Рыбанов А.А., Свиридова О.В.

ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ И МЕТОДОВ ТРАНСЛЯЦИИ

Волжский
2022

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ВОЛЖСКИЙ ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ (ФИЛИАЛ)
ФЕДЕРАЛЬНОГО ГОСУДАРСТВЕННОГО БЮДЖЕТНОГО ОБРАЗОВАТЕЛЬНОГО
УЧРЕЖДЕНИЯ ВЫСШЕГО ОБРАЗОВАНИЯ
«ВОЛГОГРАДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Рыбанов А.А., Свиридова О.В.

ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ И МЕТОДОВ ТРАНСЛЯЦИИ: ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Электронное учебное пособие

2022

УДК 004.43(07)
ББК 32.973я73
Р 931

Рецензенты:

Волжский филиал Волгоградского государственного университета, заведующий кафедрой математики, информатики и естественных наук, доцент, кандидат физико-математических наук
Полковников А.А.,
Волгоградский государственный социально-педагогический университет, доцент кафедры физики, методики преподавания физики и математики, ИКТ, кандидат педагогических наук
Филиппова Е.М.

Издается по решению редакционно-издательского совета
Волгоградского технического университета

Рыбанов, А.А.

Теория формальных языков и методов трансляции [Электронный ресурс] : учебное пособие / А.А. Рыбанов, О.В. Свиридова; ВПИ (филиал) ВолгГТУ, – Электрон. текстовые дан. (1 файл: 1,75 МКБ). – Волжский, 2022. – Режим доступа: http://lib.volpi.ru. – Загл. с титул. экрана.

ISBN 978-5-9948-4329-1

В учебном пособии рассмотрены стадии работы транслятора и схемы взаимодействия блоков транслятора. Описан генератор лексических анализаторов TP Lex. Приведен обзор инструментальных средств автоматизированной разработки трансляторов. Предназначено для студентов высших учебных заведений, обучающихся по направлениям 09.03.01 «Информатика и вычислительная техника» и 09.03.04 «Программная инженерия».

Ил. 18, табл. 1, библиограф.: 6 назв.

ISBN 978-5-9948-4329-1

© Волгоградский государственный технический университет, 2022
© Волжский политехнический институт, 2022

ВВЕДЕНИЕ

Теоретические знания и практические навыки, полученные при изучении методов создания трансляторов, широко используются в различных областях программирования. Преподавание данного курса обычно опирается на широко известные книги [1], [2], ставшие в настоящее время библиографической редкостью. Кроме этого теория и практика разработки трансляторов не стоит на месте. В настоящее время при создании трансляторов широко используются инструментальные средства автоматизированной разработки. На web-сайте http://kit.kulichki.ru предложена классификация инструментальных средств разработки трансляторов, которая разделяет все инструментальные средства на следующие группы: пакеты разработки компиляторов; генераторы лексических и синтаксических анализаторов; системы атрибутной грамматики; средства преобразования грамматик; средства генерации кода; средства анализа и оптимизации; генераторы среды разработки; инфраструктура, компоненты, инструменты.

В учебном пособии описывается генератор лексических анализаторов TP Lex – инструментальное средство автоматизированной разработки лексических анализаторов, базирующихся на теории конечных автоматов.

Рассматривается язык описания лексических анализаторов Lex, базирующийся на аппарате регулярных множеств и выражений.

На основе утилиты Kwtbl рассмотрен один из вариантов организации таблицы ключевых слов транслятора для последующего использования в лексических анализаторах таких, как yylex.

Таким образом, в учебном пособии показано, что использование средств автоматизированной разработки лексических анализаторов требует не только практических знаний в области программирования, но и теоретической подготовки.

1. МОДЕЛИ ОПИСАНИЯ ГРАММАТИК ПРЕДМЕТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ

В любом языке – естественном или искусственном – можно выделить три составляющие: лексику, синтаксис и семантику.

Определение 1. *Лексика* – это правила построения слов языка.

Определение 2. *Синтаксис* (грамматика языка) – это совокупность правил, согласно которым строятся допустимые в данном языке конструкции.

Определение 3. *Семантика* – смысловая сторона языка – соотносит единицы и конструкции языка с некоторым внешним миром, для описания которого язык используется.

Для описания формального языка необходим другой язык, с помощью которого будут создаваться языковые конструкции.

Определение 4. Язык, средствами которого производится описание грамматики, называется *метаязыком*.

Метаязык должен обеспечивать как описание структурных единиц языка и правил объединения их в допустимые предложения, так и содержательную (смысловую) сторону языковых конструкций. Существуют различные способы записи синтаксических правил, что в основном определяется условными обозначениям и ограничениями на структуру правил, принятыми в используемых метаязыках.

Определение 5. Любая грамматика начинается с указания *алфавита*, т.е. набора символов, посредством которого строятся конструкции языка.

Синтаксис формального языка задается некоторой системой правил (порождающей системой), которая из небольшого набора исходных конструкций порождает все допустимые их комбинации, т.е. язык образуется как множество разрешенных правилами сочетаний исходных конструкций. Кроме этого, синтаксис содержит формулировку условия, которое выпол-

няется для законченных конструкций языка и не выполняется в противном случае.

Определение 6. *Формальная грамматика* – система правил, описывающая множество конечных последовательностей символов формального алфавита.

Определение 7. Конечные цепочки символов называются *предложениями формального языка*, а само множество цепочек – *языком*, описываемым данной грамматикой.

Определение 8. *Вывод в данной порождающей грамматике* есть последовательность цепочек, в которой любая, начиная со второй, получается из предыдущей применением какого-либо правила вывода.

Формальная грамматика G задается упорядоченной четверкой (V_T, V_N, S, P), где V_T и V_N – непересекающиеся конечные множества, образующие алфавит или словарь порождаемого формального языка; V_T называется множеством (словарем) терминальных символов; V_N – множеством (словарем) нетерминальных (вспомогательных) символов; S – начальный (выделенный) вспомогательный символ из множества V_N; P – набор правил вывода конструкций языка (подстановок) из выделенного вспомогательного символа, имеющие вид $\alpha \to \beta$, где α и β – цепочки, состоящие как из терминальных, так и нетерминальных символов.

Подстановки работают следующим образом: если в преобразуемой цепочке есть слово α, то оно заменяется словом β. Единственное ограничение на вид подстановок состоит в том, что в слово α не может состоять только из терминальных символов. Это означает, что получение на некотором шаге цепочки, состоящей только из терминальных символов, свидетельствует о прекращении процесса порождения – эта цепочка является правильной, завершенной конструкцией порождаемого языка. Подстанов-

ки P могут применяться к трансформируемой цепочке в произвольном порядке.

Пример 1. Пусть формальная грамматика G задается следующим образом: $V_T = \{a, b\}$ (т.е. множество терминальных символов – алфавит языка – состоит из двух символов – a и b); $V_N = \{S\}$, т.е. множество нетерминальных символов состоит из единственного символа S – он, естественно, оказывается выделенным (начальным); система подстановок P пусть имеет следующий вид: $1. S \to aSa$; $2. S \to bSb$; $3. S \to a$; $4. S \to b$. Таким образом:

$$G = \left(V_T = \{a, b\}, V_N = \{S\}, S, P = \{1.S \to aSa; 2.S \to bSb; 3.S \to a; 4.S \to b\}\right)$$

Описанная грамматика порождает язык, состоящий из всех *«слов-перевертышей»* в алфавите $V_T = \{a, b\}$, имеющих нечетную длину, т.е. слов, которые слева направо читаются также, как справа налево, например, aba, $abababa$, $bbbbb$ и т.д. Легко видеть, что применение первых двух правил (в любом числе и любой последовательности) порождает цепочки (слова) типа $\beta S \beta^l$, где β^l означает слово β, записанное справа налево; применение третьего и четвертого правил завершает процесс порождения слова и формируют слова типа $\beta a \beta^l$ или $\beta b \beta^l$.

Построим последовательность вывода цепочки $aababab aa$ в грамматике G. Последовательность вывода всегда начинается с начального символа грамматики. Над стрелками указывается номер применяемого правила.

$$S \xrightarrow{1} aSa \xrightarrow{1} aaSaa \xrightarrow{2} aabSbaa \xrightarrow{1} aabaSabaa \xrightarrow{4} aababab aa$$

Пример 2. Рассмотрим формальную грамматику, порождающую фрагмент естественного языка. Пусть $V_T = \{а, б, \ldots я, А, Б, \ldots Я\}$ – множество терминальных символов – букв русского алфавита. Нетерминальный алфа-

вит строится из символов $V_N=\{Q,R,S\}$, где $Q=\{q_1,\ldots q_n\}$ – множество имен людей в русском алфавите, $R=\{r_1,\ldots r_m\}$– множество глаголов, стоящих в третьем лице единственного числа настоящего времени. Элементы r_i и q_j записываются с помощью терминальных символов. Пусть система подстановок P имеет вид: $1.S\to QR$; 2. $Q\to q_1$, $Q\to q_2$, $\ldots Q\to q_n$; 3. $R\to r_1$, $R\to r_2$, $\ldots R\to r_m$.

Например, грамматика G имеет вид:

$$G=\left(\begin{array}{c} V_T=\{а,б,\ldots я,А,Б,\ldots Я\},V_N=\{Q,R,S\},S, \\ P=\left\{\begin{array}{lll} 1.S\to QR & 4.Q\to Наталья & 7.R\to учится \\ 2.Q\to Александр & 5.R\to бежит & 8.R\to спит \\ 3.Q\to Мария & 6.R\to работает & 9.R\to читает \end{array}\right\} \end{array}\right)$$

Очевидно, эта грамматика порождает язык, состоящий из фраз типа: *«такой-то делает то-то»*.

Например: $S\xrightarrow{1}QR\xrightarrow{4}НатальяR\xrightarrow{7}Наталья\ учится$

Работает грамматика следующим образом: на первом шаге определяется тип фразы; второй шаг порождает конкретное имя, а третий шаг – конкретное действие (глагол). Из данного примера виден содержательный смысл нетерминальных символов – они могут обозначать различные классы конкретных слов, в частности, традиционные грамматические классы – части речи, члены предложения и пр.

Недостаток последовательностей вывода – громоздкость и многократное переписывание. Для представления вывода можно использовать *синтаксическое дерево вывода*. Синтаксические деревья имеют следующую структуру (рис. 1.):

- корень дерева вывода помечен начальным символом грамматики;

- в каждом внутреннем узле стоит нетерминал;

- листья помечены терминальными символами так, что читаемые слева направо, они представляют терминальную цепочку, выведенную из начального символа.

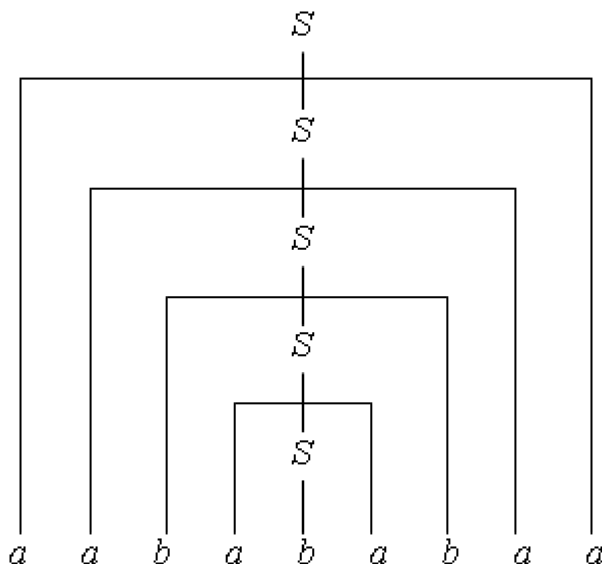

Рис. 1. Синтаксическое дерево вывода $aababab aa$ в грамматике G

Из любого узла дерева, помеченного нетерминальным символом A, идут вниз ветви к узлам, помеченным X_1, $X_2, \ldots,$ X_n, если $A \to X_1 X_2 \ldots X_n$ одно из правил грамматики.

На основе формальных грамматик создаются языки программирования и трансляторы к ним. При решении задач искусственного интеллекта они используются в системах машинного перевода, а также для генерации синтаксически правильных предложений в ответах экспертных систем на запросы пользователей. Формальные грамматики могут применяться в учебных и иных программах (например, *Microsoft Word*), где требуется проверка правильности вводимого текста и поиск в нем ошибок.

1.1. Метаязык Хомского

Метаязык Хомского вышел из недр математической логики. Он имеет следующую систему обозначений:

- символ «$\to$» отделяет левую часть правила от правой (читается как «порождает» и «это есть»);

- нетерминалы A_i – символы, используемые для обозначения конструкций языка, где i – индекс, указывающий номер нетерминала (нетериминал A_1 – начальный символ грамматики);

- терминалы – символы, используемые в описываемом языке;

- каждое правило определяет порождение одной новой цепочки, причем один и тот же нетерминал может встречаться в нескольких правилах слева.

Пример 3. Описание конструкции «*идентификатор*» (A_1) на метаязыке Хомского будет выглядеть следующим образом:

1. $A_1 \to A_2$
2. $A_1 \to A_1 A_2$
3. $A_1 \to A_1 A_3$
4. $A_2 \to a$
5. $A_2 \to b$
6. $A_2 \to c$
7. $A_2 \to d$
8. $A_2 \to e$
9. $A_2 \to f$
10. $A_2 \to g$
11. $A_2 \to h$
12. $A_2 \to i$
13. $A_2 \to j$
14. $A_2 \to k$
15. $A_2 \to l$
16. $A_2 \to m$
17. $A_2 \to n$
18. $A_2 \to o$
19. $A_2 \to p$
20. $A_2 \to q$
21. $A_2 \to r$
22. $A_2 \to s$
23. $A_2 \to t$
24. $A_2 \to u$
25. $A_2 \to v$
26. $A_2 \to w$
27. $A_2 \to x$
28. $A_2 \to y$
29. $A_2 \to z$
30. $A_3 \to 0$

31 $A_3 \to 1$

32. $A_3 \to 2$
33. $A_3 \to 3$
34. $A_3 \to 4$
35. $A_3 \to 5$
36. $A_3 \to 6$
37. $A_3 \to 7$
38. $A_3 \to 8$
39. $A_3 \to 9$

Примеры последовательностей вывода:

1. Последовательность вывода идентификатора $f5u$:

$$A_1 \xrightarrow{2} A_1A_2 \xrightarrow{3} A_1A_3A_2 \xrightarrow{1} A_2A_3A_2 \xrightarrow{9} fA_3A_2 \xrightarrow{35} f5A_2 \xrightarrow{24} f5u$$
.

2. Последовательность вывода идентификатора $id3$:

$$A_1 \xrightarrow{3} A_1A_3 \xrightarrow{2} A_1A_2A_2 \xrightarrow{1} A_2A_2A_3 \xrightarrow{12} iA_2A_3 \xrightarrow{7} idA_3 \xrightarrow{33} id3$$
.

