые коды ошибок. Вот текст соответствующих им сообщений (записываются в файле служебного модуля):

```
    ...
/* 8 */ "Переполнение стека",
/* 9 */ "Опустошение стека",
/* 10 */ "Переполнение памяти программы"
};
```

После внесения всех изменений остаётся только дополнить make-программу. Необходимо указать появление нового файла `code.c` и зависимость некоторых других файлов дополнительно от `code.h`.

Работа нового калькулятора `calc5` ничем внешне не отличается от работы калькулятора `calc4a`. Единственное отличие, незаметное на современных скоростных процессорах, — небольшое увеличение времени вычисления выражений. Однако `calc5` может быть теперь дополнен новыми возможностями, которые в принципе невозможно реализовать в `calc4a`. Эти возможности мы рассмотрим в следующей главе.

9.3.4. Замечание о команде pop

Описанная в настоящей главе стековая машина в некоторых случаях может работать неустойчиво. Проблема скрыта в том, что «стандартная» функция `pop()` возвращает в качестве результата работы значение типа `Stack` (`double`). Поскольку функция используется не только для получения значений из вершины стека, но и просто для очистки стека, то возможно возникновение внутренней ошибки работы с плавающей арифметикой.

Для исключения этой ошибки необходимо все вызовы функции `pop()`, связанные с очисткой стека, заменить на вызовы специальной функции `mpop()`, которая выполняет ту же работу, но ничего не возвращает:

```
mpop()
{
    if(pstack<=stack)
        calc_error(9);
    --pstack;
}
```
Во избежание недоразумений укажем, что заменены должны быть только вызовы функции `pop()` в действиях правил yacc-программы.

Глава 10
Структуры управления

10.1. Принципы организации структур управления в простой машине

Под структурами управления будем понимать традиционные операторы большинства языков программирования, предназначенные для проверки условий (операторы `if then else`), организации циклов (операторы `while`, `for`), печати значений переменных (оператор `print`), а также группирования операторов в блоки.

Для «формулирования» условий нам потребуются операции отношений, которые также необходимо ввести в синтаксис нашего калькулятора. Это могут быть как собственно операции отношений (`<`, `<=`, `>` и т. п.), так и логические операции (`&&`, `||` и др.). Эти операции могут рассматриваться в выражениях наравне с арифметическими операциями, и результат их работы также должен помещаться в вершину стека. Только в качестве результата будет выступать логическое значение, которое мы условимся кодировать нулём, если результатом вычисления выражения будет ЛОЖЬ, и единицей, если ИСТИНА.

В данной главе мы не будем стремиться к реализации в нашем простом калькуляторе всех перечисленных структур. Ограничимся возможностью группировать операторы в блоки (как в языке Си), оператором печати и оператором `while`. Из операций для формулирования условий ограничимся лишь собственно операциями отношений. Любые другие операторы могут быть реализованы по аналогии с рассмотренными.

Наибольший интерес и сложность из рассмотренных операторов представляет реализация оператора `while`. В нашей простой машине «машинной командой», соответствующей этому оператору, будет являться функция `fwhile()`. Она предполагает, что вслед за её адресом в программной памяти располагаются два слова, содержащие адреса начала тела цикла и первой команды, следующей за циклом, а вслед за этими словами — программа проверки условия цикла. Программа проверки условия и программа тела цикла заканчиваются командами `STOP` (рис. 10.1).

Принцип работы такой машинной программы заключается в том, что функция `execute()` выполняет команду за командой, пока не наткнётся на команду `STOP`, после чего она осуществляет возврат в вызвавшую её программу. Тело цикла `while` и условие

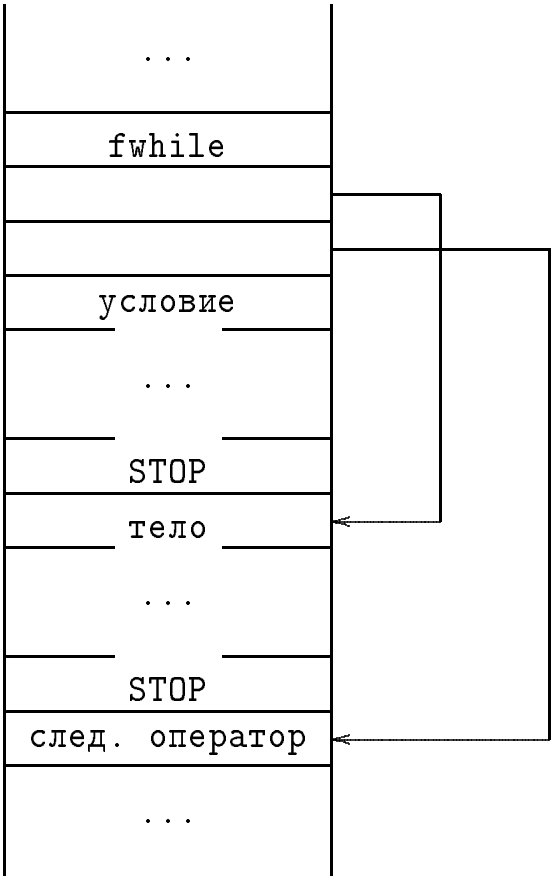

Рис. 10.1. Принцип внутренней организации оператора `while`

завершения цикла обрабатываются рекурсивными вызовами функции `rexecute()`, осуществляющими возврат на один уровень вложенности выше при обнаружении команды `STOP`. Команда-функция `fwhile()` организует эти рекурсивные вызовы. Нам придётся немного изменить текст и самой функции `execute()`. Вот текст этих трёх функций:

```
execute()
{
    rexecute(prog);
}

rexecute(prg)
Prog *prg;
{
    for(pc=prg;*pc!=STOP;)
        (*pc++)();
}
```

```
fwhile()
{
    Stack d;
    Prog *opc;

    opc=pc;          /* тело цикла */
    rexecute(opc+2);    /* условие цикла */
    d=pop();
    while(d!=0.){
        rexecute(*opc);
        rexecute(opc+2);
        d=pop();
    }
    pc=*((Prog**)opc+1);    /* первый оператор за циклом */
}
```

Чтобы понять принцип работы функции `fwhile()`, необходимо вспомнить, что, когда функция получает управление, счётчик команд `pc` уже указывает на следующее слово в программе, которое в данном случае является адресом начала тела цикла. Следующее слово при этом оказывается адресом первой команды, следующей за циклом, а слово по смещению 2 — началом программы проверки условия.

Другие структуры управления могут быть реализованы аналогичным способом. Только, например, для оператора ветвления `if` потребуется не два, а три адресных слова: для ветви `then`, для ветви `else` и для первой команды за оператором ветвления.

10.2. Доработки программы калькулятора

10.2.1. Изменения в yacc-программе

Изменения в yacc-программе значительны. Кроме добавления новых правил, изменены действия в уже существующих правилах. Теперь значением каждого правила является адрес начала машинной программы, реализующей это правило:

```
%{
#include <math.h>
#include "names.h"
#include "code.h"
%}

%union{
Names *name;    /* указатель на имя */
```

```
int index;  /* номер функции в~таблице функций */
Prog *cprog;    /* команда программы */
}

%token <name> DATA
%token <name> VAR
%token <index> FUN
%token <name> PRINT WHILE
%type <cprog> wyrag oper operspis while prisw
%right '='
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok prisw '\n'   {   code(mpop);
                code(STOP);
                return 1;
            }
  | spisok wyrag '\n'   {   code(print);
                code(STOP);
                return 1;
            }
  | spisok oper '\n'    {   code(STOP);
                return 1;
            }
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
oper : prisw        { code(mpop); }
  | wyrag       { code(mpop); }
  | PRINT wyrag     { code(pri); $$=$2; }
  | while '(' wyrag ')' { code(STOP); } oper    {
                code(STOP);
```

```
                pc=$1;
                pc[1]=(Prog)$6; /* тело цикла */
                pc[2]=(Prog)pprog;  /* конец цикла */
            }
  | '{' operspis '}'    { $$=$2; }
  ;
while : WHILE       {   $$=code(fwhile);
                code(STOP);
                code(STOP);
            }
operspis : /* пусто */  { $$=pprog; }
  | operspis '\n'
  | operspis oper
  ;
prisw : VAR '=' wyrag   { code(assign); code((Prog)$1);
                          $$=$3; }
  | VAR '=' prisw   { code(assign); code((Prog)$1);
                          $$=$3; }
  ;
wyrag : DATA        { $$=code(pushconst); code((Prog)$1); }
  | VAR         { $$=code(eval); code((Prog)$1); }
  | FUN '(' wyrag ')'   { code(fval); code((Prog)$1);
                          $$=$3; }
  | wyrag '+' wyrag { code(add); }
  | wyrag '-' wyrag { code(sub); }
  | wyrag '*' wyrag { code(mul); }
  | wyrag '/' wyrag { code(div); }
  | '(' wyrag ')'   { $$=$2; }
  | wyrag '^' wyrag { code(power); }
  | '-' wyrag %prec UNARYMINUS { code(negate); $$=$2; }
  | wyrag GT wyrag  { code(gt); }
  | wyrag GE wyrag  { code(ge); }
  | wyrag LT wyrag  { code(lt); }
  | wyrag LE wyrag  { code(le); }
  | wyrag EQ wyrag  { code(eq); }
  | wyrag NE wyrag  { code(ne); }
  ;
%%
```