3. Последовательность вывода идентификатора $2fs$:

$A_1 \xrightarrow{3} A_1A_2 \xrightarrow{2} A_1A_2A_2 \xrightarrow{9} A_1fA_2 \xrightarrow{22} A_1fs$, следовательно, идентификатор $2fs$ в грамматике вывести невозможно, т.к. в грамматике нет правила $A_1 \to A_3$.

1.2. Метаязык Хомского-Щутценберже

Приведенный в предыдущем разделе пример описания конструкции *«идентификатор»* показывает громоздкость метаязыка Хомского, что позволяет эффективно использовать его только для описания небольших абстрактных языков. Более компактное описание возможно с применением метаязыка Хомского-Щутценберже, использующего следующие обозначения метасимволов:

- символ «$=$» отделяет левую часть правила от правой (вместо символа «$\rightarrow$»);

- нетерминалы A_i – символы, используемые для обозначения конструкций языка, где i – индекс, указывающий номер нетерминала (нетериминал A_1 – начальный символ грамматики);

- терминалы – символы, используемые в описываемом языке;

- каждое правило определяет порождение нескольких альтернативных цепочек, отделяемых друг от друга символом «$+$».

Введение возможности альтернативного перечисления позволило сократить описание языков.

Пример 4. Описание конструкции *«идентификатор»* на метаязыке Хомского-Щутценберже будет выглядеть следующим образом:

$$A_1 = A_2 + A_1A_2 + A_1A_3$$

$$A_2 = a + b + c + d + e + f + g + h + i + j + k + l + m + n + o + p + q + r + s + t + u + v + w + x + y + z$$

$$A_3 = 0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$$

1.3. Нормальная форма Бэкуса–Наура (БНФ)

Метаязыки Хомского и Хомского-Щутценберже использовались в математической литературе при описании простых абстрактных языков. Метаязык, предложенный Бэкусом и Науром, впервые использовался для описания синтаксиса реального языка программирования Алгол 60. Наряду с новыми обозначениями метасимволов, в нем использовались содержательные обозначения нетерминалов. Это сделало описание языка нагляднее и позволило в дальнейшем широко использовать данную нотацию для описания реальных языков программирования. Для формирования предложений в *нормальной форме Бекуса-Наура* используются универсальные метасимволы: $\{<, >, ::=, |\}$. Метасимволы «$<$» и «$>$» называют *«угловыми скобками»* – они служат для обрамления нетерминального символа. Метасимвол «::=» читается *«по определению есть»*. Метасимвол «|» читается как *«или»*.

В предложениях, записанных в форме Бекуса-Наура, нетерминальный символ, стоящий в угловых скобках, играет роль определяемой конструкции языка. В формулах Бекуса-Наура могут использоваться терминальные символы из алфавита языка, отличные от универсальных метасимволов.

Описание формального языка строится из последовательности формул, каждая из которых в левой части содержит один метасимвол, обозна-

чающий некоторую конструкцию языка. Правая часть такой формулы содержит либо перечисление метасимволов и терминальных символов языка (никаких разделителей при этом не ставится), либо совокупности перечислений, разделенных метасимволом «|». Правая и левая части объединяются в единую формулу метасимволом «::=».

Язык можно считать полностью определенным в нормальной форме Бекуса-Наура, если любой нетерминальный символ можно представить последовательностью терминальных символов.

Пример 5. Рассмотрим конструкцию «*идентификатор*», которая используется во многих языках программирования. На естественном языке определение конструкции звучит следующим образом: *«Идентификатор – это любая последовательность букв и цифр, начинающаяся с буквы»*. В нормальной форме Бекуса-Наура оно будет выглядеть следующим образом:

$$<\textit{идентификатор}> \quad ::= \quad <\textit{буква}>|<\textit{идентификатор}><\textit{буква}>| \\ <\textit{идентификатор}><\textit{цифра}>$$

$$<\textit{буква}> \quad ::= \quad a\,|\,b\,|\,c\,|\,d\,|\,e\,|\mathrm{K}\,|\,z$$

$$<\textit{цифра}> \quad ::= \quad 0|1|2|3|\mathrm{K}\,|9$$

Видно, что в определении данного понятия присутствует рекурсивность, поскольку понятие «идентификатор» определяется через само себя. Элементарным оказывается идентификатор из одной буквы.

Пример 6. Рассмотрим грамматику целых чисел без знака:

$$<\textit{число}> \quad ::= \quad <\textit{цифра}>|<\textit{цифра}><\textit{число}>$$

$$<\textit{цифра}> \quad ::= \quad 0|1|2|3|\mathrm{K}\,|9$$

Пример 7. Рассмотрим язык простейших арифметических формул

$$<\textit{формула}> \quad ::= \quad (<\textit{формула}>)\,|<\textit{число}>|<\textit{формула}><\textit{знак}><\textit{формула}>$$

$$<\textit{число}> \quad ::= \quad <\textit{цифра}>|<\textit{цифра}><\textit{число}>$$

$$<\textit{цифра}> \quad ::= \quad 0|1|2|3|\mathrm{K}\,|9$$

$< знак > \quad ::= \quad + | - | * | /$

Приведем последовательность преобразований цепочки «$3+5*2$» (так называемый *разбор* или *вывод*):

$< формула > \quad ::= \quad < формула >< знак >< формула > \quad ::=$

$< формула >< знак >< формула >< знак >< формула > \quad ::=$

$< число >< знак >< формула >< знак >< формула > \quad ::=$

$< цифра >< знак >< формула >< знак >< формула > \quad ::=$

$3 < знак >< формула >< знак >< формула > \quad ::=$

$3+ < формула >< знак >< формула > \quad ::=$

$3+ < число >< знак >< формула > \quad ::=$

$3+ < цифра >< знак >< формула > \quad ::=$

$3+5 < знак >< формула > \quad ::=$

$3+5* < формула > \quad ::=$

$3+5* < число > \quad ::=$

$3+5* < цифра > \quad ::=$

$3+5*2$

Синтаксическое дерево вывода цепочки «$3+5*2$» представлено на рисунке 2.

Большинство грамматик допускают *несколько различных выводов* для одной и той же цепочки из языка.

Если в процессе вывода цепочки правила грамматики применяются только к самому левому нетерминалу, говорят, что получен левый вывод цепочки. Аналогично определяется правый вывод. В примере показан левый вывод.

Достоинство БНФ в том, что грамматика языка представляется в буквенном виде; неудобна БНФ однообразностью способов построения

предложений языка – запись грамматики оказывается громоздкой и плохо воспринимаемой.

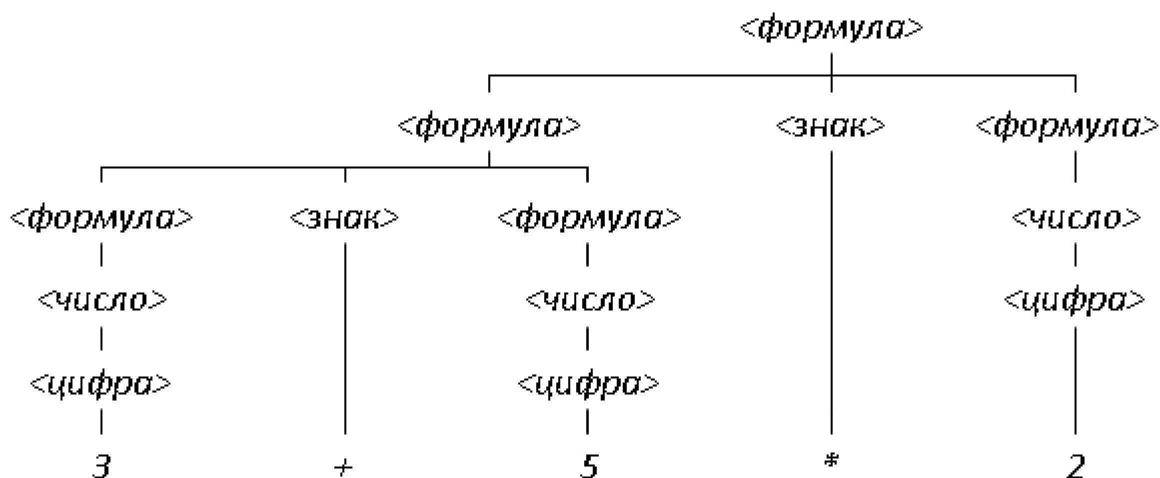

Рис. 2. Синтаксическое дерево вывода цепочки $3+5*2$

1.4. Расширенные Бэкуса-Наура формы (РБНФ)

Метаязыки, представленные выше, позволяют описывать любой синтаксис. Однако для повышения удобства и компактности описания, целесообразно вести в язык дополнительные конструкции. В частности, специальные метасимволы были разработаны для описания необязательных цепочек, повторяющихся цепочек, обязательных альтернативных цепочек. Существуют различные расширенные формы метаязыков, незначительно отличающиеся друг от друга. Их разнообразие зачастую объясняется желанием разработчиков языков программирования по-своему описать создаваемый язык. К примерам таких широко известных метаязыков можно отнести: метаязык PL/I, метаязык Вирта, используемый при описании Модулы-2, метаязык Кернигана-Ритчи, описывающий Си. Зачастую такие языки называются расширенными формами Бэкуса-Наура (РБНФ).

В частности, РБНФ, используемые Виртом, имеют следующие особенности:

- квадратные скобки «[» и «]» означают, что заключенная в них синтаксическая конструкция может отсутствовать;

- фигурные скобки «{» и «}» означают повторение синтаксической конструкции (возможно, 0 раз);

- круглые скобки «(» и «)» используются для ограничения альтернативных конструкций;

- сочетание фигурных скобок и косой черты «{/» и «/}» используется для обозначения повторения один и более раз.

Нетерминальные символы изображаются словами, выражающими их интуитивный смысл и написанными на русском языке.

Если нетерминал состоит из нескольких смысловых слов, то они должны быть написаны слитно. В этом случае для повышения удобства в восприятии фразы целесообразно каждое ее слово начинать с заглавной буквы или разделять слова во фразах символом подчеркивания. Терминальные символы изображаются словами, написанными буквами латинского алфавита (зарезервированные слова) или цепочками знаков, заключенными в кавычки. Синтаксическим правилам предшествует знак «$» в начале строки. Каждое правило оканчивается знаком «.» (точка). Левая часть правила отделяется от правой знаком «::=» (равно), а альтернативы - вертикальной чертой «|».

Пример 8. В соответствии с данными правилами синтаксис идентификатора будет выглядеть следующим образом:

$\$идентификатор \quad ::= \quad буква\{буква \mid цифра\}.$

$\$буква \quad ::= \quad "a"|"b"|"c"|"d"|"e"|\text{K} \;|"z".$

$\$цифра \quad ::= \quad "0"|"1"|"2"|"3"|\text{K} \;|"9".$

1.5. Диаграммы Вирта

Гораздо более наглядной следует считать другой способ описания формального языка, который был предложен Никласом Виртом, создателем языка программирования *PASCAL,* и который получил название *«син-*

таксические диаграммы». Синтаксическая диаграмма – это схема (графическое представление) описания какого-либо нетерминального символа языка-объекта. Схема всегда имеет один вход и один выход. Элементами схемы могут служить терминальные символы языка, заключенные в окружность (или овал) или нетерминальные символы (конструкции) языка, заключенные в прямоугольник. Элементы соединяются между собой направленными линиями, указывающие порядок следования объектов в определяемом нетерминальном символе. Приняты следующие обозначения (рис. 3).

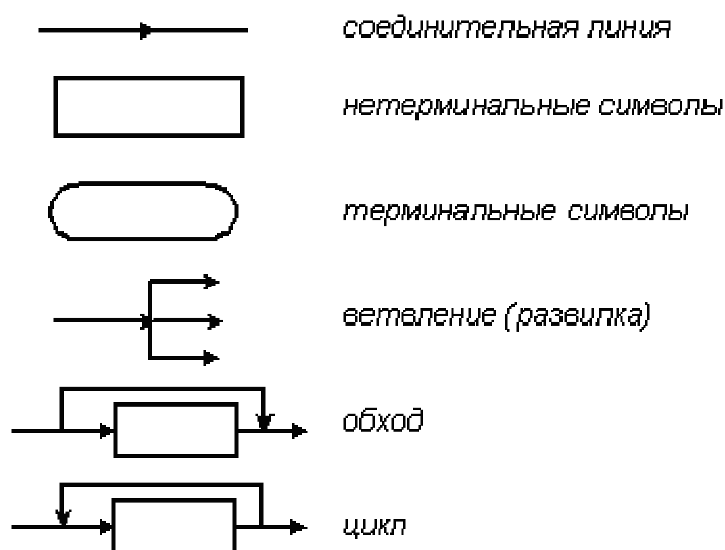

Рис. 3. Обозначения диаграмм Вирта

Структура синтаксических диаграмм идентична структурам языков программирования, что позволило широко использовать диаграммы для написания трансляторов различных языков. Первым языком, описанным с помощью синтаксических диаграмм, был язык *PASCAL.*

Чтение диаграммы производится в направлении стрелок; в точке ветвления может выбираться любой маршрут. В качестве метаязыка может использоваться естественный русский язык; языки программирования строятся на англоязычной основе. Терминальные символы переписываются в конструкции формального языка дословно. Нетерминальные символы могут выражаться через терминальные или другие нетерминальные – в

этом случае для них строятся уточняющие диаграммы; в конечном счете, все нетерминальные символы должны быть выражены через терминальные. При использовании синтаксических диаграмм принимается условие, что среди терминальных символов языка-объекта не должно быть одинаковых, а также ни один из терминальных символов не может служить началом другого. При нарушении данного условия возможно неоднозначное чтение диаграммы и, как следствие, построение или распознавание неверной конструкции языка.

Рассмотрим ряд примеров построения синтаксических диаграмм, первым из которых будет определение понятия *«идентификатор»* (рис. 4) для сопоставления с приведенной выше нормальной формой Бекуса-Наура (см. пример 5).

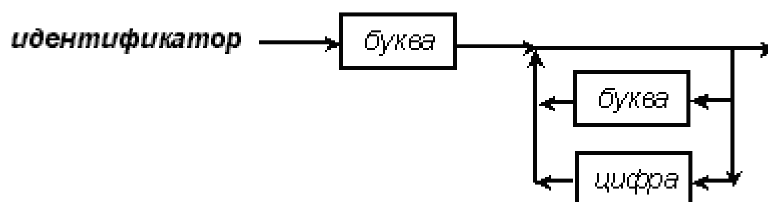

Рис. 4. Синтаксическая диаграмма конструкции *идентификатор*

Необходимы уточняющие диаграммы (рис. 5):

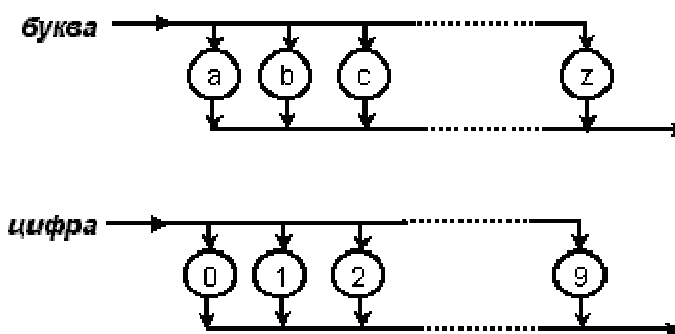

Рис. 5. Синтаксические диаграммы конструкций *буква* и *идентификатор*

Примерами построения англоязычных идентификаторов в соответствии с этой диаграммой являются: *q, a123, identificator, e2e4*.

Диаграмма, задающая общий вид программы на языке *PASCAL*, выглядит следующим образом (рис. 6):

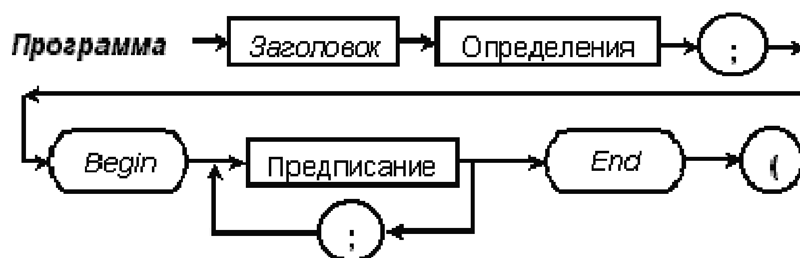

Рис. 6. Синтаксическая диаграмма конструкции *программа* на языке Pascal

В качестве примеров предписаний рассмотрим *условное* и *циклическое* (рис. 7).

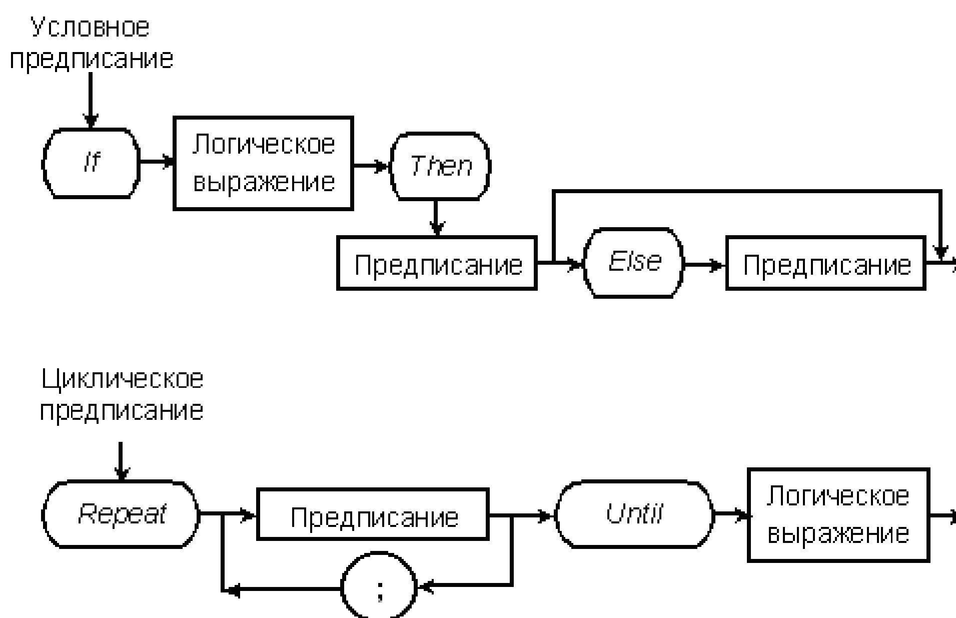

Рис. 7. Синтаксические диаграммы конструкций *условное предписание* и *циклическое предписание*

В соответствии с этими и подобными диаграммами строятся допустимые синтаксические конструкции языка.

Итак, нотации Бекуса-Наура и синтаксические диаграммы – это два альтернативных способа описания конструкций метаязыка, с помощью которого строится формальный язык. После того, как построена формальная грамматика, и ею порожден язык, он может быть использован для решения прикладных задач – коммуникации, хранения и обработки информации. Последний класс задач приводит к необходимости формулировки с помощью языков последовательностей обработки информации, т.е. алгоритмов,

и их представлению в форме, доступной для понимания и исполнения лицом или техническим устройством, которые обработку производят.

Диаграммы Вирта позволяют задавать альтернативы, рекурсии, итерации и по изобразительной мощности эквивалентны РБНФ. Но графическое отображение правил более наглядно. Кроме этого допускается произвольное проведение дуг, что уменьшает количество элементов в правиле за счет его неструктурированности. Диаграммы Вирта являются удобным исходным документом для построения лексического и синтаксического анализаторов.

2. СТАДИИ РАБОТЫ ТРАНСЛЯТОРА

Работа компилятора состоит из нескольких стадий, которые могут выполняться последовательно либо совмещаться по времени. Эти стадии могут быть представлены в виде схемы (рис. 8).

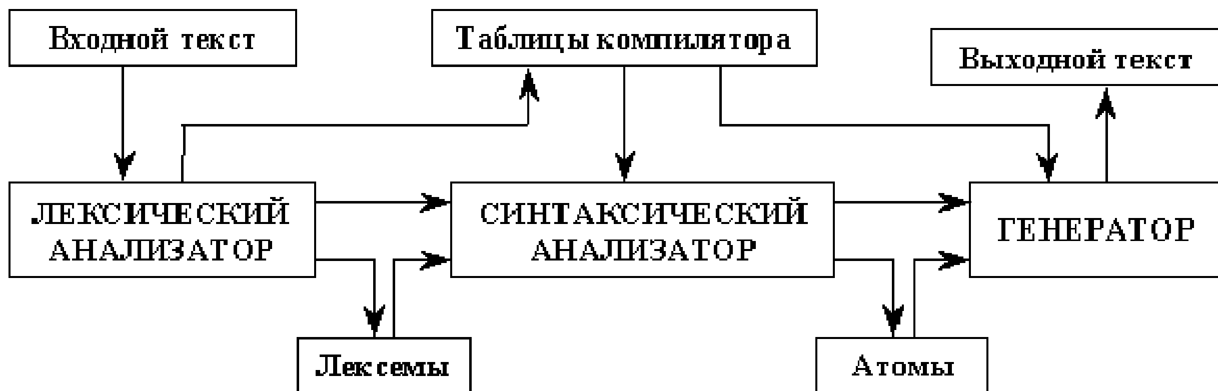

Рис. 8. Стадии работы компилятора

Первая стадия работы компилятора называется лексическим анализом, а программа, её реализующая, – лексическим анализатором (ЛА). На вход лексического анализатора подаётся последовательность символов входного языка. ЛА выделяет в этой последовательности простейшие конструкции языка, которые называют лексическими единицами. Примерами лексических единиц являются идентификаторы, числа, символы операций, служебные слова и т.д. ЛА преобразует исходный текст, заменяя лексические единицы их внутренним представлением – лексемами. Лексема может включать информацию о классе лексической единицы и её значении. Кроме того, для некоторых классов лексических единиц ЛА строит таблицы, например, таблицу идентификаторов, констант, которые используются на последующих стадиях компиляции.

Вторую стадию работы компилятора называют синтаксическим анализом, а соответствующую программу – синтаксическим анализатором (СА). На вход СА подается последовательность лексем, которая преобразуется в промежуточный код, представляющий собой последовательность символов действия или атомов. Каждый атом включает описание опера-

ции, которую нужно выполнить, с указанием используемых операндов. При этом последовательность расположения атомов, в отличие от лексем, соответствует порядку выполнения операций, необходимому для получения результата.

На третьей стадии работы компилятора осуществляется построение выходного текста. Программа, реализующая эту стадию, называется генератором выходного текста (Г). Генератор каждому символу действия, поступающему на его вход, ставит в соответствие одну или несколько команд выходного языка. В качестве выходного языка могут быть использованы команды устройства, команды ассемблера либо операторы какого-либо другого языка.

Рассмотренная схема компилятора является упрощенной, поскольку реальные компиляторы, как правило, включают стадии оптимизации.

2.1. Лексический анализ

Лексический анализ применяется во многих случаях, например, для построения пакетного редактора или в качестве распознавателя директив в диалоговой программе и т.д. Однако наиболее важное применение ЛА – это использование его в компиляторе. Здесь лексический анализатор выполняет функцию программы ввода данных.

ЛА выполняет первую стадию компиляции – читает строки компилируемой программы, выделяет лексемы и передает их на дальнейшие стадии компиляции (синтаксический анализ и кодогенерацию).

ЛА должен не только выделить лексему, но и выполнить некоторые преобразования. Например, если лексема – число, то его необходимо перевести во внутреннюю (двоичную) форму записи как число с плавающей или фиксированной точкой. А если лексема – идентификатор, то его необходимо разместить в таблице идентификаторов, чтобы в дальнейшем обращаться к нему не по имени, а по адресу в таблице.

Хотя лексический анализ по своей идее прост, тем не менее, эта фаза работы компилятора часто занимает больше времени, чем любая другая. Частично это происходит из-за необходимости просматривать и анализировать исходный текст символ за символом. Иногда даже бывает необходимо вернуть прочитанный символ во входной поток с тем, чтобы повторить просмотр и анализ. Происходит это потому, что часто бывает трудно определить, где проходят границы лексемы. Допустим, имеются две лексемы: **make** и **makefile**. Пусть поток входного текста содержит цепочку символов:

source makefile file compiler

При анализе потока символов входного текста будет выделена лексема **make**, хотя правильно было бы выделить лексему **makefile**.

Единственный способ преодолеть это затруднение – просмотр полученной цепочки символов назад и вперед. В нашем примере при выделении лексемы **make** мы должны просмотреть следующий поступающий символ и, если он будет символом «**f**», то вполне возможно, что поступает лексема **makefile**.

s o u r c e   m a k e f i l e   f i l e   c o m p i l e r

⇐ ⇒

Процесс просмотра потока входного текста можно рассматривать как движение влево и вправо рамки над цепочкой символов. При этом анализируется только тот символ, который охвачен рамкой.

На стадии лексического анализа также обнаруживаются некоторые ошибки (недопустимые символы, неправильная запись чисел, идентификаторов и другие). Работа лексического анализатора описывается формализмом конечных автоматов. Однако описание лексического анализатора в виде конечного автомата неудобно практически. Поэтому для описания лексических анализаторов, как правило, используют либо формализм регулярных выражений, либо формализм контекстно-свободных грамматик.

Все три формализма имеют одинаковую выразительную мощность. По описанию лексического анализатора в виде регулярного выражения или автоматной грамматики строится конечный автомат, распознающий соответствующий язык.

TP Lex генерирует лексический анализатор, работа которого заключается в определении соответствия рассматриваемой последовательности символов некоторому так называемому регулярному выражению. Например, записанное на lex-языке регулярное выражение:

(+|-)?[0-9]+

позволяет выделить в цепочке все лексемы типа «целое число», перед которыми либо указан знак («+» или «–»), либо не указан. Для вещественных чисел это выражение имело бы вид:

(+|-)?[0-9]+[.][0-9]+

В тех случаях, когда выделение лексемы затруднено либо по причине того, что одно регулярное выражение не позволяет ее однозначно определить, либо из-за того, что лексема является частью другой, приходится прибегать к контекстно-зависимым алгоритмам анализа с использованием левого и правого направлений просмотра входной цепочки символов.

2.2. Варианты взаимодействия блоков транслятора

Организация процессов трансляции, определяющая реализацию основных фаз, может осуществляться различным образом. Это определяется различными вариантами взаимодействия блоков транслятора: лексического анализатора, синтаксического анализатора и генератора кода. Несмотря на одинаковый конечный результат, различные варианты взаимодействия блоков транслятора обеспечивают различные варианты хранения промежуточных данных. Можно выделить два основных варианта взаимодействия блоков транслятора:

- многопроходную организацию, при которой каждая из фаз является независимым процессом, передающим управление следующей фазе только после окончания полной обработки своих данных;

- однопроходную организацию, при которой все фазы представляют единый процесс и передают друг другу данные небольшими фрагментами.

На основе двух основных вариантов можно также создавать их разнообразные сочетания. Многопроходная организация взаимодействия блоков транслятора представлена на рисунке 9.

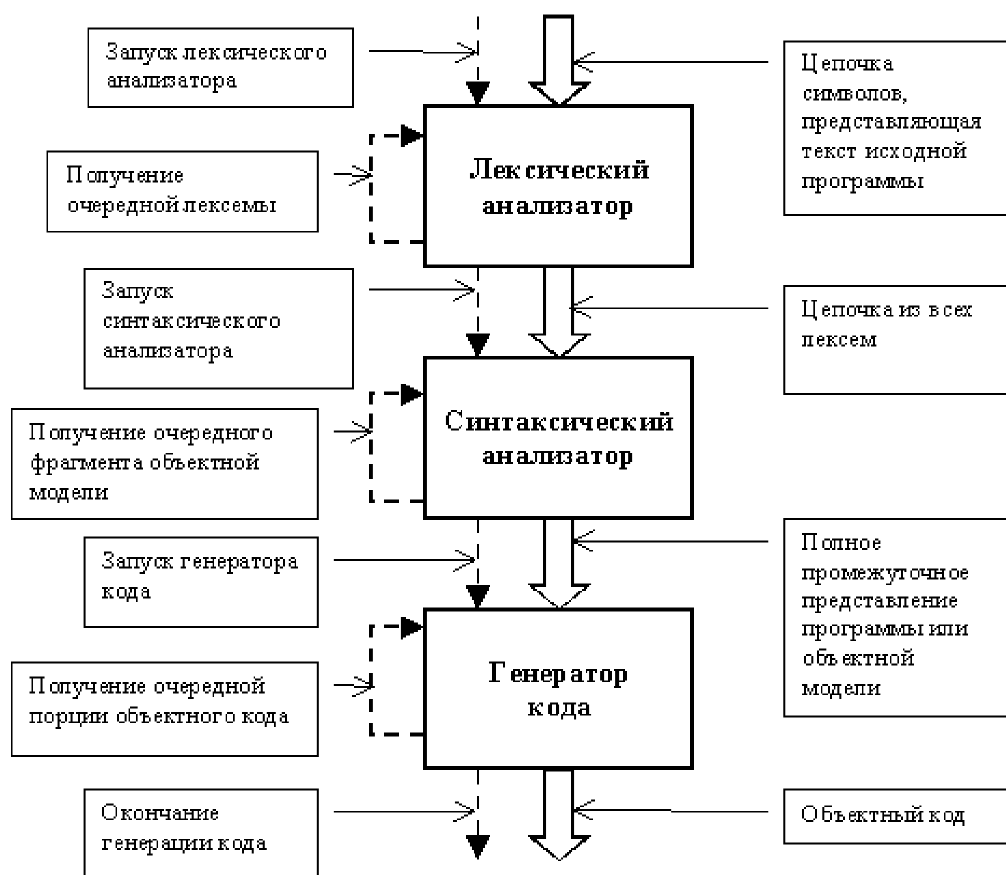

Рис. 9. Многопрохожное взаимодействие блоков компилятора

Лексический анализатор полностью обрабатывает исходный текст, формируя на выходе цепочку, состоящую из всех полученных лексем. Только после этого управление передается синтаксическому анализатору. Синтаксический анализатор получает сформированную цепочку лексем и на ее основе формирует промежуточное представление или объектную мо-

дель. После получения всей объектной модели он передает управление генератору кода. Генератор кода, на основе объектной модели языка, строит требуемый машинный код.