Функция `code()` теперь возвращает адрес записанной машинной команды. Вот её новый текст:

```
Prog* code(f)
```

```
Prog f;
{
    Prog *oprog;

    oprog=pprog;
    if(pprog>=&prog[MAXPROG])
        calc_error(10);
    *pprog++=f;

    return oprog;
}
```

Обратите внимание на приоритет операций отношения. В данном случае он выбран в соответствии с приоритетом таких операций в языке Си: их приоритет выше приоритета операции присваивания. Однако ничто не мешает Вам его изменить.

При описании оператора `while` в yacc-программе использована возможность задавать действия не только для всего правила, но и для части уже распознанного правила.

Приведённая здесь грамматика содержит четыре конфликта типа сдвиг/свёртка (об этом выдаётся сообщение при трансляции), что, однако, не мешает ей правильно работать.

Внимательный читатель обратил, вероятно, внимание на то, что не во всех правилах вычисления выражений заданы действия определения значения правила (типа `$$=`). Напомним, что если такое действие не задано, то значением правила является значение первого элемента правила (`$$=$1`).

10.2.2. Изменения в lex-программе

Здесь изменения сведутся к необходимости распознавать ключевые слова и знаки операций отношения и группирования:

```
    ...
<PROG>">"       {
    return GT;
    }
<PROG>">="  {
    return GE;
    }
<PROG>"<"   {
    return LT;
    }
<PROG>"<="  {
```

```
    return LE;
    }
<PROG>"=="  {
    return EQ;
    }
<PROG>"!="  {
    return NE;
    }
<PROG>"+"|"-"|"*"|"/"|"("|")"|"^"|"="|"{"|"}"   {
    return *yytext;
    }
    ...
<PROG>{ИМЯ} {
    if((idata=search_kword(yytext))!=-1)
        return idata;
    if((yylval.index=search_fun(yytext))!=-1)
        return FUN;
    yylval.name=search_name(yytext);
    if(yylval.name==Nulln)
        yylval.name=add_name(yytext);
    return VAR;
    }
    ...
```

Ключевые слова могут распознаваться и передаваться синтаксическому анализатору различными способами. В данном случае выбран способ, при котором все ключевые слова сведены в таблицу, где каждому слову сопоставлен номер терминала. Для поиска по этой таблице создана специальная функция:

```
/* таблица ключевых слов */
struct Tw{
char name[MAXLEN+1];    /* слово */
int term;       /* номер терминала */
}Tablword[]={
"print",PRINT,
"while",WHILE
};

int len_Tablword=sizeof(Tablword)/sizeof(struct Tw);

search_kword(name)
char *name;
```

```
{
    int i;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* ограничение длины имени */

    for(i=0;i<len_Tablword;i++){
        if(strcmp(name,Tablword[i].name)==0)
            return Tablword[i].term;    /* слово найдено */
    }

    return -1;  /* поиск неудачен */
}
```

Таблица ключевых слов и функция поиска по этой таблице записаны в файл `names.c`. Найденный номер терминала просто передаётся синтаксическому анализатору. Никакая дополнительная информация при этом не требуется.

10.2.3. Изменения в других модулях

В этом пункте мы уделим внимание тем новым командам-функциям, которые не были рассмотрены ранее.

Оператор печати, представленный функцией `pri()`, — это не та операция печати, которая выполняется по завершении разбора выражения. Этот оператор печатает заданное значение без перевода строки, используя символ табуляции в качестве разделителя, что позволяет выводить в одну строку несколько значений:

```
pri()
{
    Stack d;

    d=pop();
    printf("\t%g",d);
}
```

Команды-функции операций отношения весьма просты и могут быть получены копированием одной функции с минимальными последующими правками. Всего необходимо шесть функций (по числу возможных условий отношения): `gt()`, `ge()`, `lt()`, `le()`, `eq()`, `ne()`. Их текст будет отличаться только знаком проверки условия. Поэтому здесь приведём только одну функцию:

```
gt()
{
```

```
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1>d2)
        push(1.);
    else
        push(0.);
}
```

В заключение отметим, что изменения в программе калькулятора потребовали дополнительного включения файлов `syntax.h` и `code.h` в другие модули. В результате несколько усложняются и зависимости в make-программе, хотя никаких новых модулей в программу добавлено не было:

```
PROG = calc6
YFLAGS = -d
CFLAGS = -g

$(PROG) : $(PROG).o syntax.o lexical.o \
            getbin.o names.o code.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o \
            getbin.o names.o code.o

lexical.o : y.tab.h names.h code.h
syntax.o : names.h code.h
names.o : y.tab.h code.h
code.o : names.h
```

Новую программу калькулятора назовём `calc6`. Внесём соответствующие изменения в make-программу и выполним трансляцию.

Полный текст программы приведён в приложении 1.

10.3. Пример использования новых возможностей

Прежде всего полезно убедиться, что калькулятор сохранил в полном объёме свои прежние вычислительные возможности. Выполните эту проверку самостоятельно.

Теперь проверим работу новых возможностей калькулятора. Для начала выполним простой цикл со счётчиком:

```
a=0
while(a<=10){
    print a
    a=a+1
}
    0   1   2   3   4   5   6   7   8
9   10
```

Если результат работы соответствует приведённому в книге, то можно попробовать вычислить что-нибудь более полезное. Запишем, например, программу расчёта чисел Фибоначчи:

```
a=0
b=1
while(b<1000){
    print b
    c=b
    b=a+b
    a=c
}
    1   1   2   3   5   8   13  21  34
55  89  144 233 377 610 987
```

Глава 11
Дальнейшее развитие калькулятора

11.1. Развитие языка калькулятора

Уже с имеющимися возможностями наш калькулятор может оказаться достаточно удобным средством для решения различных вычислительных задач. Мы можем, например, вычислить определённый интеграл (допустим, методом трапеций) или получить таблицу значений функции. Мы можем, наконец, просто вычислять достаточно сложные выражения, сохраняя результаты вычислений в ячейках памяти и используя их в дальнейших расчётах.

Однако в процессе работы с калькулятором может возникнуть необходимость в автоматизации вычислений специального вида. Это в свою очередь потребует доработки калькулятора. Некоторые из возможных полезных доработок мы сейчас рассмотрим.

Прежде всего заметим, что вычислительные возможности калькулятора значительно возрастут, если в нём реализовать бóльшую часть идей, предложенных в упражнениях (см. приложение 3). Кроме того, можно ввести в калькулятор такой интересный структурный элемент, как функции и подпрограммы с параметрами, определяемые пользователем. Это значительно повысит гибкость языка калькулятора.

Интересной представляется доработка, позволяющая вычислять определённые интегралы не программированием циклов, а вызовом специальной функции интегрирования. В этом случае, кстати, путём последовательных доработок интегрирующей функции можно значительно повысить точность результата.

Ещё одна полезная доработка — введение в калькулятор возможности работы с матрицами. Здесь необходимо определить конструкции языка, позволяющие задавать матрицы или формировать их в процессе вычислений, а также набор операторов для преобразования матриц. Необходимо также решить вопрос о внутреннем способе хранения матриц.

Можно предложить и более экзотические расширения — численное дифференцирование функций, решение систем линейных уравнений, решение задач оптимизации, решение дифференциальных уравнений и т. п.

К числу полезных расширений можно также отнести возможность построения графика функции. Выше уже было сказано о том, что калькулятор позволяет получить таблицу значений заданной функции. Поэтому эта задача может быть решена двумя путями.

Во-первых, в язык калькулятора можно ввести операторы типа `начать график`, `построить точку`, `задать подпись`. Во-вторых, можно ввести один оператор типа `построить график` для заданной функции.

В калькулятор могут быть также внесены дополнения, ориентированные на определённую область деятельности пользователя. Например, для химиков могут быть полезны функции расчёта баланса в химических реакциях; для радиоэлектронщиков — функции расчёта различных фильтров и т. д.