К достоинствам такого подхода можно отнести:

1) Обособленность отдельных фаз, что позволяет обеспечить их независимую друг от друга реализацию и использование.

2) Возможность хранения данных, получаемых в результате работы каждой из фаз, на внешних запоминающих устройствах и их использования по мере надобности.

3) Возможность уменьшения объема оперативной памяти, требуемой для работы транслятора, за счет последовательного вызова фаз.

К недостаткам следует отнести:

1) Наличие больших объемов промежуточной информации, из которой в данный момент времени требуется только небольшая часть.

2) Замедление скорости трансляции из-за последовательного выполнения фаз и использования для экономии оперативной памяти внешних запоминающих устройств.

Данный подход может оказаться удобным при построении трансляторов с языков программирования, обладающих сложной синтаксической и семантической структурой (например, PL/I). В таких ситуациях трансляцию сложно осуществить за один проход, поэтому результаты предыдущих проходов проще представлять в виде дополнительных промежуточных данных.

Следует отметить, что сложные семантическая и синтаксическая структуры языка могут привести к дополнительным проходам, необходимым для установления требуемых зависимостей. Общее количество проходов может оказаться более десяти. На число проходов может также влиять использование в программе конкретных возможностей языка, таких

как объявление переменных и процедур после их использования, применение правил объявления по умолчанию и т.д.

Однопроходная организация взаимодействия блоков транслятора при управлении, инициируемым лексическим анализатором представлена на рисунке 10.

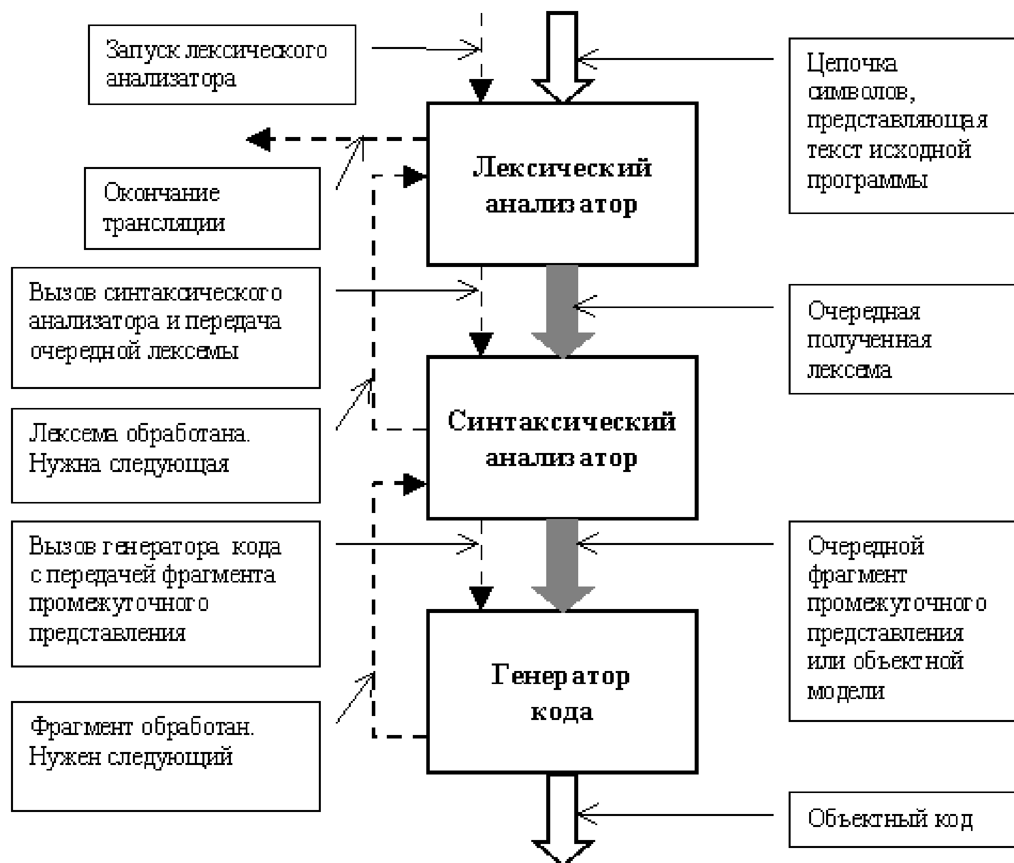

Рис. 10. Однопроходная организация взаимодействия блоков транслятора при управлении, инициируемым лексическим анализатором

В этом случае процесс трансляции протекает следующим образом. Лексический анализатор читает фрагмент исходного текста, необходимый для получения одной лексемы. После формирования лексемы им осуществляется вызов синтаксического анализатора и передача ему созданной лексемы в качестве параметра. Если синтаксический анализатор может построить очередной элемент промежуточного представления, то он делает это и передает построенный фрагмент генератору кода. В противном слу-

чае синтаксический анализатор возвращает управление сканеру, давая тем самым понять, что очередная лексема обработана и нужны новые данные.

Генератор кода функционирует аналогичным образом. По полученному фрагменту промежуточного представления он создает соответствующий фрагмент объектного кода. После этого управление возвращается синтаксическому анализатору.

По окончании исходного текста и завершении обработки всех промежуточных данных каждым из блоков лексический анализатор инициирует процесс завершения программы.

Чаще всего в однопроходных трансляторах используется другая схема управления, в которой роль основного блока играет синтаксический анализатор (рис. 11).

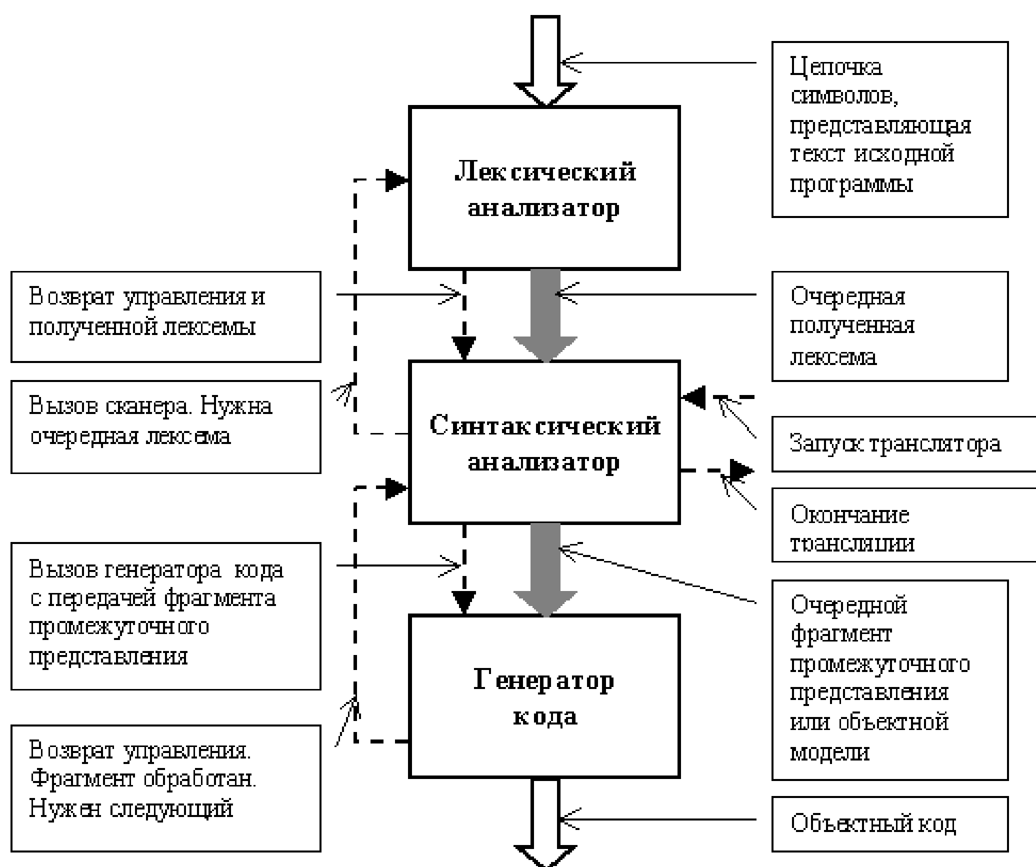

Рис. 11. Однопроходное взаимодействие блоков компилятора при управлении, инициируемом синтаксическим анализатором

Лексический анализатор и генератор кода выступают в роли вызываемых им подпрограмм. Как только синтаксическому анализатору нужна очередная лексема, он вызывает сканер. При получении фрагмента промежуточного представления осуществляется обращение к генератору кода. Завершение процесса трансляции происходит после получения и обработки последней лексемы и инициируется синтаксическим анализатором.

К достоинствам однопроходной схемы следует отнести отсутствие больших объемов промежуточных данных, высокую скорость обработки из-за совмещения фаз в едином процессе и отсутствие обращений к внешним запоминающим устройствам.

К недостаткам относятся: невозможность реализации такой схемы трансляции для сложных по структуре языков и отсутствие промежуточных данных, которые можно использовать для комплексного анализа и оптимизации.

Рис. 12. Однопроходное взаимодействие блоков интерпретатора

Такая схема часто применяется для простых по семантической и синтаксической структурам языков программирования как в компиляторах, так и в интерпретаторах. Примерами таких языков могут служить Basic и Pascal.

Классический интерпретатор обычно строится по однопроходной схеме, так как непосредственное исполнение осуществляется на уровне отдельных фрагментов промежуточного представления. Организация взаимодействия блоков такого интерпретатора представлена на рисунке 12.

Сочетания многопроходной и однопроходной схем трансляции порождают разнообразные комбинированные варианты, многие из которых успешно используются. В качестве примера можно рассмотреть некоторые из них.

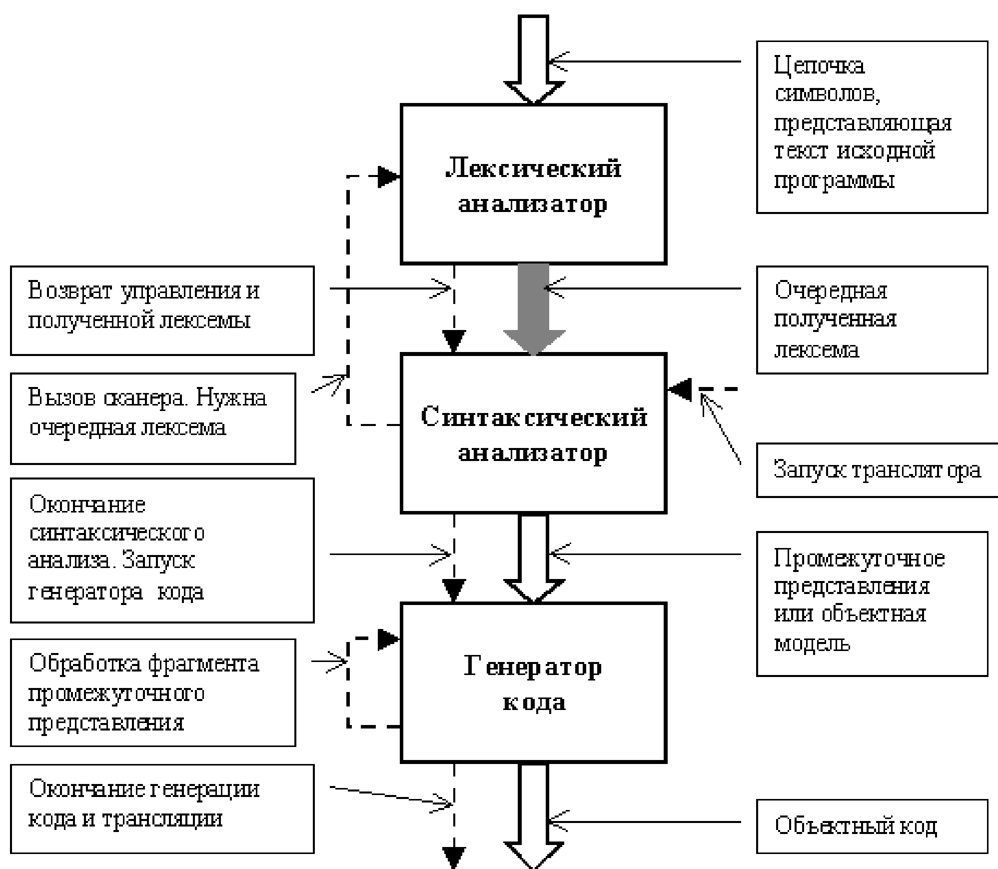

Рис. 13. Двухпроходная схема с генерацией полного промежуточного представления

На рисунке 13 представлена схема взаимодействия блоков транслятора, разбивающая весь процесс на два прохода. На первом проходе по-

рождается полное промежуточное представление программы, а на втором осуществляется генерация кода. Использование такой схемы позволяет легко переносить транслятор с одной вычислительной системы на другую путем переписывания генератора кода. Кроме этого, вместо генератора кода легко подключить эмулятор промежуточного представления, что достаточно просто позволяет разработать систему программирования на некотором языке, ориентированную на различные среды исполнения. Пример подобной организации взаимодействия блоков транслятора представлен на рисунке 14.

Рис. 14. Эмулятор промежуточного представления

Наряду со схемами, предполагающими замену генератора кода на эмулятор, существуют трансляторы, допускающие их совместное использование. Одна из таких схем представлена на рисунке 15.

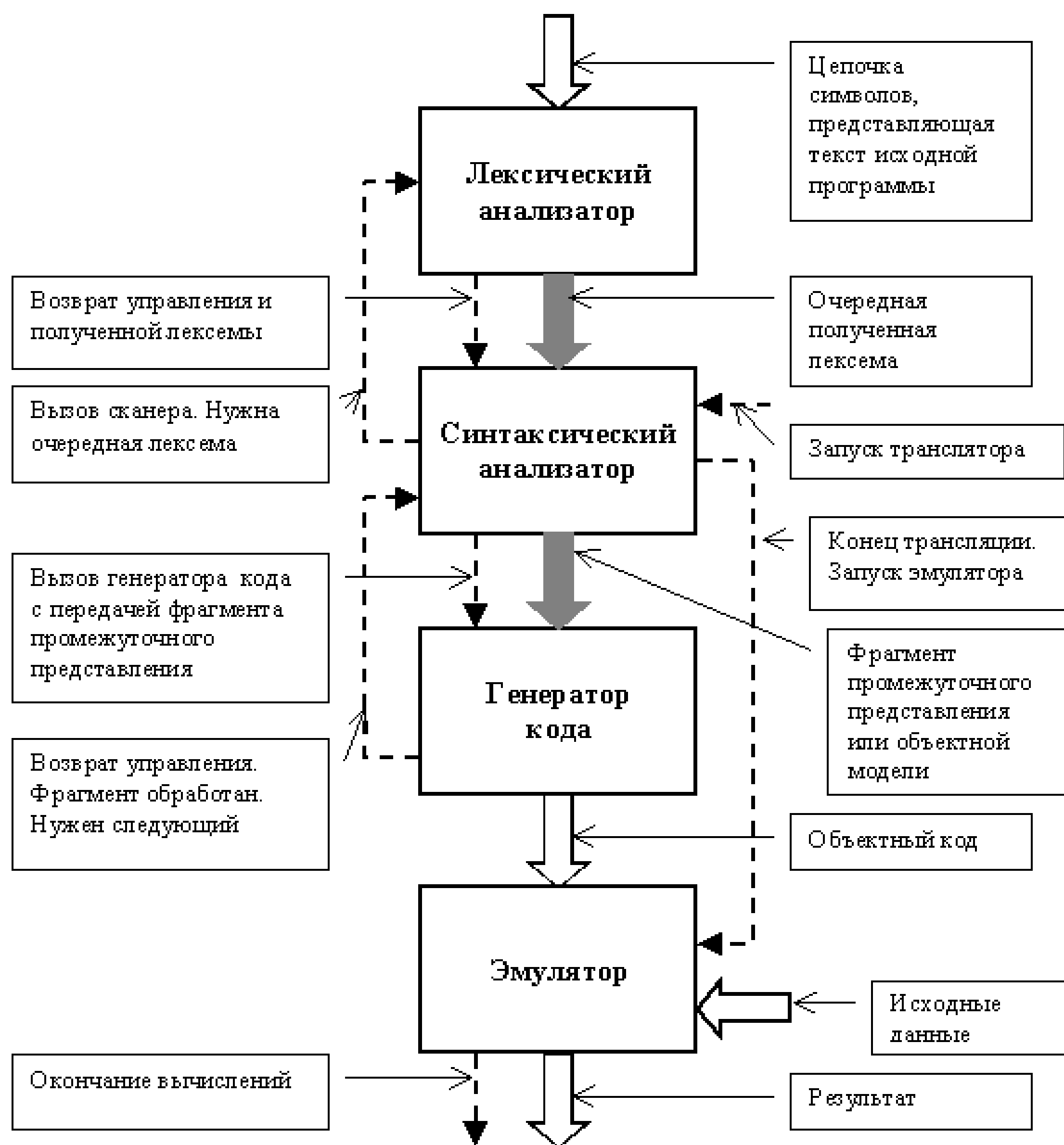

Рис.15. Эмуляция скомпилированного объектного кода

Процесс трансляции, включая и генерацию кода, может быть выполнен за любое число проходов. Однако сформированный объектный код не исполняется на соответствующей ему вычислительной системе, а эмулируется на компьютере с другой архитектурой. Такая схема применяется в среде построенной вокруг языка программировании Java. Сам транслятор генерирует код виртуальной Java-машины, эмуляция которого осуществляется специальными средствами как автономно, так и в среде Internet браузера.

Представленная схема может иметь и более широкое толкование применительно к любому компилятору, порождающему машинный код. Все дело в том, что большинство современных вычислительных машин реализованы с использованием микропрограммного управления. А микропрограммное устройство управления можно рассматривать как программу, эмулирующую машинный код. Это позволяет говорить о повсеместном использовании последней представленной схемы.

3. ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ TP LEX

3.1. Назначение TP LEX и Yacc

Комплект инструментальных средств TP Lex и Yacc предназначен для автоматизации процесса создания трансляторов на языке программирования Turbo Pascal.

TP Lex и Yacc – это адаптация для Turbo Pascal, хорошо известных UNIX утилит Lex и Yacc, используемых совместно с языком программирования Си. TP Lex и Yacc задумывались как «совместимые» с этими программами. Однако они получили независимое развитие и разработаны на методах, описанных в известной книге Aho, Sethi, Ullman «Compilers: principles, techniques and tools».

Версия TP Lex и Yacc 4.1 работает с программными продуктами в основу которых заложен Turbo/Borland Pascal: Delphi, компилятор Pascal, а также Turbo Pascal совместимыми компиляторами, которые работают под Dos и Linux.

TP Lex и Yacc, подобно другим инструментальным средствам этого вида, не предназначены для новичков и начинающих программистов. Они требуют не только большого опыта программирования, но также и полного понимания принципов проектирования и работы лексических и синтаксических анализаторов. Но если вы имеете опыт программирования на языке Turbo Pascal и обладаете некоторыми знаниями о процессе проекти-

рования трансляторов и теории формальных языков, то вы найдете для себя TP Lex и Yacc мощным инструментальным средством, расширяющим возможности Turbo Pascal.

В методических указаниях рассматривается инструментальное средство TP Lex. TP Lex частично или полностью автоматизирует процесс создания лексического анализатора. TP Lex – это программирующая программа или генератор лексических анализаторов.

TP Lex строит программу – лексический анализатор на так называемом host-языке (или «главном» языке). Это значит, что lex-программа пишется на lex-языке (специальном языке описания лексических анализаторов), а TP Lex, в свою очередь, генерирует программу лексического анализа на host-языке.

TP Lex генерирует программный код лексического анализатора на host-языке (в данной версии TP Lex в качестве host-языка будем использовать язык Turbo Pascal), используя его описание на входном lex-языке. Эта программа выполняет просмотр потока входного текста, распознавание и классификацию различных лексем (см. рис. 16).

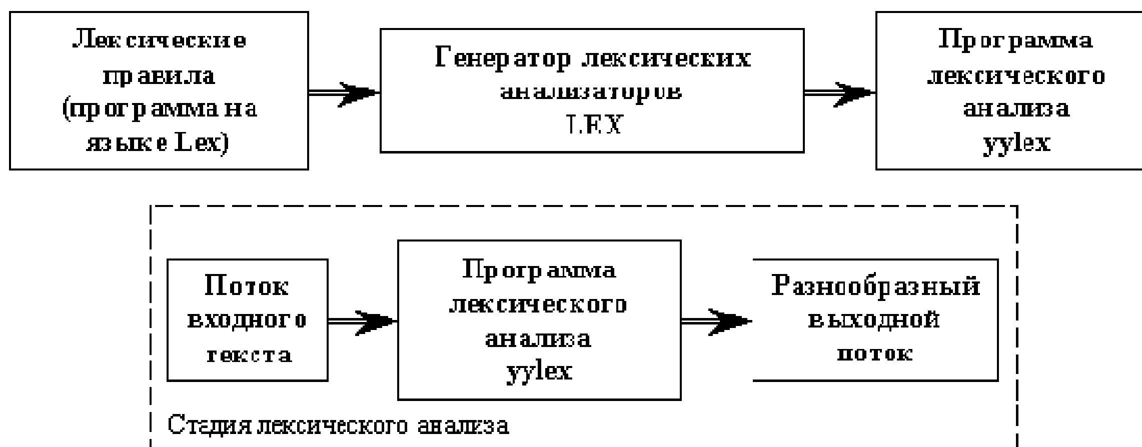

Рис. 16. Генерация лексического анализатора

Лексический анализатор описывается множеством регулярных выражений, каждое из которых задаёт одну или множество лексем.

Сгенерированный TP Lex'ом выходной файл содержит стандартную функцию лексического анализатора yylex, описываемую как:

function yylex : Integer;

Эта функция должна вызываться из создаваемой пользователем главной программы для выполнения лексического анализа. Возвращаемое функцией yylex значение обычно обозначает код лексемы, распознанной лексическим анализатором (см. процедуры return и returnc в модуле библиотеки LexLib). При достижении конца текста входного потока функция yylex возвращает 0.

TP Lex можно использовать как самостоятельное инструментальное средство для генерации программ простых преобразований или же программ анализа и сбора статистики на лексическом уровне. TP Lex также может использоваться совместно с генератором синтаксических анализаторов TP Yacc для выполнения стадии лексического анализа.

Особенно простым является взаимодействие между TP Lex и Yacc'ом. Программы, сгенерированные TP Lex, распознают только регулярные выражения, а TP Yacc создает синтаксические анализаторы, распознающие большой класс контекстно-свободных грамматик, но требует лексического анализатора для распознавания лексем в потоке входного текста. Таким образом, удобно использование TP Lex и Yacc в комплексе.

TP Lex является препроцессором для синтаксического анализатора. TP Lex разбивает поток входного текста на лексемы, а синтаксический анализатор TP Yacc структурирует получившиеся лексические единицы. Поток управления в данном случае (например, это может быть первой частью компилятора) может быть представлен в виде схемы (см. рис. 17).

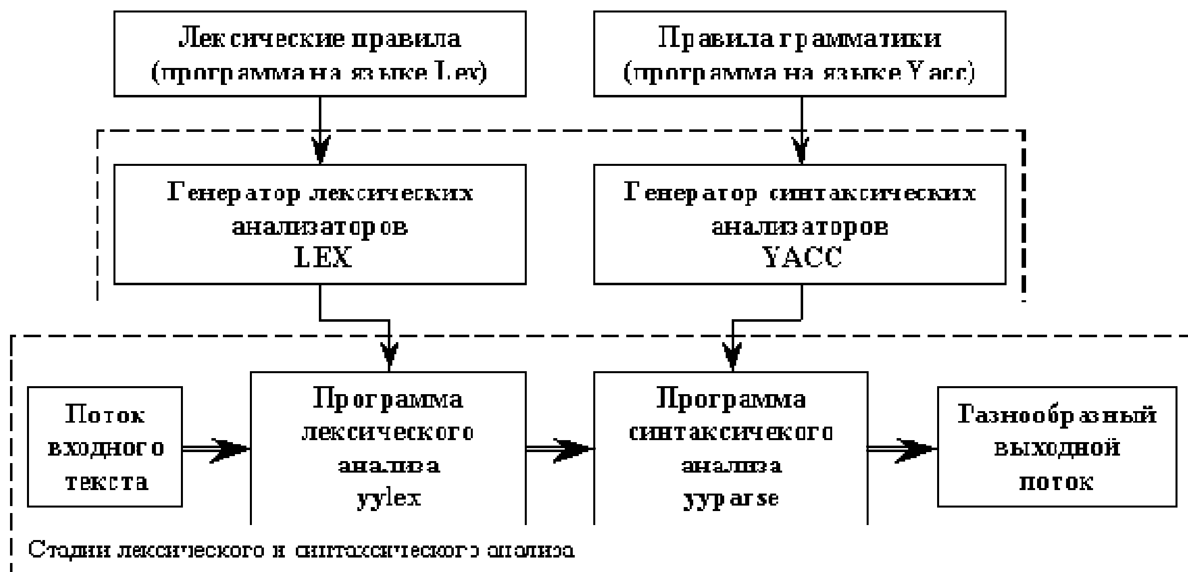

Рис. 17. Взаимодействие TP Lex и Yacc

К программам, сгенерированным Lex'ом, могут легко быть добавлены программы, написанные вручную или с помощью других генераторов.

Шаблон программного кода для функции yylex может быть найден в файле yylex.cod. Этот файл используется TP Lex'ом для создания выходного файла лексического анализатора с программным кодом на host-языке. Он является уже готовой программой лексического анализа, но в нём не определены действия, которые необходимо выполнять при распознавании лексем, отсутствуют и сами лексемы, не сформированы рабочие массивы и т.д.

Файл yylex.cod должен присутствовать или в текущем каталоге, или в специальном каталоге, в зависимости от версии TP Lex (TP Lex производит поиск этого файла в указанных каталогах в перечисленном выше порядке). Например, для версии Linux/Free Pascal поиск шаблона программного кода происходит в специальной дирректории /usr/lib/fpc/lexyacc. TP Lex на основе lex-программы достраивает файл yylex.cod . В результате мы получаем на host-языке программный код, выполняющий лексический анализ.

TP Lex библиотека (модуль LexLib) необходима программам, использующим созданные генератором TP Lex'ом лексические анализаторы. Поэтому нужно включить в создаваемую программу лексического анализатора модуль библиотеки LexLib, содержащий стандартные программы, входящие в состав сгенерированного TP Lex'ом лексического анализатора. Модуль библиотеки LexLib также содержит полезные процедуры и функции, которые можно использовать при построении лексического анализатора. Этот модуль будет описан далее.

3.2. Использование TP Lex

Командная строка вызова генератора лексических анализаторов TP Lex имеет следующий формат:

lex [options] lex-file[.l] [output-file[.pas]],

где lex-file.l – входной файл описания лексического анализатора на lex-языке; output-file.pas – выходной файл программного кода лексического анализатора на языке Pascal; options – описанные ниже опции.

Квадратные скобки означают, что элементы, заключенные в них, могут как присутствовать, так и отсутствовать.

В командной строке вызова TP Lex могут быть указаны следующие опции:

– v «Пояснение:» – TP Lex создаёт файл пояснения с расширением «.lst» для генерируемого лексического анализатора;

– o «Оптимизация:» – TP Lex оптимизирует функцию переходов конечного автомата для построения минимального конечного автомата.

3.3. Различия TP Lex и UNIX Lex

Основные различия между TP Lex и UNIX Lex состоят в следующем:

– TP Lex генерирует выходной файл лексического анализатора на языке Turbo Pascal, а UNIX Lex – на языке C.

– В TP Lex не поддерживается определение таблицы наборов символов (в UNIX Lex это дирректива %T); нет ни одной диррективы для переопределения размеров внутренних таблиц (в UNIX Lex это диррективы %p, %n и т.д.).

– Некоторые компоненты стандартной библиотеки TP Lex (переменные, процедуры и функции) имеют другое название и описание, в отличие от библиотеки UNIX версии (например, стандартная процедура Start() заменяет макроподстановку BEGIN версии UNIX Lex); все макроподстановки библиотеки UNIX Lex (ECHO, REJECT, и др.) описаны как процедуры в библиотеке TP Lex.

– В TP Lex библиотеке начальное значение переменной yylineno, содержащей номер текущей считанной строки, равно 0, и увеличивается прежде, чем строка будет считана из входного потока (в отличие от этого, в UNIX Lex библиотеке начальное значение yylineno равно 1 и увеличивается после того, как строка будет считана). Эта особенность не влияет на создаваемые программы, если в них явно не изменяется значение yylineno (например, при открытии нового входного файла). В том случае, если в программе явно открывается новый входной файл, переменной yylineno должно быть присвоено значение 0, а не 1.

4. ЯЗЫК ОПИСАНИЯ ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ LEX

4.1. Структура lex-программы

Как уже было сказано ранее, TP Lex генерирует лексический анализатор, используя его описание на входном lex-языке. Рассмотрим структуру lex-программы, содержащей это описание.

Lex-программа включает разделы определений, правил и пользовательских программ и имеет следующий формат (см. рис. 18).

Раздел определений и ***раздел программ пользователя*** могут быть пустыми. Первая пара **%%** является обязательной и отмечает начало раздела правил. Вторая пара **%%** является необязательной и указывается только тогда, когда в программе присутствует ***раздел программ пользователя***.