Наконец, можно так расширить язык калькулятора, чтобы он допускал вычисление выражений, записанных по правилам издательской системы TEX. Это достаточно сложная задача, поскольку выражения на TEX'е могут быть записаны с использованием макрокоманд с большой вложенностью. Однако для простых случаев задача может быть решена достаточно полно.

Можно также предусмотреть форматирование выходного потока калькулятора в соответствии с правилами TEX'а. Это позволило бы включать результаты расчётов в документы на TEX'е без дополнительного форматирования.

11.2. Методы реализации

11.2.1. Интерпретатор

Все варианты реализаций калькулятора в этом учебном пособии представляют собой различные варианты интерпретаторов. На ранних стадиях разработки мы применяли подход, при котором вычисление выражения производилось в процессе его разбора. Такой подход достаточно просто реализуем, программа получается простой и понятной, не требует дополнительных накладных расходов, все вычисления выполняются прямо в стеке синтаксического анализатора, по окончании разбора сразу выдаётся ответ.

Но такой подход хорош лишь для случаев, когда выражение должно быть вычислено один раз. Поэтому в дальнейшем мы ввели в наш калькулятор простую стековую машину, позволившую эффективно реализовать управляющие конструкции и многократное вычисление выражений в цикле. Многие из предлагаемых выше дополнений к возможностям калькулятора относительно легко реализуемы только с применением стековой машины (или реализаций, рассмотренных далее).

К отличительным особенностям интерпретатора относится необходимость «читать» исходную программу всякий раз, когда возникает необходимость в её исполнении.

11.2.2. Транслятор на P-код

Транслятор на P-код, также как и интерпретатор, выполняет преобразование исходной программы в некоторую внутреннюю форму представления. Отличие от интерпретатора состоит в том, что транслятор на P-код использует в качестве внутренней формы представления программы систему команд некоторой гипотетической ЭВМ, называемой **псевдомашиной** (отсюда и название — P-код). Полученная в результате программа в P-кодах затем выполняется специальным интерпретатором.

Достоинством такого подхода является относительно лёгкая переносимость готового программного обеспечения на другие типы ЭВМ. Достаточно обеспечить на всех типах ЭВМ интерпретаторы P-кода, «понимающие» один набор команд псевдомашины. Реализация такого интерпретатора для каждого типа ЭВМ является более простой задачей, чем реализация на каждой ЭВМ транслятора в её объектный код.

Программы, оттранслированные на P-код, обычно намного короче, чем аналогичные им программы в объектных кодах, и часто короче, чем исходные программы. Это может быть особенно важно для ЭВМ с ограниченным объёмом памяти.

Система команд псевдомашины может быть подобрана так, чтобы обеспечить выполнение некоторых высокоуровневых операций: выборка из массива по индексу, передача управления подпрограмме и возврат из подпрограммы, копирование сложных структур данных и др. Кроме того, хорошо продуманная система команд может «сгладить» некоторые аппаратные различия между разными ЭВМ (например, способы вывода графики на экран) и тем самым упростить перенос программы с одной ЭВМ на другую.

Недостатком P-кода является необходимость его интерпретации, что снижает скорость выполнения программы по сравнению с объектным кодом конкретной ЭВМ.

Реализация нашего калькулятора как транслятора на P-код и интерпретатора к нему может стать хорошим упражнением для студентов как будущих системных программистов.

11.2.3. Компилятор в объектный код

В этом случае результатом трансляции является модуль в объектном коде ЭВМ, который может быть загружен в оперативную память и выполнен. Как известно, программы в объектном коде обладают наибольшей скоростью работы. Однако достаточно сложно создать компилятор, порождающий объектный код, который сразу можно загружать в память и исполнять. Обычно результатом трансляции является объектный код, содержащий вызовы стандартных системных подпрограмм, обеспечивающих взаимодействие исполняемой программы с операционной системой и аппаратурой ЭВМ. В этом случае объектный код, создаваемый компилятором, должен удовлетворять некоторому стандарту, позволяющему обрабатывать его редактором связей для получения исполняемой программы. Кроме того, он должен быть совместим с необходимыми системными библиотеками.

Конечно, такой способ реализации нашего калькулятора может быть полезен для студентов в качестве упражнения. Однако, памятуя о том, что изначально мы создавали интерактивное (!) вычислительное средство, необходимо признать, что с практической точки зрения наш калькулятор реализовывать как компилятор невыгодно.

Заключение

В настоящем учебном пособии на примере создания интерактивной программы, предназначенной для выполнения вычислений значений сложных выражений, показано использование инструментальных средств программиста, предоставляемых ОС UNIX, для ускорения создания сложных языковых программ и улучшения их качества.

Основной идеей пособия является стремление показать студентам, как, используя инструментальные средства программиста и двигаясь небольшими шагами от простого к сложному, можно создавать качественные и удобные в использовании программы.

Особенно необходимо подчеркнуть, что созданный в книге калькулятор, ориентированный на использование командной строки и стандартных потоков ввода и вывода, значительно превосходит как по вычислительной мощи, так и по удобству использования своих экранных «собратьев».

Описанные здесь методы программирования могут быть с успехом использованы будущими инженерами для разработки любых программ, оперирующих со входными языками, будь то языки программирования вычислений или непроцедурные языки конфигурирования систем.

Приложение 1
Полный исходный текст избранных вариантов калькулятора

В этом приложении даётся полный исходный текст двух окончательных вариантов нашего калькулятора: `calc4a` и `calc6`. Первый из них — это окончательный вариант калькулятора без простой машины. Второй — полный вариант калькулятора, разработанный в настоящем учебном пособии, без учёта доработок, предлагаемых в упражнениях.

П1.1. Распечатка исходного текста `calc4a`

```
********************************************** syntax.y
%{
#include <math.h>
#include "names.h"
%}

%union{
double data;    /* числовое значение */
Names *name;    /* указатель на имя */
int index;  /* номер функции в таблице функций */
}

%token <data> DATA
%token <name> VAR
%token <index> FUN
%type <data> wyrag
%type <data> prisw
%right '='
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok prisw '\n'
  | spisok wyrag '\n'   { printf("\t%g\n",$2); }
```

```
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
prisw : VAR '=' wyrag   { $$=$1->val=$3;
              $1->def=1;
            }
  | VAR '=' prisw   { $$=$1->val=$3;
              $1->def=1;
            }
wyrag : DATA        { $$=$1; }
  | VAR         { if($1->def)
                $$=$1->val;
              else
                calc_error(7,$1->name);
            }
  | FUN '(' wyrag ')'   { $$=val_fun($1,$3); }
  | wyrag '+' wyrag { $$=$1+$3; }
  | wyrag '-' wyrag { $$=$1-$3; }
  | wyrag '*' wyrag { $$=$1*$3; }
  | wyrag '/' wyrag {
                if($3==0.)
                    calc_error(1);
                $$=$1/$3;
            }
  | '(' wyrag ')'   { $$=$2; }
  | wyrag '^' wyrag { $$=pow($1,$3); }
  | '-' wyrag %prec UNARYMINUS { $$=-$2; }
  ;
%%
```

```
********************************************** lexical.l
%START PROG COMM
%{
#include <stdio.h>
#include "names.h"
#include "y.tab.h"

extern unsigned int nline;

long int idata;
%}
BUKVA   [a-zA-Z]
CIFRA   [0-9]
IMJA    {BUKVA}({BUKVA}|{CIFRA}|"_")*
DCHISLO {CIFRA}+
OCHISLO [0-7]+[Oo]
SCHISLO {CIFRA}({CIFRA}|[AaBbCcDdEeFf])*[Hh]
BCHISLO [01]+[Bb]
FCHISLOe    {CIFRA}*\.{CIFRA}*(e(\+|-)?{CIFRA}+)?
FCHISLOE    {CIFRA}*\.{CIFRA}*(E(\+|-)?{CIFRA}+)?
%%
    {
    BEGIN PROG;
    }
<PROG>"+"|"-"|"*"|"/"|"("|")"|"^"|"="   {
    return *yytext;
    }
<PROG>\t|" "    {
        ;
    }
<PROG>\n    {
    nline++;
    return *yytext;
    }
<PROG>{IMJA}    {
    if((yylval.index=search_fun(yytext))!=-1)
        return FUN;
    yylval.name=search_name(yytext);
    if(yylval.name==Nulln)
        yylval.name=add_name(yytext);
    return VAR;
```

```
    }
<PROG>{DCHISLO} {
    sscanf(yytext,"%ld",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{OCHISLO} {
    sscanf(yytext,"%lo",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{SCHISLO} {
    sscanf(yytext,"%lx",&idata);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{BCHISLO} {
    getbin(&idata,yytext);
    yylval.data=(double)idata;
    return DATA;
    }
<PROG>{FCHISLOe}    {
    sscanf(yytext,"%lg",&(yylval.data));
    return DATA;
    }
<PROG>{FCHISLOE}    {
    sscanf(yytext,"%lG",&(yylval.data));
    return DATA;
    }
<PROG>"/*"  {   /* Комментарий */
    BEGIN COMM;
    }
<COMM>"*/"  {
    BEGIN PROG;
    }
<COMM>.     {
    ;
    }
<COMM>\n    {
    nline++;
    }
```

```
<PROG>.     {   /* Нераспознанная лексема */
    resynch();
    calc_error(3,yytext);
    }
%%

yywrap()
{
    return 1;
}

resynch()
{
    int c;

    do{
        c=getchar();
    }while(c!='\n');

    ungetc(c,stdin);
}
```