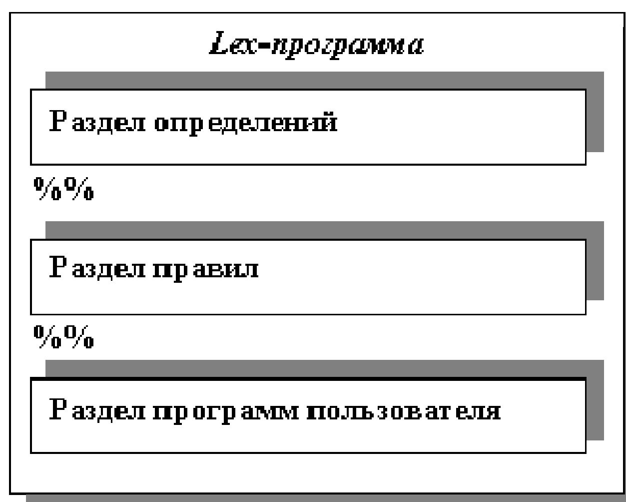

Рис. 18. Структура lex-программы

Все строки, в которых первая позиция занята не символом пробела или символом табуляции, относятся к lex-программе. Любая строка, не являющаяся частью правила (действием), которая начинается с символа пробела или символа табуляции, копируется в сгенерированную TP Lex'ом программу лексического анализатора. В lex-языке нет никаких специальных обозначений комментариев, но комментарии, описываемые по прави-

лам host-языка, могут быть включены в lex-программу по следующим правилам:

– формат комментариев должен соответствовать формату комментариев host-языка;

– комментарии можно указывать во всех разделах lex-программы. Однако в каждом разделе lex-программы комментарии указываются по-разному: в разделе определений комментарии должны начинаться не с первой позиции строки; в разделе правил комментарии можно указывать только внутри блоков, принадлежащих действиям; в разделе программ пользователя, комментарии указываются по правилам host-языка.

Рассмотрим подробнее способы оформления этих разделов.

4.1.1. Раздел определений

Раздел определений lex-программы помещается перед первой парой **%%**.

Раздел определений может состоять из следующих элементов:

1) *Начальные условия*: Для динамического изменения списка активных правил используются метки начальных условий. Предварительно они должны быть описаны в разделе определений. Описание меток начальных условий имеет следующий формат:

%Start Метка1 Метка2 ...

Пример:

%Start CommentOne CommentTwo

Ключевое слово *Start* можно указывать и так: *START*, или *S*, или *s*. За ключевым словом *Start* указывается список меток начальных условий. В качестве разделителей использууюся один и более пробелов или символов табуляции. *Метка1*, *Метка2, …* – метки начальных условий, составленные по правилам формирования меток host-языка. Начальные условия могут

отсутствовать. Команда описания начальных условий в разделе определений может быть указана только один раз.

2) *Определения*: Для сокращения регулярных выражений и их записи в более компактной форме используют определения, которые имеют следующий формат:

Имя_определения Трансляция

В качестве разделителя используется один и более пробелов или символов табуляции. Пример:

```
ALPHA        [A–Za–z_]
DIGIT        [0–9]
IDENTIFIER   {ALPHA}({ALPHA}|{DIGIT})*
```

Имя_определения – идентификатор, составленный по правилам host-языка. *Трансляция* – это регулярное выражение (или его часть), которое будет подставлено всюду: там, где указано *Имя_определения*, заключенное в «{…}» (смотрите третью строку этого примера). Определения могут отсутствовать.

3) *Фрагменты программ пользователя*: Фрагменты программ пользователя описываются на host-языке и заключаются в «%{…%}». Они будут включены в выходной файл TP Lex (в глобальную группу: раздел описания меток, раздел описания констант, раздел описания типов, раздел описания переменных, раздел описания процедур и функций). Фрагменты программ пользователя могут отсутствовать.

Порядок следования в разделе определений описанных выше элементов определяется следующими условиями:

– начальные условия всегда предшествуют определениям;

– фрагменты программ пользователя могут занимать любое положение.

4.1.2. Раздел правил

Всё, что указано после первой пары **%%** и до второй пары **%%** (или до конца lex-программы, если раздел программ пользователя отсутствует), относится к разделу правил.

Раздел правил lex-программы содержит описание лексического анализатора. Он может быть рассмотрен как большой оператор выбора, селектором которого является цепочка символов (последовательно считываемых из потока входного текста), которая сравнивается на принадлежность регулярным множествам (описываемым в виде регулярных выражений), каждому из которых поставлено в соответствие определенное действие. При завершении оператора выбора из потока входного текста к селектору добавляется один очередной символ. Оператор выбора выполняется до тех пор, пока не будет достигнут конец потока входного текста. Причём при однократном выполнении оператора выбора возможны два варианта:

– селектор не принадлежит ни одному регулярному множеству: в этом оператор выбора выполняется повторно, с добавлением к селектору очередного символа из потока входного текста;

– селектор принадлежит одному или нескольким регулярным множествам: в этом случае выполняется одно действие, соответствующее регулярному множеству (о приоритете выбора действий в случае принадлежности селектора нескольким регулярным множествам будет описано далее), селектор обнуляется, т.е. ему присваивается пустая цепочка, и оператор выбора выполняется повторно, с добавлением к селектору очередного символа из потока входного текста.

Раздел правил может состоять из следующих элементов:

1) *Фрагменты программ пользователя*. Раздел правил может начинаться с некоторых фрагментов программ пользователя на host-языке (заключенных в «%{…%}»). Они обрабатываются как локальные объявления стандартной процедуры yyaction выходного файла. Процедуры yyaction

осуществляет выполнение действий активных правил. Причём фрагменты программ пользователя должны предшествовать описанию активных и неактивных правил. Фрагменты программ пользователя в разделе правил могут отсутствовать.

2) *Активные и неактивные правила* (классификация правил будет рассмотрена далее). Каждое правило состоит из регулярного выражения и соответствующего ему действия. Регулярное выражение описывает одну или множество лексем, которые должны распознаваться из текста входного потока. Действия должны выполняться в том случае, если лексема, определяемая регулярным выражением, распознана. Регулярное выражение и соответствующее ему действие разграничены пробелами (или символами табуляции). Т.о. lex-правило имеет следующий формат:

Регулярное_выражение   Действие;

Обратите внимание на то, что действие должно стоять в той же строке, что и регулярное выражение, соответствующее этому действию, оно должно быть одно и заканчиваться «;» (для составных действий используется пара begin-end).

Действие может занимать несколько строк, если они, по крайней мере, отделены одним пробелом (или символом табуляции) от первой позиции. Действие в правиле может быть заменено символом «|», обозначающим, что действие для текущего правила такое же, как и для следующего.

Библиотека TP Lex обеспечивает доступ к различным переменным и стандартным подпрограммам, которые могут быть использованы при определении действий. В частности, переменная yytext:String содержит символьную строку с текущей распознанной лексемой, переменная yyleng:Byte – длину этой строки.

Регулярные выражения используются для описания лексемы или множества лексем. Они строятся из обычных конструкций, описывающих

символьные классы, последовательности символов, и операторов, определяющих повторения, альтернативы и т.д.

По регулярным выражениям, содержащимся в левой части правил, TP Lex строит детерминированный конечный автомат. Этот автомат осуществляет интерпретацию, а не компиляцию. Количество правил и их сложность не влияют на скорость лексического анализа, если только правила не требуют слишком большого объема повторных просмотров входной последовательности символов. Однако с ростом числа правил и их сложности растёт размер конечного автомата, интерпретирующего их и, следовательно, растёт размер программы на host-языке, реализующей этот конечный автомат.

4.1.3. Раздел программ пользователя

Все, что указано после второй пары **%%**, относится к разделу программ пользователя.

Раздел программ пользователя может содержать произвольный код на host-языке (такой, как вспомогательные подпрограммы (процедуры и функции) и/или основную программу), который просто копируется в конец выходного файла генератора TP Lex, без каких-либо изменений. Функции и процедуры, описанные в этом разделе, могут вызываться в действиях правил.

Этот раздел является необязательным.

4.2. Пример простейшей программы на lex-языке

Рассмотрим lex-программу, используя которую TP Lex строит конечный автомат, который распознаёт английские наименования месяцев и дней недели. Пример:

```
{Раздел определений}
%{
Uses LexLib;
%}
%%
^[ \t]*[jJ][aA][nN][uU][aA][rR][yY][ \t]* $          Writeln('Январь');
^[ \t]*[fF][eE][bB][rR][uU][aA][rR][yY][ \t]*$       Writeln('Февраль');
^[ \t]*[mM][aA][rR][cC][hH][ \t]*$                   Writeln('Март');
^[ \t]*[aA][pP][rR][iI][lL][ \t]*$                   Writeln('Апрель');
^[ \t]*[mM][aA][yY][ \t]*$                           Writeln('Май');
^[ \t]*[jJ][uU][nN][eE][ \t]*$                       Writeln('Июнь');
^[ \t]*[jJ][uU][lL][yY][ \t]*$                       Writeln('Июль');
^[ \t]*[aA][uU][gG][uU][sS][tT][ \t]*$               Writeln('Август');
^[ \t]*[sS][eE][pP][tT][eE][mM][bB][eE][rR][ \t]*$   Writeln('Сентябрь');
^[ \t]*[oO][cC][tT][oO][bB][eE][rR][ \t]*$           Writeln('Октябрь');
^[ \t]*[nN][oO][vV][eE][mM][bB][eE][rR][ \t]*$       Writeln('Ноябрь');
^[ \t]*[dD][eE][cC][eE][mM][bB][eE][rR][ \t]*$       Writeln('Декабрь');
^[ \t]*[mM][oO][nN][dD][aA][yY][ \t]*$               Writeln('Понедельник');
^[ \t]*[tT][uU][eE][sS][dD][aA][yY][ \t]*$           Writeln('Вторник');
^[ \t]*[wW][eE][dD][nN][eE][sS][dD][aA][yY][ \t]*$   Writeln('Среда');
^[ \t]*[tT][hH][uU][rR][sS][dD][aA][yY][ \t]*$       Writeln('Четверг');
^[ \t]*[fF][rR][iI][dD][aA][yY][ \t]*$               Writeln('Пятница');
^[ \t]*[sS][aA][tT][uU][rR][dD][aA][yY][ \t]*$       Writeln('Суббота');
^[ \t]*[sS][uU][nN][dD][aA][yY][ \t]*$               Writeln('Воскресенье');
.*                                                   Begin
                                                       {Сообщение
                                                         об ощибке}
                                                       Writeln('ОШИБКА!')
                                                     End;
\n                                                    ;
%%
   {Раздел программ пользователя}
Begin
```

Ylex;

End.

Каждое регулярное выражение здесь определяет действие. Действие в каждом правиле данной lex-программы – это вывод русского значения найденного английского слова. В качестве оператора, выполняющего действие, используется библиотечная процедура языка Pascal.

В программе содержатся все разделы lex-программы. Раздел правил содержит только правила, их всего двадцать одно. Регулярное выражение первых девятнадцати правил определяет английское слово, написанное маленькими или большими латинскими символами (с учетом лидирующих и завершающих пробелов). Например, цепочка символов «Monday» (Понедельник) определена регулярным выражением «^[\t]*[mM][oO][nN][dD][aA][yY][\t]*$». По этому регулярному выражению будет выделена во входном потоке символов лексема «Monday», а по действию этого правила будет выведено «Понедельник». Наличие большой и малой буквы в квадратных скобках обеспечивает распознавание слова «Monday», написанного любыми латинскими символами. А регулярные выражения «^[\t]*» и «[\t]*$» обозначают произвольное количество лидирующих и завершающих пробелов соответственно.

Таким образом, данная lex-программа строит Pascal-программу, которая переводит на русский язык наименования месяцев и дней недели.

Допустим, lex-программа размещена в файле source.l. Тогда, чтобы получить лексический анализатор на Pascal, необходимо выполнить следующий набор команд:

Lex source.l

Tpc source.pas

TP Lex всегда, если не указано другое имя, строит выходной файл с именем, совпадающим с именем входного файла (в данном случае будет сгенерирован файл source.pas). Во второй строке этой последовательности

команд запускается Pascal-компилятор, который выводит результат в файл source.exe.

Sourse.exe может работать как фильтр в конвейере команд, как самостоятельная команда и в интерактивном режиме.

Если необходимо получить самостоятельную программу, как в данном случае, необходимо в файле source.l в разделе программ пользователя описать головной раздел программы.

Также важно учитывать подключение модуля библиотеки LexLib, т.к. он необходим для работы генерируемых TP Lex'ом лексических анализаторов, поскольку содержит сервисные процедуры и функции, используемые функцией yylex.

4.3. Правила записи регулярных выражений

Регулярные выражения определяют лексему. Регулярное выражение может содержать символы латинского и русского алфавитов в верхнем и нижнем регистрах, другие символы (цифры, знаки препинания и т.д.), специальные символы и символы-операторы lex-языка.

Операторы регулярных выражений обозначаются символами-операторами. К ним относятся: «^», «$», «/», «|»,«?», «*», «+», «\», «"…"», «(…)», «[…]», «{…}». Каждый из этих символов или пар скобок в регулярном выражении играет роль оператора. Если необходимо отменить специальное значение символа, обозначающего оператора, перед ним нужно поставить символ «\» или указать его в двойных кавычках.

Введем следующие обозначения:

C – символ регулярного выражения **R**. Символом может быть любой текстовый или специальный символ.

S – строка символов регулярного выражения **R**. Строка может состоять из одного или нескольких символов **C** регулярного выражения.

R, **R_1**, **R_2** – регулярные выражения. Регулярное выражение может состоять из строки **S** или совокупности строк с соответствующими символами-операторами lex-языка: «^», «$», «/», «|»,«?», «*», «+», «\», «”…”», «(…)», «[…]», «{…}».

Операторы регулярных выражений подробно описаны в таблице 1.

Таблица 1

Операторы регулярных выражений TP Lex

Выражение	*Назначение*
C	C
\C	C, даже если C оператор
S	S
“S”	S
.	Любой символ, кроме символа новой строки \n
^R	Если R начинает строку
R$	Если R заканчивает строку
[S]	Любой символ из S
[C_1-C_n]	Символ C_1, C_2, …, C_n
[^S]	Любой символ не из S
R*	0 и более раз R
R+	1 и более раз R
R?	0 или 1 раз R
R{m,n}	От m до n раз R
R{m}	m раз R
$R_1 \vert R_2$	R_1 или R_2
R_1R_2	R_1 и R_2
(R)	R
R_1/R_2	R_1, если за ним следует R_2

4.3.1. Специальные символы

В регулярных выражениях могут использоваться следующие специальные символы:

«.»	точка означает любой символ, кроме символа новой строки «\n»;
«\n»	означает символ новой строки;
«\r»	означает символ возврата каретки;
«\t»	означает символ горизонтальной табуляции;
«\b»	означает символ возврата на один шаг;
«\f»	означает символ перевода формата;
«\NNN»	означает указание символа его восьмеричным кодом NNN, например, \041 обозначает символ «!».

4.3.2. Операторы контекстно-зависимого выбора

Два самых простых оператора из данной группы это «^» и «$».

Оператор «^» обозначает, что цепочка символов, принадлежащая регулярному множеству (описываемому регулярным выражением, стоящим после оператора «^»), учитывается только тогда, когда она стоит после символа «\n», т.е. является первой в строке. Например: ^d обозначает символ «d», если он является первым символом строки; ^abc обозначает цепочку символов «abc», если она начинает строку.

Оператор «$» обозначает, что цепочка символов, принадлежащая регулярному множеству (описываемому регулярным выражением, стоящим перед оператором «$»), учитывается только тогда, когда она стоит перед символом «\n», т.е. является последней в строке. Например: d$ обозначает символ «d», если он является последним символом строки; abc$ обозначает цепочку символов «abc», если она завершает строку.

Следующий последний оператор контекстно-зависимого выбора – оператор специального выбора «/», который означает, что цепочка симво-

лов, принадлежащая регулярному множеству (описываемому регулярным выражением, стоящим перед оператором «/»), распознается только тогда, когда за ней следует цепочка символов, принадлежащая регулярному множеству, описываемому регулярным выражением, стоящим поле оператора «/». Например: ab/cd обозначает, что цепочка «ab» учитывается только тогда, когда за ней следует цепочка «cd».

4.3.3. Оператор альтернативного выбора

Оператор «|» обозначает альтернативы регулярных выражений, например: ab|cd обозначает или цепочку «ab», или цепочку «cd».

4.3.4. Оператор необязательного присутствия

Оператор «?» означает необязательное присутствие регулярного выражения, например:

_?[A–Za–z]* обозначает, что перед цепочкой любого количества латинских букв может быть необязательный знак подчеркивания;

–?[0–9]+ обозначает любое целое число с необязательным знаком минус впереди.

4.3.5. Операторы итерации

Когда необходимо указать повторяемость вхождения регулярного выражения, используют операторы итерации «*» или «+».

Оператор «*» означает любое (в том числе и 0) число повторений регулярного выражения. Например:

R* обозначает множество всех цепочек, образованных любым числом повторений цепочек, описываемых регулярным выражением R, т.е. {ε, R, RR, RRR, …};

abc* обозначает множество всех цепочек, начинающихся с «ab» и заканчивающихся любым числом повторений символа «c», т.е. описывает регулярное множество {ab, abc, abcc, abccc, ….};

[ab]* обозначает множество всех цепочек, образованных любым количеством повторений «a» и «b», т.е. описывает регулярное множество {□, a, aa, …, aaaa, …, b, bb, …, ab, aab, …, aab, …};

[A–ZA–Яa–za–я_0–9]* обозначает множество цепочек, полученных любым количеством операций конкатенации таких подцепочек, что каждая подцепочка образована любым числом повторений любого символа, являющегося либо русской или латинской буквой, либо знаком подчеркивания или цифрой.

Оператор «+» означает одно и более повторений регулярного выражения. Например:

R+ обозначает множество всех цепочек, образованных одним и более числом повторений цепочек, описываемых регулярным выражением R, т.е. {R, RR, RRR, …};

a+ обозначает множество всех цепочек, образованных одним или более числом повторений символа «a», т.е. описывает регулярное множество {a, aa, …, aaaa, …};

[A–z]+ обозначает множество цепочек, полученных любым количеством операций конкатенации таких подцепочек, что каждая подцепочка образована любым числом повторений любой латинской буквы.

4.3.6. Операторы преобразования

Операторы преобразования используются для преобразования символов-операторов и некоторых специальных символов lex-языка в текстовые символы. В lex-языке для осуществления подобного рода преобразований существуют два оператора : «”…”» и «\».

Оператор «”…”» обозначает то, что вне зависимости от того, какой символ находится между парой кавычек, он должен быть воспринят как текстовый символ, например:

abc+ обозначает регулярное множество {abc, abcc, …, abcc…c};

“abc+” обозначает цепочку символов «abc+».

Таким образом, заключая в кавычки каждый не алфавитно-цифровой символ, используемый как текстовый, пользователь может не запоминать список всех специальных символов и символов-операторов lex-языка, чтобы не допустить ошибки при описании регулярного выражения.

Символ-оператор может быть преобразован в текстовый символ, если ему предшествует оператор «\». Например:

abc\+ обозначает цепочку символов «abc+».

Чтобы использовать в регулярном выражении символ «\», необходимо в него ввести «\\». Например:

ab\\c обозначает цепочку символов «ab\c».

Так как пробелы и признаки табуляции используются для разделения регулярных выражений и действий, то для включения символа пробела или символа табуляции в регулярное выражение их необходимо заключить либо в кавычки, либо в «[…]». Для символа табуляции можно использовать так же специальное обозначение «\t».

4.3.7. Оператор группирования

Оператор «(…)» предназначен для использования в сложных регулярных выражениях, с целью группирования их отдельных элементов, например:

(ab|cd+)?(ef)* обозначает регулярное множество {ab, abef, abefef, …, abef…ef, cd, cdef, cdefef, …, cdef…ef, □, ef, efef, …, efef…ef}

4.3.8. Оператор определения символьного класса

Оператор определения символьного класса «[…]» описывает класс, состоящий из символов, заключенных в квадратные скобки.

Например:

[abc] обозначает один символ, которым может быть либо «a», либо «b» или «c».

В пределах «[]» большинство операторов игнорируются и воспринимаются как текстовые символы. Только три оператора являются специальными: «\», «^» и «–».

Оператор «–» указывает диапазон. Например:

[a–z0–9_] обозначает один символ, которым может быть либо любая строчная латинская буква, либо любая цифра или знак подчеркивания.

Если необходимо включить символ «–» в символьный класс, он должен занимать в этом классе первую или последнюю позицию.

Оператор «^» наполнения означает, что символьный класс должен быть наполнен всеми текстовыми символами, исключая символ, перед которым стоит оператор «^». В символьном классе оператор «^» должен стоять в первой позиции слева. Например,

[^abc] включает все текстовые символы, кроме «a», «b» и «c».

[^a–zA–Z] включает все текстовые символы, кроме строчных и прописных латинских букв.

Оператор «\» имеет в пределах скобок символьного класса обычное для него назначение.

4.3.9. Оператор повторений и подстановки

Оператор «{…}» повторения регулярного выражения (если в операторе заданы список чисел или число) или подстановку (если в операторе задано имя).

Если оператор «{…}» обозначает повторения регулярного выражения, то он может быть записан в двух вариантах:

1) R{n,m}, где n и m натуральные числа, m>n. В данной записи оператор означает повторения регулярного выражения R от n до m. Например:

abc{2,7} обозначает регулярное множество {abcc, abccc, abcccc, abccccc, abcccccc, abccccccc};

2) R{n}, где n – натуральное число. В данной записи оператор означает n повторений регулярного выражения R. Например:

a(bc){4} обозначает цепочку символов «abcbcbcbc».

Если оператор «{…}» обозначает подстановку, то он записывается в следующем виде:

{***имя***}, где ***имя*** – предварительно описанное имя в разделе определений.

Пример:

```
%{
Uses LexLib;
%}
ALPHA       [A–Za–z_]
DIGIT        [0–9]
IDENTIFIER {ALPHA}({ALPHA}|{DIGIT})*
%%
{IDENTIFIER}                                   writeln(yytext);
.|\n                                           ;
 %%
Begin
  yylex;
End.
```

В данной записи оператор {IDENTIFIER} обозначает регулярное выражение, предварительно описанное в разделе определений как IDENTIFIER.

TP Lex построит лексический анализатор, который будет определять и выводить все идентификаторы Pascal-программы. В этом примере {IDENTIFIER} будет заменен на {ALPHA}({ALPHA}|{DIGIT})*, а затем на [A–Za–z_]([A–Za–z_]|[0–9])*. yytext – это внешний массив строкового типа программы лексического анализатора, которую стоит Lex. yytext формируется в процессе чтения входного потока текста и содержит цепочку символов, для которой установлено соответствие какому-либо регулярному выражению. Этот массив доступен пользовательским разделам lex-программы.

Оператор writeln выводит каждый идентификатор на новой строке. Правило «.|\n ;» в разделе правил lex-программы используется для того, чтобы игнорировать во входном потоке текста все цепочки символов, которые не соответствуют регулярному выражению {IDENTIFIER}.

4.4. Действия в правилах lex-программы

Действие можно представлять либо как оператор lex-языка (из модуля библиотеки LexLib), например, «*Start(Метка);*», либо как оператор Pascal. Если имеется необходимость выполнить достаточно большой набор преобразований, то действие оформляют как блок Pascal-программы (он начинается зарезервированным словом begin и завершается зарезервированным словом end), содержащий необходимые фрагменты программы.

Действие в правиле указывается не менее чем через один пробел или символ табуляции после выражения (обязательно в той же строке, где и регулярное выражение), а его продолжение может быть указано в следующих строках только в том случае, если действие оформлено как блок Pascal-программы.

Область действия переменных, объявленных внутри блока, распространяется только на этот блок. Внешними переменными для всех дей-

ствий будут являться только те переменные, которые объявлены в разделе определений lex-программы.

Действия в правилах lex-программы выполняются, если правило активно и если автомат распознает цепочку символов из потока входного текста как соответствующую регулярному выражению данного правила. Однако одно действие выполняется всегда – оно заключается в копировании потока входных символов в поток выходных символов. Это копирование осуществляется для всех входных строк, которые не соответствуют регулярным выражениям, описанным в активных правилах lex-программы. Комбинация символов, не учтенная в правилах и появившаяся на входе, будет напечатана на выходе. Можно сказать, что действие – это то, что делается вместо копирования потока входных символов в поток выходных символов. Часто бывает необходимо не копировать в поток выходных символов некоторую цепочку символов, которая удовлетворяет некоторому регулярному выражению. Для этой цели используется пустой оператор Pascal, например:

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

Это правило игнорирует (запрещает) вывод пробелов, символа табуляции и символа новая строка. Запрет выражается в том, что на указанные символы в потоке входного текста осуществляется действие «;» – пустой оператор Pascal, и эти символы не копируются в поток выходных символов.

Существует возможность для нескольких регулярных выражений указывать одно действие. Для этого используется символ «|», который указывает, что действие данного правила совпадает с действием для следующего.

Результат будет тот же, что и в примере, указанном выше.

Когда необходимо вывести или преобразовать текст, соответствующий некоторому регулярному выражению, используется внешний массив

символов, который формирует TP Lex. Он имеет имя yytext и доступен в действиях правил. Например:

```
[A–Z]+                    Writeln(yytext);
```

По этому правилу распознается слово, которое содержит прописные латинские буквы и которое выводится с помощью процедуры Writeln, если оно выделено. Операция вывода распознанного выражения используется очень часто, поэтому имеется сокращенная форма записи этого действия:

```
[A–Z]+                    Echo;
```

Результат действия этого правила будет аналогичен результату предыдущего примера. В модуле библиотеки LexLib оператор Echo определён как процедура.

Когда необходимо знать длину обнаруженной последовательности символов, используется счетчик найденных символов yyleng, который также доступен в действиях. Например:

```
[A–Z]+                    Writeln(yytext[yyleng–1]);
```

В этом примере будет выводиться последний символ слова, соответствующего регулярному выражению [A-Z]+. Рассмотрим еще один пример:

```
[A–Z]+                begin
                        number_word:=number_word+1;
                        number_aplha:=number_alpha+yyleng
                      end
```

Здесь ведется подсчет числа распознанных слов и количества символов во всех словах.

4.5. Активные и неактивные правила. Оператор-метка

Раздел правил lex-программы может содержать активные и неактивные (помеченные) правила. Они могут быть указаны в любом порядке, в

том числе могут быть «перемешанными» в разделе правил. Активные правила выполняются всегда. Неактивные выполняются только по ссылке на них процедурой *Start()*, т.е. в тех случаях, когда выполняется некоторое начальное условие.

Начальные условия lex-программы помещаются в раздел определений, а неактивные правила помечаются соответствующими условиями. Оператор *Start* позволяет указать список меток начальных условий lex-программы, а процедура *Start()* позволяет активировать правила, помеченные метками начальных условий. Активные правила имеют следующий синтаксис:

Регулярное_выражение Действие

Неактивные правила имеют следующий синтаксис:

<Метка>Регулярное_Выражение Действие

или

<Список_меток>Регулярное_Выражение Действие

где *Список_меток* имеет вид:

Метка1, Метка2, …

Любое правило должно начинаться с первой позиции строки, пробелы и символы табуляции недопустимы – они используются как разделители между регулярным выражением и действием в правиле.

Важно отметить следующее. Если lex-программа содержит активные и неактивные правила, то активные правила работают всегда. Оператор «*Start(Метка);*» просто расширяет список активных правил, активируя правила, помеченные меткой *Метка*. А оператор «*Start(0);*» удаляет из списка активных правил все помеченные правила, которые до этого были активированы. Кроме того, если из помеченного и активного в данный момент времени правила осуществляется действие «*Start(Метка)*;», то из помеченных правил активными останутся только те, которые помечены меткой *Метка*.

Рассмотрим пример:

```
Uses LexLib;
%Start IGNORE COMMENT
CommentBegin                    ”/*”
CommentEnd                      ”*/”
%%
<IGNORE>{CommentBegin}          begin
                                  Echo;
                                  Start(COMMENT)
                                end;
<IGNORE>.|\n                    ;
    <COMMENT>[^*]*              Echo;
<COMMENT>[^/]                   Echo;
<COMMENT>{CommentEnd}           begin
                                  Echo;
                                  Writeln;
                                  Start(IGNORE)
                                end;
%%
Begin
 Start(IGNORE);
 Yylex;
End.
```

TP Lex построит лексический анализатор, который выделяет комментарии в Си-программе и записывает их в стандартный файл вывода. Программа начинается с фрагмента программы пользователя

Uses LexLib.

Вслед за ним идёт ключевое слово *Start*, которое указано после символа «%». За ключевым словом Start указан список меток начальных условий IGNORE и COMMENT.

Оператор «<Метка>R» означает – регулярное выражение **R**, если анализатор находится в начальном условии *Метка*.

Процедура *Start(IGNORE)* переводит анализатор в начальное условие IGNORE (смотрите первое правило раздела правил этой lex-программы). После этого анализатор уже находится в новом состоянии, и теперь разбор входного потока символов будет осуществляться теми правилами, которые начинаются оператором «<IGNORE>». Например, правило:

```
<IGNORE>{CommentBegin}    begin
                           Echo;
                           Start(COMMENT)
                          end;
```

выполняется только тогда, когда во входном потоке символов будет обнаружено начало комментариев («/*»). В этом случае анализатор записывает в стандартный файл вывода цепочку символов «/*». Процедура *Start(COMMENT)* переводит лексический анализатор в начальное условие COMMENT, и теперь разбор входного потока символов будет осуществляться теми правилами, которые начинаются оператором «<COMMENT>».

Lex-программа может содержать несколько помеченных начальных условий. Например, если lex-программа начинается строкой:

%Start ONE TWO THREE FOUR,

то это означает, что она управляет четырьмя начальными состояниями анализатора. В каждое из этих начальных состояний анализатор можно перевести, используя процедуру *Start()*.

Каждое правило, перед которым указан оператор типа «<Метка>**»,** мы будем называть помеченным правилом. Метка формируется так же, как и метка в языке Pascal.

Количество помеченных правил не ограничивается. Кроме того, разрешается одно правило помечать несколькими метками, например:

<Метка1,Метка2,Метка3>Регулярное_Выражение Действие

Запятая – обязательный разделитель списка меток условий.