********** calc4a.c

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

void fpeerr();

jmp_buf jmp_env;

unsigned int nline=1;   /* счётчик строк */

main()
{
    init_names();

    signal(SIGFPE,fpeerr);
    if(setjmp(jmp_env));
    yyparse();
}
```

```
void fpeerr()
{
    calc_error(2);
}

yyerror()
{
    fprintf(stderr,
    "\nОшибка! Строка %u\n(00) Синтаксическая ошибка\n",
                        nline);
}

char *ms_calc_error[]={
/* 0 */ "",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного",
/* 3 */ "Нераспознанная лексема %s",
/* 4 */ "Ошибка в записи бинарной константы %s",
/* 5 */ "Бинарная константа больше 32 знаков",
/* 6 */ "Переполнение доступной оперативной памяти",
/* 7 */ "Значение имени %s не определено"
};

calc_error(num,s)
int num;
char *s;
{
    fprintf(stderr,"\nОшибка! Строка %u\n(%02d) ",
                        nline,num);
    fprintf(stderr,ms_calc_error[num],s);
    fprintf(stderr,"\n");
    longjmp(jmp_env,1);
}
```

********** getbin.c

```
getbin(bin,str)
char *str;
long int *bin;
{
    long int rez;
    int i;

    rez=0;
    for(i=0;i<32 && *str;i++){
        rez<<=1;
        switch(*str){
        case '0' :
            break;
        case '1' :
            rez|=1;
            break;
        case 'b' :
        case 'B' :
            rez>>=1;
            *str='\0';
            goto endbin;
        default :
            calc_error(4,str);
            break;
        }
        str++;
    }
endbin:
    if(i==32 && *str!='\0')
        calc_error(5);

    *bin=rez;
}
```

```
********************************************** names.h
#define MAXLEN  20        /* максимальная длина имени */
#define Nulln   (Names*)0 /* значение пустого указателя */

typedef struct Ns{
char name[MAXLEN+1];    /* имя переменной */
double val;     /* значение переменной */
int def;    /* если =1, то имени уже присвоено значение */
struct Ns *sled;    /* указатель на следующую запись */
}Names;


Names* add_name();
Names* search_name();

double val_fun();
********************************************** names.c
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "names.h"

Names *names=Nulln; /* указатель на начало списка имён */

Names* malloc_name()
{
    Names *p;

    if((p=(Names*)malloc(sizeof(Names)))==NULL){
        calc_error(6);
    }

    return p;
}

Names* add_name(name)
char *name;
{
    Names *p;
```

```
    p=malloc_name();

    p->sled=names;
    names=p;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* принудительное ограничение
                               длины имени */
    strcpy(p->name,name);
    p->def=0;       /* значение ещё не присвоено */

    return p;
}

Names* search_name(name)
char *name;
{
    Names *p;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* принудительное ограничение
                               длины имени */

    for(p=names;p!=Nulln;p=p->sled){
        if(strcmp(p->name,name)==0)
            break;  /* имя найдено */
    }

    return p;
}

/* таблица констант */
struct Tn{
char name[MAXLEN+1];    /* имя константы */
double val;     /* значение */
}Tnames[]={
"Pi",3.14159,       /* число Пи */
"E",2.718282,       /* основание натурального логарифма */
"DEG",57.2958,      /* отношение радиана к градусу */
"PHI",1.61803,      /* золотое сечение */
};
```

```
int len_Tnames=sizeof(Tnames)/sizeof(struct Tn);

init_names()
{
    Names *p;
    int lentab;
    int i;

    for(i=0;i<len_Tnames;i++){
        p=add_name(Tnames[i].name);
        p->val=Tnames[i].val;
        p->def=1;
    }
}

/* таблица математических функций */
struct Tf{
char name[MAXLEN+1];    /* имя функции */
double (*fun)();    /* указатель на функцию */
}Tablf[]={
"sin",sin,
"cos",cos,
"tg",tan,
};

int len_Tablf=sizeof(Tablf)/sizeof(struct Tf);

search_fun(name)
char *name;
{
    int i;

    if(strlen(name)>MAXLEN)
        name[MAXLEN]='\0';  /* принудительное ограничение
                               длины имени */

    for(i=0;i<len_Tablf;i++){
        if(strcmp(name,Tablf[i].name)==0)
            return i;   /* функция найдена */
    }
```

```
    return -1;  /* поиск неудачен */
}

double val_fun(index,arg)
int index;
double arg;
{
    return (*Tablf[index].fun)(arg);
}
```

********** makefile

```
PROG = calc4a
YFLAGS = -d

$(PROG) : $(PROG).o syntax.o lexical.o getbin.o names.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o \
                    getbin.o names.o

lexical.o : y.tab.h names.h
syntax.o : names.h
```

П1.2. Распечатка исходного текста calc6

```
********************************************** syntax.y
%{
#include <math.h>
#include "names.h"
#include "code.h"
%}

%union{
Names *name;     /* указатель на имя */
int index;  /* номер функции в таблице функций */
Prog *cprog;     /* команда программы */
}

%token <name> DATA
%token <name> VAR
%token <index> FUN
%token <name> PRINT WHILE
%type <cprog> wyrag oper operspis while prisw
%right '='
%left GT GE LT LE EQ NE
%left '+' '-'
%left '*' '/'
%left '^'
%left UNARYMINUS
%%

spisok :    /* пусто */
  | spisok '\n'
  | spisok prisw '\n'   {   code(mpop);
                code(STOP);
                return 1;
            }
  | spisok wyrag '\n'   {   code(print);
                code(STOP);
                return 1;
            }
  | spisok oper '\n'    {   code(STOP);
                return 1;
            }
```

```
  | spisok error    {
                resynch();
                yyclearin;
                yyerrok;
            }
  ;
oper : prisw        { code(mpop); }
  | wyrag       { code(mpop); }
  | PRINT wyrag     { code(pri); $$=$2; }
  | while '(' wyrag ')' { code(STOP); } oper    {
                code(STOP);
                pc=$1;
                pc[1]=(Prog)$6; /* тело цикла */
                pc[2]=(Prog)pprog;  /* конец цикла */
            }
  | '{' operspis '}'    { $$=$2; }
  ;
while : WHILE       {   $$=code(fwhile);
                code(STOP);
                code(STOP);
            }
operspis : /* пусто */  { $$=pprog; }
  | operspis '\n'
  | operspis oper
  ;
prisw : VAR '=' wyrag   { code(assign); code((Prog)$1);
                          $$=$3; }
  | VAR '=' prisw   { code(assign); code((Prog)$1);
                          $$=$3; }
  ;
wyrag : DATA        { $$=code(pushconst); code((Prog)$1); }
  | VAR         { $$=code(eval); code((Prog)$1); }
  | FUN '(' wyrag ')'   { code(fval); code((Prog)$1);
                          $$=$3; }
  | wyrag '+' wyrag { code(add); }
  | wyrag '-' wyrag { code(sub); }
  | wyrag '*' wyrag { code(mul); }
  | wyrag '/' wyrag { code(calcdiv); }
  | '(' wyrag ')'   { $$=$2; }
  | wyrag '^' wyrag { code(power); }
  | '-' wyrag %prec UNARYMINUS { code(negate); $$=$2; }
```

```
   | wyrag GT wyrag  { code(gt); }
   | wyrag GE wyrag  { code(ge); }
   | wyrag LT wyrag  { code(lt); }
   | wyrag LE wyrag  { code(le); }
   | wyrag EQ wyrag  { code(eq); }
   | wyrag NE wyrag  { code(ne); }
   ;
%%
```

********** lexical.l

```
%START PROG COMM
%{
#include <stdio.h>
#include "names.h"
#include "code.h"
#include "y.tab.h"

extern unsigned int nline;

long int idata;
double data;
%}
BUKVA   [a-zA-Z]
CIFRA   [0-9]
IMJA    {BUKVA}({BUKVA}|{CIFRA}|"_")*
DCHISLO {CIFRA}+
OCHISLO [0-7]+[Oo]
SCHISLO {CIFRA}({CIFRA}|[AaBbCcDdEeFf])*[Hh]
BCHISLO [01]+[Bb]
FCHISLOe    {CIFRA}*\.{CIFRA}*(e(\+|-)?{CIFRA}+)?
FCHISLOE    {CIFRA}*\.{CIFRA}*(E(\+|-)?{CIFRA}+)?
%%
    {
    BEGIN PROG;
    }
<PROG>">"   {
    return GT;
    }
<PROG>">="  {
    return GE;
    }
```

```
<PROG>"<"   {
    return LT;
    }
<PROG>"<="  {
    return LE;
    }
<PROG>"=="  {
    return EQ;
    }
<PROG>"!="  {
    return NE;
    }
<PROG>"+"|"-"|"*"|"/"|"("|")"|"^"|"="|"{"|"}"   {
    return *yytext;
    }
<PROG>\t|" "    {
        ;
    }
<PROG>\n    {
    nline++;
    return *yytext;
    }
<PROG>{IMJA}    {
    if((idata=search_kword(yytext))!=-1)
        return idata;
    if((yylval.index=search_fun(yytext))!=-1)
        return FUN;
    yylval.name=search_name(yytext);
    if(yylval.name==Nulln)
        yylval.name=add_name(yytext);
    return VAR;
    }
<PROG>{DCHISLO} {
    sscanf(yytext,"%ld",&idata);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{OCHISLO} {
    sscanf(yytext,"%lo",&idata);
    data=(double)idata;
```

```
    make_const(data);
    return DATA;
    }
<PROG>{SCHISLO} {
    sscanf(yytext,"%lx",&idata);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{BCHISLO} {
    getbin(&idata,yytext);
    data=(double)idata;
    make_const(data);
    return DATA;
    }
<PROG>{FCHISLOe}    {
    sscanf(yytext,"%lg",&data);
    make_const(data);
    return DATA;
    }
<PROG>{FCHISLOE}    {
    sscanf(yytext,"%lG",&data);
    make_const(data);
    return DATA;
    }
<PROG>"/*"  {   /* Комментарий */
    BEGIN COMM;
    }
<COMM>"*/"  {
    BEGIN PROG;
    }
<COMM>.     {
    ;
    }
<COMM>\n    {
    nline++;
    }
<PROG>.     {   /* Нераспознанная лексема */
    resynch();
    calc_error(3,yytext);
    }
```

```
%%

yywrap()
{
    return 1;
}

resynch()
{
    int c;

    do{
        c=getchar();
    }while(c!='\n');

    ungetc(c,stdin);
}

make_const(data)
double data;
{
    yylval.name=add_name("");
    (yylval.name)->val=data;
}
```

********** calc6.c

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>

void fpeerr();

jmp_buf jmp_env;

unsigned int nline=1;   /* счётчик строк */

main()
{
    init_names();

    signal(SIGFPE,fpeerr);
```

```
    if(setjmp(jmp_env));
    for(code_init();yyparse();code_init()){
        execute();
    }
}


void fpeerr()
{
    calc_error(2);
}

yyerror()
{
    fprintf(stderr,
    "\nОшибка! Строка %u\n(00) Синтаксическая ошибка\n",
                        nline);
}

char *ms_calc_error[]={
/* 0 */ "",
/* 1 */ "Деление на 0",
/* 2 */ "Переполнение вещественного",
/* 3 */ "Нераспознанная лексема %s",
/* 4 */ "Ошибка в записи бинарной константы %s",
/* 5 */ "Бинарная константа больше 32 знаков",
/* 6 */ "Переполнение доступной оперативной памяти",
/* 7 */ "Значение имени %s не определено",
/* 8 */ "Переполнение стека",
/* 9 */ "Опустошение стека",
/* 10 */ "Переполнение памяти программы"
};

calc_error(num,s)
int num;
char *s;
{
    fprintf(stderr,"\nОшибка! Строка %u\n(%02d) ",
                        nline,num);
    fprintf(stderr,ms_calc_error[num],s);
    fprintf(stderr,"\n");
    longjmp(jmp_env,1);
}
```

```
********************************************** code.h
#define MAXSTACK      2000    /* размер стека машины */
#define MAXPROG     2000    /* размер программы машины */

#define STOP    (Prog) 0

typedef double Stack;   /* тип данных стека */

typedef int (*Prog)();  /* тип данных программы */

extern Prog *pprog; /* указатель на первый
                       свободный элемент */
extern Prog *pc;    /* счётчик команд */

add(), sub(), mul(), calcdiv(), power(), negate();
assign(), pushconst(), eval(), fval(), print();
mpop();
Stack pop();
pri(), fwhile();
gt(), ge(), lt(), le(), eq(), ne();

Prog* code();
********************************************** code.c
#include <stdio.h>
#include <math.h>
#include "names.h"
#include "code.h"

Stack stack[MAXSTACK];  /* стек машины */

Stack *pstack;      /* указатель на вершину стека */

Prog prog[MAXPROG]; /* массив программы машины */

Prog *pprog;        /* указатель на первый
                       свободный элемент */
Prog *pc;       /* счётчик команд */

code_init()
{
```

```
    pprog=prog;
    pstack=stack;
}

Prog* code(f)
Prog f;
{
    Prog *oprog;

    oprog=pprog;
    if(pprog>=&prog[MAXPROG])
        calc_error(10);
    *pprog++=f;

    return oprog;
}

execute()
{
    rexecute(prog);
}

rexecute(prg)
Prog *prg;
{
    for(pc=prg;*pc!=STOP;)
        (*pc++)();
}

push(d)
double d;
{
    if(pstack>=&stack[MAXSTACK])
        calc_error(8);
    *pstack++=d;
}

Stack pop()
{
    if(pstack<=stack)
        calc_error(9);
```

```
    return *--pstack;
}

mpop()
{
    if(pstack<=stack)
        calc_error(9);
    --pstack;
}

add()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1+=d2;
    push(d1);
}

sub()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1-=d2;
    push(d1);
}

mul()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1*=d2;
    push(d1);
}

calcdiv()
```

```
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d2==0.)
        calc_error(1);
    d1/=d2;
    push(d1);
}

power()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    d1=pow(d1,d2);
    push(d1);
}

negate()
{
    Stack d;

    d=pop();
    d=-d;
    push(d);
}

assign()
{
    Stack d;
    Names *pn;

    d=pop();
    pn=(Names*)(*pc++);
    pn->val=d;
    pn->def=1;
    push(d);
}
```

```
eval()
{
    Stack d;
    Names *pn;

    pn=(Names*)(*pc++);
    if(pn->def==0)
        calc_error(7,pn->name);

    d=pn->val;
    push(d);
}

pushconst()
{
    Names *pn;

    pn=(Names*)(*pc++);
    push((Stack)(pn->val));
}

fval()
{
    Stack d;

    d=pop();
    d=val_fun((int)(*pc++),d);
    push(d);
}

print()
{
    Stack d;

    d=pop();
    printf("\t%g\n",d);
}

pri()
{
```

```
    Stack d;

    d=pop();
    printf("\t%g",d);
}

gt()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1>d2)
        push(1.);
    else
        push(0.);
}

ge()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1>=d2)
        push(1.);
    else
        push(0.);
}

lt()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1<d2)
        push(1.);
    else
        push(0.);
}
```

```
le()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1<=d2)
        push(1.);
    else
        push(0.);
}

eq()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1==d2)
        push(1.);
    else
        push(0.);
}

ne()
{
    Stack d1,d2;

    d2=pop();
    d1=pop();
    if(d1!=d2)
        push(1.);
    else
        push(0.);
}

fwhile()
{
    Stack d;
    Prog *opc;
```

```
    opc=pc;          /* тело цикла */
    rexecute(opc+2);    /* условие цикла */
    d=pop();
    while(d!=0.){
        rexecute(*opc);
        rexecute(opc+2);
        d=pop();
    }
    pc=*((Prog**)opc+1);    /* первый оператор за циклом */
}
```

********** getbin.c

```
getbin(bin,str)
char *str;
long int *bin;
{
    long int rez;
    int i;

    rez=0;
    for(i=0;i<32 && *str;i++){
        rez<<=1;
        switch(*str){
        case '0' :
            break;
        case '1' :
            rez|=1;
            break;
        case 'b' :
        case 'B' :
            rez>>=1;
            *str='\0';
            goto endbin;
        default :
            calc_error(4,str);
            break;
        }
        str++;
    }
endbin:
```

```
    if(i==32 && *str!='\0')
        calc_error(5);

    *bin=rez;
}
```

********** names.h

```
#define MAXLEN  20        /* максимальная длина имени */
#define Nulln   (Names*)0 /* значение пустого указателя */

typedef struct Ns{
char name[MAXLEN+1];    /* имя переменной */
double val;     /* значение переменной */
int def;    /* если =1, то имени уже присвоено значение */
struct Ns *sled;    /* указатель на следующую запись */
}Names;


Names* add_name();
Names* search_name();

double val_fun();
```

********** names.c

```
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "names.h"
#include "code.h"
#include "y.tab.h"

Names *names=Nulln; /* указатель на начало списка имён */

Names* malloc_name()
{
    Names *p;

    if((p=(Names*)malloc(sizeof(Names)))==NULL){
        calc_error(6);
    }
```

```
    return p;
}

Names* add_name(name)
char *name;
{
    Names *p;

    p=malloc_name();

    p->sled=names;
    names=p;

    if(strlen(name)>=MAXLEN){
        /* принудительное ограничение длины имени */
        name[MAXLEN]='\0';
    }
    strcpy(p->name,name);
    p->def=0;       /* значение ещё не присвоено */

    return p;
}

Names* search_name(name)
char *name;
{
    Names *p;

    if(strlen(name)>=MAXLEN){
        /* принудительное ограничение длины имени */
        name[MAXLEN]='\0';
    }

    for(p=names;p!=Nulln;p=p->sled){
        if(strcmp(p->name,name)==0)
            break;  /* имя найдено */
    }

    return p;
}
```

```
/* таблица констант */
struct Tn{
char name[MAXLEN+1];    /* имя константы */
double val;     /* значение */
}Tnames[]={
"Pi",3.14159,       /* число Пи */
"E",2.718282,       /* основание натурального логарифма */
"DEG",57.2958,      /* отношение радиана к градусу */
"PHI",1.61803,      /* золотое сечение */
};

int len_Tnames=sizeof(Tnames)/sizeof(struct Tn);

init_names()
{
    Names *p;
    int lentab;
    int i;

    for(i=0;i<len_Tnames;i++){
        p=add_name(Tnames[i].name);
        p->val=Tnames[i].val;
        p->def=1;
    }
}

/* таблица математических функций */
struct Tf{
char name[MAXLEN+1];    /* имя функции */
double (*fun)();    /* указатель на функцию */
}Tablf[]={
"sin",sin,
"cos",cos,
"tg",tan,
};

int len_Tablf=sizeof(Tablf)/sizeof(struct Tf);

search_fun(name)
char *name;
{
```

```
    int i;

    if(strlen(name)>=MAXLEN){
        /* принудительное ограничение длины имени */
        name[MAXLEN]='\0';
    }

    for(i=0;i<len_Tablf;i++){
        if(strcmp(name,Tablf[i].name)==0)
            return i;   /* функция найдена */
    }

    return -1;  /* поиск неудачен */
}

double val_fun(index,arg)
int index;
double arg;
{
    return (*Tablf[index].fun)(arg);
}

/* таблица ключевых слов */
struct Tw{
char name[MAXLEN+1];    /* слово */
int term;       /* номер терминала */
}Tablword[]={
"print",PRINT,
"while",WHILE
};

int len_Tablword=sizeof(Tablword)/sizeof(struct Tw);

search_kword(name)
char *name;
{
    int i;

    if(strlen(name)>=MAXLEN){
        /* принудительное ограничение длины имени */
        name[MAXLEN]='\0';
```

```
    }

    for(i=0;i<len_Tablword;i++){
        if(strcmp(name,Tablword[i].name)==0)
            return Tablword[i].term;    /* слово найдено */
    }

    return -1;  /* поиск неудачен */
}
```

********** makefile

```
PROG = calc6
YFLAGS = -d
CFLAGS = -g

$(PROG) : $(PROG).o syntax.o lexical.o getbin.o names.o \
                                                    code.o
    cc -lm -o $(PROG) $(PROG).o syntax.o lexical.o \
            getbin.o names.o code.o

lexical.o : y.tab.h names.h code.h
syntax.o : names.h code.h
names.o : y.tab.h code.h
code.o : names.h
```

Приложение 2
Краткий справочник по `calc6`

Программа `calc6` — это диалоговый интерпретатор для вычисления выражений с плавающей точкой. Он воспринимает ввод исходной программы вычислений из стандартного потока ввода. Вывод результатов осуществляется на стандартный поток вывода.

Выражения

Выражение представляет собой последовательность чисел, имён переменных и функций, объединённых символами операций. Грамматика выражений имеет вид

```
выражение : число
     | переменная
     | (выражение)
     | выражение бинарная_операция выражение
     | унарный_минус выражение
     | функция (аргумент)
```

Числа представляются либо с плавающей точкой, либо целые с постфиксным указанием системы счисления (`h` — шестнадцатеричное; `o` — восьмеричное; `b` — бинарное). Если система счисления не указана, предполагается десятичная система счисления. Во внутреннем представлении все числа хранятся в вещественном формате. Числа обязательно начинаются с цифры.

Имена переменных начинаются с буквы, за которой следует произвольное число букв, цифр и знаков подчёркивания. Идентификация имён переменных ведётся по первым 20 символам.

Бинарные_операции — это обычные арифметические операции, операция возведения в степень, а также операции отношения.

Операции по порядку уменьшения приоритета

Все операции, кроме специально оговоренных, левоассоциативны.

`-` Унарный минус
`^` Возведение в степень
`* /` Умножение, деление
`+ -` Сложение, вычитание
`> >= < <= == !=` Операции отношения
`=` Присваивание, правоассоциативна

Встроенные функции

`sin(x)` Синус x
`cos(x)` Косинус x
`tg(x)` Тангенс x
Аргумент x задаётся в радианах.

Встроенные константы

`Pi` Круговое трансцендентное число π
`E` Основание натурального логарифма e
`DEG` Отношение $180°/\pi$
`PHI` Золотое сечение $(\sqrt{5}+1)/2$

Управляющие структуры

Интерпретатор допускает использование цикла `while` и фигурных скобок группирования операторов. Ниже показан пример использования цикла:

```
a=0
while(a<10){
    print a
    a=a+1
}
```

Комментарии

Любой текст, заключённый между парами символов `/*` и `*/`, считается комментарием. Один комментарий может включать несколько строк текста. Вложенные комментарии не допускаются.

Пример использования

Вычислим определённый интеграл

$$y = \int_0^{2\pi} \sin x \, dx$$

модифицированным методом трапеций. Как известно, значение такого интеграла равно нулю. Проверим это на калькуляторе:

```
int=0
di=2*Pi/40
i=di
while(i<2*Pi){
    int=int+2*di*sin(i)
    i=i+2*di
```

```
}
print int
    1.41383e-11
```

С учётом погрешности метода можно считать, что результат вычислений на калькуляторе соответствует действительности.

Сообщения об ошибках

Приводятся коды сообщений об ошибках и их расшифровка. В необходимых случаях даются разъяснения.

0 Синтаксическая ошибка

1 Деление на 0 — при вычислении выражения обнаружено деление на ноль

2 Переполнение вещественного — размер результата выражения превышает возможности разрядной сетки

3 Нераспознанная лексема ЛЕКСЕМА

4 Ошибка в записи бинарной константы КОНСТАНТА — бинарная константа должна содержать только цифры `0` и `1`

5 Бинарная константа больше 32 знаков — калькулятор допускает ввод бинарных констант размером не более 32 знаков

6 Переполнение доступной оперативной памяти — невозможно выделить память для хранения именованной переменной или константы

7 Значение имени ИМЯ не определено — переменной с именем ИМЯ не было присвоено значение

8 Переполнение стека — для вычисления выражения потребовался размер стека, превышающий определённый в программе

9 Опустошение стека — сообщение говорит о внутренней ошибке калькулятора; в норме возникать не должно

10 Переполнение памяти программы — полученная при трансляции машинная программа превышает объём выделенной для её хранения памяти

Ошибки 8 и 10 могут быть устранены перекомпиляцией программы `calc6` с увеличенными значениями констант `MAXSTACK` и `MAXPROG` в файле `code.h`.

Приложение 3
Задания на самостоятельную доработку калькулятора

К главе 5

Для `calc1a`

1. Добавьте операцию унарный плюс.

2. Добавьте операцию взятие остатка (%). Помните, что эта операция применима только к целым числам.

Для `calc1b`

3. Добавьте обработку ошибок математических функций библиотеки Си. Для этого посмотрите документацию на функцию `matherr()`.

4. Доработайте алгоритм вызова функций обработки ошибок так, чтобы сохранить единство вызова процедуры печати сообщения об ошибках.

5. Доработайте функцию лексического анализатора для правильного подсчёта обработанных строк. Не забудьте изменить алгоритм работы функции `resynch()`.

Для `calc2`

6. Обеспечьте начальную инициализацию регистровых переменных каким-либо значением.

7. Реализуйте механизм контроля за определённостью регистровых переменных, чтобы не разрешать пользователю использовать в выражениях имена регистров, для которых ещё не выполнялась операция присваивания.

8. Обеспечьте возможность запоминания последнего вычисленного значения, чтобы его не приходилось вводить снова для последовательности связанных вычислений. Одним из решений может быть использование какой-либо переменной, например `p`, в качестве «предыдущего» (previous) значения.

К главе 7

Для `calc3a`

9. Доработайте лексический анализатор, чтобы он допускал вложенные комментарии.

10. Предложите способ печатать результат вычислений в задаваемой пользователем системе счисления.

11. Добавьте в калькулятор поразрядные операции И, ИЛИ, НЕ, сложение по модулю 2. Не забудьте, что стек калькулятора хранит вещественные числа, а поразрядные операции полезны лишь для целых. Одним из вариантов может быть выполнение поразрядных операций с целыми частями чисел.

К главе 8

Для `calc4`

12. Предложите способ восстановления в строке ввода предыдущего выражения с возможностью его редактирования, чтобы не приходилось, например, повторно вводить сложное выражение из-за случайной ошибки.

13. Предоставьте пользователю возможность просматривать список определённых в процессе работы имён переменных.

Для `calc4a`

14. В данной версии калькулятора допустимо присваивание предопределённым константам новых значений. Как бы Вы доработали калькулятор, чтобы запретить эту возможность?

15. Доработайте функцию реализации математических функций, чтобы она выполняла проверку ошибок работы математических функций.

16. Добавьте в калькулятор вызов встроенной функции `atan2(x,y)` для вычисления величины угла, тангенс которого равен y/x. Добавьте встроенную функцию `rand()`, вырабатывающую случайные вещественные числа в интервале $(0, 1)$. Как необходимо изменить грамматику, чтобы разрешить использование встроенных функций с разным числом аргументов?

К главе 9

Для `calc5`

17. При выбранном способе хранения констант, распознанных в вычисляемом выражении, память постепенно будет «захламляться» ненужными ячейками памяти, не имеющими имени. Предложите способ очистки памяти от этих безымянных записей.

18. Сделайте размеры стека и программной памяти динамическими, чтобы для `calc5` всегда хватало объёма памяти, если только её можно получить.

19. Измените `calc5` так, чтобы использовать в функции `execute()` вместо вызова функций по указателю переключатель по виду операции. Каково соотношение версий по размеру исходного текста и по времени выполнения? Как приблизительно их сопоставить по сложности развития и поддержки?

К главе 10

Для `calc6`

20. Добавьте в калькулятор операторы `if then else`, `for`, `break` и `continue`.

21. Добавьте в калькулятор логические выражения `&&` и `||`.

22. Добавьте для отладки к калькулятору средство печати создаваемых машинных команд в понятной (читабельной) форме.

23. Введите в калькулятор операции присваивания из языка Си `+=`, `-=` и т. п., а также операции инкремента и декремента `++` и `--`.

24. Как бы Вы ввели массивы в язык калькулятора?

25. Доработайте оператор `print`, чтобы он позволял печатать результат вычислений в задаваемой пользователем форме.

К главе 11

26. Реализуйте одно из возможных дополнений к калькулятору, предложенное в главе 11.

Приложение 4
Краткая историческая справка

Операционная система UNIX была создана в 1969 г. для машины фирмы DEC PDP-7. Считается, что система была разработана К. Томпсоном при поддержке Р. Канадея, Д. МакИлроя, Д. Осанна и Д. Ритчи. В 1970 г. система с помощью Д. Ритчи была перенесена на PDP-11. Д. Ритчи также разработал и написал компилятор с языка программирования Си. В 1973 г. Д. Ритчи и К. Томпсон переписали ядро ОС UNIX на языке Си, что обеспечило системе возможность переноса на другие платформы и широкий диапазон применения. Большую роль в популяризации ОС UNIX и языка программирования Си сыграл Б. Керниган.

Генератор YACC создан С. Джонсоном в 1972 г. Вместе с этим генератором появилось понятие раздельных описаний для задания приоритетов и разрешения неоднозначностей в грамматике. Буквально YACC расшифровывается как «yet another compiler–compiler» (ещё один компилятор компиляторов). Такое странное название обусловлено тем, что во времена его разработки существовало уже довольно большое число подобных программ. Однако YACC оказался одним из немногих, получивших признание. Это связано прежде всего с использованием его как инструментального средства в составе ОС UNIX.

Генератор LEX первоначально был написан М. Леском. На практике он используется гораздо меньше, чем YACC, так как в большинстве случаев лексический анализатор, написанный вручную, оказывается много компактнее и эффективнее, чем созданный генератором LEX.

Генераторы YACC и в меньшей степени LEX использовались для реализации многих языковых процессоров UNIX, включая переносимый компилятор Си, процессоры на Паскале, Фортране 77, Ратфоре, awk, bc, eqn и pic.

В последнее время, благодаря распространению программных продуктов GNU, большую известность получили аналогичные по назначению генераторы BISON и FLEX.

Программа MAKE создана С. Фельдманом. В UNIX она используется чрезвычайно широко, и не только для управления трансляцией программ. Она с успехом применяется для любых работ, требующих слежения за временем модификации файлов и их зависимостями.

Документация на все описанные программы входит в состав справочного руководства по UNIX.

Приложение 5
Краткое описание команд систем программирования на языке Си ОС UNIX и ОС Linux

Целью данного приложения является упорядочение сведений о трансляции программ, написанных на языке Си, в системах программирования, ОС UNIX и ОС Linux.

В классической ОС UNIX используется система программирования на базе компилятора `cc`. В ОС Linux применяется компилятор языка Си, являющийся частью комплекта GCC (Gnu Compiler Collection — коллекция компиляторов GNU), который, помимо Си, включает компиляторы ещё для нескольких языков программирования. В существующих сегодня коммерческих ОС, поддерживающих принципы реализации UNIX, используются коммерческие реализации системы программирования Си.

С точки зрения программиста эти системы очень похожи, хотя и представляют собой разные реализации одного и того же языка.

Здесь будут сопоставлены только средства, рассмотренные в книге.

Генератор программ синтаксического разбора

В ОС UNIX этот генератор вызывается командой `yacc`, в ОС Linux — командой `bison` (команда `yacc` также присутствует, но является ссылкой на `bison`). Вторая команда запускает обновлённую версию, совместимую с предшествующей, и имеющую расширенные возможности. Команды имеют стандартный набор ключей управления.

Генератор программ лексического разбора

В ОС UNIX генератор запускается на выполнение командой `lex`, в ОС Linux — командой `flex` (команда `lex` также имеется и является ссылкой на `flex`). Последняя команда является улучшенной версией `lex` с расширенными возможностями. Команды имеют стандартный набор ключей.

Компилятор с языка Си

В ОС UNIX компилятор вызывается командой `cc`. В ОС Linux применяется свободная версия компилятора, вызываемая командой `gcc`. С точки зрения задач, рассматриваемых в книге, компиляторы совместимы по ключам управления и принципам работы. Для выполнения трансляции исходной программы в объектный код используется ключ `-c`.

Команду `cc` можно использовать и в ОС Linux. При этом будет запущен компилятор `gcc`.

Редактор связей

И в ОС UNIX, и в ОС Linux редактор связей называется `link`. Но на практике удобнее пользоваться свойством компилятора Си самостоятельно вызывать редактор связей для получения рабочей программы. По умолчанию рабочая программа помещается в файл `a.out`. Для указания явного имени рабочей программы используется ключ `-o`. Для примера покажем сборку рабочей программы из двух объектных файлов `test1.o` и `test2.o` с присвоением итоговой программе имени `test`:

```
cc -o test test1.o test2.o
```

или

```
gcc -o test test1.o test2.o
```

Утилита MAKE

В обеих рассматриваемых системах программирования утилита запускается одинаковой командой: `make`. Общий принцип её работы одинаков для обеих систем. Утилита снабжена большим числом правил получения целевых файлов из исходных по умолчанию. Поэтому файлы make-программы для этих систем программирования обычно невелики.

Однако при использовании утилиты MAKE в других ОС и системах программирования программисту необходимо быть готовым к тому, что правила по умолчанию не являются обязательной частью утилиты MAKE. Поэтому любому программисту полезно уметь самостоятельно описывать правила зависимостей на языке MAKE.

Приложение 6
Возможные источники приобретения необходимого программного обеспечения

Большинство программистов, пожелающих использовать в своей практической деятельности описанное в книге программное обеспечение, наверняка столкнутся с проблемой, которую кратко можно сформулировать так: «существуют ли варианты рассмотренных инструментальных средств программирования для той ОС, в которой они обычно работают, и как их можно приобрести?».

Прежде всего отметим, что эти средства в обязательном порядке присутствуют в любой UNIX-подобной ОС (FreeBSD, Linux, Solaris, QNX и др.). Поэтому у программистов, работающих в этих ОС, никаких проблем с получением доступа к инструментальным средствам программиста возникнуть не должно.

К сожалению, большое число программистов нашей страны вынуждены работать в других ОС, для которых само существование свободно распространяемого переносимого компилятора с языка Си является проблемой. Существующие системы программирования являются разработками коммерческих фирм, и часто существует даже проблема переносимости исходного текста программы с транслятора одной фирмы на транслятор другой. Каждая фирма стремится снабдить свою систему программирования набором уникальных утилит, каким-то образом «облегчающих» программисту разработку программы. Поразительно, что среди этих утилит практически нет утилит, прошедших многолетнюю апробацию в ОС UNIX и доказавших свою полезность. Единственным исключением, пожалуй, являются утилиты `make` и `touch`. Но и здесь приходится констатировать ограниченность и несовместимость по возможностям их реализаций разными фирмами.

Для практических целей программирования полезными будут следующие варианты. Практически все они базируются на системе программирования `GCC`, распространяемой свободно в соответствии с Лицензией GNU. Проект `GCC` первоначально был реализован на ОС семейства UNIX* и впоследствии перенесён на ОС MS DOS, OS/2, Windows 95/98/NT/2000 и другие. Все реализации совместимы по базовому набору утилит и исходному тексту транслируемых программ (в некоторых реализациях есть специализированные утилиты расширения возможностей ОС). Все реализации распространяются свободно в соответствии с Лицензией GNU. Все реализации позволяют разрабатывать программы, запускаемые из командной строки.

* Конкретную ОС автор указать затрудняется.

В отдельных реализациях присутствуют уникальные библиотеки, отражающие экранные возможности конкретной ОС. Программы, разработанные в одной из реализаций с использованием только стандартных вызовов библиотек, могут быть практически без изменений оттранслированы на любой другой реализации. Любая из описанных реализаций доступна по сети Internet.

Первая из предлагаемых реализаций носит название DJGPP и предназначена исключительно для ОС MS DOS. (Хотя первоначально предполагалось разработать систему для использования и в других ОС, эти направления проекта до сих пор не развиваются.) Её официальный сайт находится по адресу

`http://www.delorie.com/djgpp`

Эта система программирования позволяет получить рабочие программы, исполняемые на MS DOS под управлением так называемого расширителя `GO32`.

Вторая реализация известна под обобщённым названием EMX GCC (или просто EMX). Её адрес в сети:

`ftp://ftp-os2.nmsu.edu/pub/os2/dev/emx/v0.9d/`

Полезную информацию о расположении дополнительных библиотек можно получить из файла `README.DOC`, который можно найти среди файлов проекта.

Для этой реализации потребуется отдельная утилита `make`. Её придётся поискать для той ОС, в которой предполагается использовать систему программирования.

Реализация EMX позволяет получать рабочие программы, которые одинаково хорошо (без перетрансляции!) работают под управлением ОС OS/2 и MS DOS с использованием специального драйвера `emx`. При наличии специального драйвера `rsx` эти программы работоспособны и в ОС Windows 95/98/NT (другие версии Windows автором не проверялись).

Третья реализация разработана специально для оконной ОС Windows 95/NT. Она носит название «Проект Cygwin». На данный момент она единственная из всех рассмотренных продолжает активно развиваться и работает во всех существующих версиях ОС Windows. Отличительной особенностью проекта является предоставление не только стандартной системы программирования `GCC`, но и набора наиболее употребительных утилит ОС UNIX, включая интерпретатор командной строки, а также специальных библиотек поддержки графического интерфейса.

Проект имеет в сети множество зеркал, поэтому здесь приведём адрес только официальной WWW страницы:

`http://www.cygwin.com`

В заключение автор хочет предупредить читателей, что не имел возможности проверить все указанные здесь адреса Internet на существование их к моменту подготовки книги к печати. С другой стороны, как уже было отмечено, многие серверы в сети имеют свои зеркала по другим адресам. Поэтому, если какой-либо адрес не отвечает, можно попробовать найти с помощью поисковых систем адреса других серверов, содержащих нужное программное обеспечение.

Библиографический список

1. **Керниган, Б. В.** UNIX — универсальная среда программирования / Б. В. Керниган, Р. Пайк / Перевод с англ.; Предисл. М. И. Белякова. — М.: Финансы и статистика, 1992. — 304 с.

2. **Тихомиров, В. П.** Операционная система ДЕМОС: Инструментальные средства программиста. / В. П. Тихомиров, М. И. Давидов — М.: Финансы и статистика, 1988. — 206 с.

3. **Беляков, М. И.** Мобильная операционная система: Справочник. / М. И. Беляков, Ю. И. Рабовер, А. Л. Фридман — М.: Радио и связь, 1991. — 208 с.

4. **Бек, Л.** Введение в системное программирование / Перевод с англ. — М.: Мир, 1988. — 448 с.

5. **Конспект** лекций по курсу «Формальные языки и основы компиляции»*. — Ульяновск, весна 1990.

6. **Ахо, А.** Компиляторы: принципы, технологии и инструменты. / А. Ахо, Р. Сети, Д. Ульман. — М.: Издательский дом «Вильямс», 2001. — 768 с.

7. **Фридл, Дж.** Регулярные выражения. Библиотека программиста. — СПб.: Питер, 2001. — 352 с.

* Конспект был получен ориентировочно в 1993 г. через BBS в электронном виде. Установить его точное авторство в настоящий момент не представляется возможным. По некоторым косвенным признакам авторами являются Д. В. Варсанофьев и А. Г. Дымченко.

Алфавитный указатель

Указатель охватывает все главы учебного пособия, а также введение, заключение, все приложения и частично список литературы. Термины, обозначаемые английскими буквами, приведены в упорядоченном порядке в конце указателя, причём сначала даются составные понятия, например make-программа, а затем чисто английские, например YACC. При упорядочивании английской части указателя не делается различия между прописными и строчными буквами. Кроме того, не учитываются символы процента (%), предваряющие слова-директивы (например `%left`).

В силу технических особенностей формата вёрстки ссылки на директивы, операторы, ключевые слова и т. п. в листингах программ, приведённых в книге, могут отличаться от указанных на одну страницу вперёд или назад. Для ссылок, относящихся к приложению 1, это отклонение может быть значительнее, так как ссылки выставлены на начало листингов файлов, содержащих адресуемые объекты.

A

Б

В

Г

Д

З

И

K

Л

M

H

O

П

P

С

T

У

Ф

Х

Ц

Ч

Ш

Э

Я

A

B

C

D

Оглавление

Учебное издание

Чернышов Александр Викторович

ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ПРОГРАММИРОВАНИЯ
ИЗ СОСТАВА ОС UNIX
И ИХ ПРИМЕНЕНИЕ В ПОВСЕДНЕВНОЙ ПРАКТИКЕ
УЧЕБНОЕ ПОСОБИЕ

Издание второе, исправленное

Под редакцией автора

Оригинал-макет выполнен в пакете teTEX с использованием кириллических шрифтов семейства LH.

Вёрстка в TEXe: А. В. Чернышов

По тематическому плану внутривузовских изданий учебной литературы на 2010 г.

Подписано в печать Формат 60×90/16. Бумага 80 г/см^2
Гарнитура «Computer Modern». Ризография. Усл. печ. л. 12
Тираж 500 экз. Заказ

Издательство Московского государственного университета леса.
141005. Мытищи-5, Московская обл., 1-я Институтская, 1, МГУЛ.
E-mail: izdat@mgul.ac.ru