Рассмотрим ещё один пример с несколькими начальными условиями:

```
 %{
Uses LexLib;
%}
%Start ONE TWO THREE
ALPHA
DIGIT                              [A–Za–z_]
IDENTIFIER                         [0–9]
%%                                 {ALPHA}({ALPHA}|{DIGIT})*
^#                                 %%
^[ \t]*([a-z]*[ \t]*)?main         Start(ONE);
^[ \t]*{IDENTIFIER}                Start(TWO);
[ \t]                              Start(THREE);
\n                                 ;
<ONE>define                        Start(0);
<ONE>include                       Writeln('Определение.');
<ONE>ifdef                         Writeln('Включение.');
<TWO>"("[ \t]*(void[ \t]*)?")"     Writeln('Условная компиляция.');
<TWO>"("[^\,\n]*(","[^\,\n]*)*")"  Writeln('main без аргументов.');
<THREE>":"/[ \t]                   Writeln('main с аргументами.');
%%                                 Writeln('Метка.');
Begin
 yylex;
End.
```

Программа содержит активные и неактивные правила. Все неактивные правила помечены, перед ними указана метка начального условия. Lex-программа управляет тремя начальными условиями, в соответствии с которыми активируются помеченные правила.

В результате работы Lex мы получим лексический анализатор, который будет находить в Си-программе строки препроцессора Си-компилятора, выделять функцию main, распознавая с аргументами она или

без них, определять метки. Лексический анализатор не выводит ничего, кроме сообщений о выделенных лексемах.

4.6. Порядок выполнения активных правил

Раздел правил lex-программы, как уже было сказано выше, может содержать *активные* и *неактивные правила*, размещенные в любом порядке. В процессе работы лексического анализатора список активных правил может видоизменяться за счет действий процедуры Start(). В процессе распознавания символов потока входного текста может оказаться так, что одна цепочка символов будет удовлетворять нескольким правилам, и, следовательно, возникает проблема выбора правила и, соответственно, выполняемого действия.

Для разрешения этой коллизии можно использовать квантование (разбиение) регулярных выражений этих правил lex-программы на такие новые регулярные выражения, которые дадут, по возможности, однозначное распознавание лексемы. Однако, когда это не сделано, TP Lex использует определенный механизм разрешения такого противоречия, который состоит в следующем:

– выбирается действие того правила, которое распознает наиболее длинную последовательность символов из входного потока текста;

– если несколько правил распознают последовательности символов одной длины, то выполняется действие того правила, которое записано первым в списке раздела правил lex-программы.

Рассмотрим пример:

```
…                          …
[Мм][Аа][Йй]               Echo;
[А-Яа-я]+                  Echo;
```

Слово «Май» распознают оба правила, однако, выполнится первое из них, так как и первое, и второе правило распознали лексему одинако-

вой длины (3 символа). Если во входном потоке текста будет, допустим, слово «майский», то первые 3 символа удовлетворяют первому правилу, а все 7 символов удовлетворяют второму правилу. Следовательно, выполнится второе правило, так как ему удовлетворяет более длинная последовательность символов.

5. МОДУЛЬ БИБЛИОТЕКИ LEXLIB

Модуль библиотеки LexLib необходим как для работы генератора лексических анализаторов TP Lex, так и созданных на базе генератора TP Lex лексических анализаторов. Он также обеспечивает доступ к потокам ввода/вывода, используемым лексическим анализатором, и обеспечивает доступ к сервисным функциям и процедурам, которые могут использоваться в действиях для правил.

Однако можно модифицировать модуль библиотеки LexLib по отношению к специализированным приложениям, создаваемым пользователем. В частности, можно обеспечить другой набор функций ввода/вывода, например, если необходимо считывать поток входного текста из файлов различных типов.

5.1. Доступные переменные модуля библиотеки LexLib

Ниже описываются переменные, которые могут использоваться пользователем в действиях правил lex-программы:

yytext (String) – содержит цепочку символов, соответствующую распознанному текущему регулярному выражению;

yyleng (Byte) – длина цепочки символов, соответствующей распознанному текущему регулярному выражению;

yyline (String) – содержит текущую входную строку символов;

yylineno (Integer) – содержит номер строки в файле;

yylinecolno (Integer) – содержит номер позиции в строке.

Значения yylineno и yylinecolno часто используются при выводе сообщений об ошибках, однако они будут содержать правильное значение, если не существует никакого перепросмотра строки из потока входного текста:

yyinput (Text) – содержит имя входного текстового файла, используемого лексическим анализатором;

yyoutput (Text) – содержит имя выходного текстового файла, используемого лексическим анализатором.

По умолчанию переменные yyinput и yyoutput связаны со стандартным устройством ввода и вывода CON, но можно изменить эти значения, для адаптации своего приложения.

5.2. Сервисные процедуры и функции модуля библиотеки LexLib

Рассмотрим сервисные функции и процедуры, которые могут использоваться пользователем, при определении действий в правилах lex-программы.

5.2.1. Функция get_char. Процедуры unget_char и put_char

Функция get_char, процедуры unget_char и put_char используются для осуществления доступа к файлам ввода и вывода. Чтение входного потока осуществляется по строкам. Ввод буферизован, что позволяет повторно просматривать текст, используя функцию unget_char.

В буфер ввода считывается символьная строка из потока входного текста, которая должна быть просмотрена. Когда входной буфер пуст, в него считывается новая строка. Символы могут быть возвращены в буфер ввода процедурой unget_char. При достижении конца входного текста возвращается пустой символ.

При обработке потока входного текста также устанавливаются значения переменных yyline, yylineno и yylinecolno.

Так как остальная часть библиотеки LexLib зависит только от этих трёх сервисных модулей: функции get_char, процедур unget_char и put_char, т.е. не имеется никаких других прямых ссылок к переменным yyinput и yyoutput или на буфер ввода, то можно легко заменить get_char, unget_char и put_char другим подходящим набором операций, в зависимости от того, как необходимо осуществлять ввод/вывод.

Функция get_char читает и возвращает символ из входного буфера. Если возвращает пустой символ, то достигнут конец потока входного текста. Описание функции имеет следующий вид:

function get_char : Char;

Процедура unget_char возвращает один символ во входной буфер для повторного чтения. Описание процедуры имеет следующий вид:

procedure unget_char (c : Char);

Процедура put_char выводит один символ в поток выходного текста. Описание процедуры имеет следующий вид:

procedure put_char (c : Char);

5.2.2. Процедура echo

Процедура echo выводит на экран символьную строку, соответствующую текущему регулярному выражению. Описание процедуры имеет следующий вид:

procedure echo;

5.2.3. Процедура reject

Процедура reject отменяет текущее соответствие регулярному выражению цепочки символов и выполняет следующую альтернативу. Описание процедуры имеет следующий вид:

procedure reject;

Обычно TP Lex разделяет входной поток, не осуществляя поиск всех возможных соответствий каждому выражению. Это означает, что каждый символ рассматривается один и только один раз. Предположим, что мы хотим подсчитать все вхождения цепочек makefile и file во входном тексте. Для этого мы могли бы записать следующие правила:

```
makefile   s:=s+1;
file       h:=h+1;
.|\n       ;
```

Так как makefile включает в себя file, лексический анализатор, описываемый этими правилами, не будет распознавать те вхождения file, которые включены в makefile, так как, прочитав один раз makefile, эти символы он не вернет во входной поток.

Иногда желательно переопределить этот выбор. Действие процедуры reject означает «выбрать следующую альтернативу». Это приводит к тому, что каким бы ни было правило, после него необходимо выполнить второй выбор. Соответственно изменится и положение указателя во входном потоке. Лексический анализатор в этом случае будет описываться следующими правилами:

```
makefile                  begin
                            s:=s+1;
                            reject
                          end;
file                      begin
                            h:=h+1;
                            reject
                          end
.|\n                      ;
```

Здесь после выполнения одного правила символы возвращаются назад во входной поток, и выполняется другое правило. Процедура reject полезна в том случае, когда она применяется для определения всех вхождений какого-либо объекта, причем вхождения могут перекрываться или включать друг друга. Предположим, необходимо получить из одного потока таблицу всех двухбуквенных сочетаний строчных латинских букв, которые обычно перекрываются (например, слово the содержит как th, так и he). Lex-программа, описывающая данный лексический анализатор, представлена ниже:

```
%{
Uses LexLib;
Var digram : array ['a'..'z','a'..'z'] of Integer;
%}
 %%
[a-z][a-z]                       Begin
                                  inc(digram[yytext[1],yytext[2]]);
                                  reject
                                 end;
.                                |
\n                               ;
%%
var c,d : char;
Begin
 For c := 'a' to 'z' do for d := 'a' to 'z' do
  digram[c,d] := 0;
  if yylex=0 then
    for c := 'a' to 'z' do for d := 'a' to 'z' do
      if digram[c,d]<>0 then
          writeln(c,d,': ',digram[c,d]);
End.
```

Здесь процедура reject используется для выделения буквенных пар, начинающихся на каждой букве, а не на каждой следующей.

5.2.4. Процедуры return и returnc

Процедуры return и returnc устанавливают значение, возвращаемое функцией yylex. Описания этих процедур имеют следующий вид:

Procedure return (n : Integer);

Procedure return(c : Char);

5.2.5. Процедура Start

Процедура start переводит лексический анализатор в заданное состояние. Причём 0 – заданное по умолчанию состояние, другие значения – значения определяемые пользователем. Описание процедуры имеет вид:

procedure start (state : Integer);

5.2.6. Процедуры yymore и yyless

Процедура yymore добавляет символьную строку, соответствующую текущему регулярному выражению, в конец предыдущей распознанной символьной строки. Описание процедуры имеет следующий вид:

procedure yymore;

В обычной ситуации содержимое yytext обновляется всякий раз, когда на входе появляется следующая строка. Напомним, что в yytext всегда находятся символы распознанной последовательности. Иногда возникает необходимость добавить к текущему содержимому yytext следующую распознанную цепочку символов. Для этой цели используется процедура yymore.

В некоторых случаях возникает необходимость использовать не все символы распознанной последовательности в yytext, а только необходимое их число. Для этой цели используется процедура yyless. Описание процедуры имеет следующий вид:

procedure yyless (n : Integer);

Формат ее вызова:

yyless(n),

где n указывает, что в данный момент необходимы только n символов строки в yytext. Остальные найденные символы будут возвращены во входной поток.

Пример использования фунцкии yymore:

```
\"[^"]*                    begin
                             if yytext[yyleng - 1] = '\' then
                               yymore()
                             else
                              { здесь должна быть часть
                                программы, обрабатывающая
                                закрывающую кавычку.      }
                           end;
```

В этом примере распознаются строки символов, взятые в двойные кавычки, причем символ «двойная кавычка» внутри этой строки может изображаться с предшествующей косой чертой. Анализатор должен распознавать кавычку, ограничивающую строку, и кавычку, являющуюся частью строки, когда она изображена как «\"».

Допустим, на вход поступает строка «"абв\"где"». Сначала будет распознана цепочка «"абв\», и так как последним символом в этой цепочке будет символ «\», выполнится вызов функции yymore(). В результате к цепочке «абв\» будет добавлено «"где», и в строковой переменной yytext мы получим: «"абв\"где», что и требовалось.

Пример использования фунции yyless:

```
…
=–[A–Za–z]       begin
                   Writeln('Оператор (= –) двусмысленный.');
```

```
                          yyless(yyleng – 2);
                         (* здесь необходимо указать действия для
                              случая ’=–’  *)
                         end;
…
```

В этом примере разрешается двусмысленность выражения «= – буква» в языке Си. Это выражение можно рассматривать как

«=– буква» (равносильно « – =»)

или

«=–буква»

Предположим, что желательно эту ситуацию рассматривать как «=– буква» и выводить предупреждение. Указанное в примере правило распознает эту ситуацию и выводит предупреждение. Затем, в результате вызова «yyless(yyleng - 2);» два символа «–буква» будут возвращены во входной поток, а знак «=» останется в yytext для обработки, как в нормальной ситуации.

Таким образом, при продолжении чтения входного потока уже будет обрабатываться цепочка «–буква», что и требовалось.

5.2.7. Функция yywrap

Функция yywrap используется для определения конца файла, из которого лексический анализатор читает поток символов. Описание функции имеет следующий вид:

function yywrap : Boolean;

Если yywrap возвращает TRUE, лексический анализатор прекращает работу. Однако иногда имеется необходимость начать ввод данных из другого источника и продолжить работу.

В этом случае пользователь должен написать свою функцию yywrap, которая организует новый входной поток и возвращает FALSE, что слу-

жит сигналом к продолжению работы анализатора. По умолчанию yywrap всегда возвращает TRUE при завершении входного потока символов.

В lex-программе невозможно записать правило, которое будет обнаруживать конец файла. Единственный способ это сделать – использовать фунцию yywrap. Эта функция также удобна, когда необходимо выполнить какие-либо действия по завершению входного потока символов, определив в разделе программ пользователя новый вариант функции yywrap.

Пример:

<table>
<tr><td colspan="2">%{
Uses LexLib;
%}
%Start ONE TWO THREE
%{
(* Строится лексический анализатор, который распознаёт наличие включений файлов в Си-программе, условных компиляций, макроопределений, меток и головной функции main. Анализатор ничего не выводит, пока осуществляется чтение входного потока текста, а по его завершении выводит статистику. *)
%}</td></tr>
<tr><td>ALPHA</td><td>[A–Za–z_]</td></tr>
<tr><td>DIGIT</td><td>[0–9]</td></tr>
<tr><td>IDENTIFIER</td><td>{ALPHA}({ALPHA}|{DIGIT})*</td></tr>
<tr><td>%{</td><td></td></tr>
<tr><td>Var a1,a2,a3,b1,b2,c : Integer;</td><td></td></tr>
<tr><td>%}</td><td></td></tr>
<tr><td>%%</td><td></td></tr>
<tr><td>^#</td><td>Start(ONE);</td></tr>
<tr><td>^[\t]*([a-z]*[\t]*)?main</td><td>Start(TWO);</td></tr>
<tr><td>^[\t]*{IDENTIFIER}</td><td>Start(THREE);</td></tr>
<tr><td>[\t]</td><td>;</td></tr>
<tr><td>\n</td><td>Start(0);</td></tr>
<tr><td><ONE>define</td><td>a1:=a1+1;</td></tr>
<tr><td><ONE>include</td><td>a2:=a2+1;</td></tr>
</table>

<ONE>ifdef	a3:=a3+1;
<TWO>”(”[\t]*(void[\t]*)?”)”	b1:=b1+1;
<TWO>”(”[^\,\n]*(”,”[^\,\n]*)*”)”	b2:=b2+1;
<THREE>”:”/[\t]	c:=c+1;

```
%%
Function yywrap: Boolean;
Begin
 if (b1=0) and (b2=0) then
  Writeln('В программе отсутствует функция main.');
 if (b1>1) or (b2>1) or ((b1>=1) and (b2>=1)) then
  Writeln('Многократное определение функции main.')
 Else
 Begin
   if b1=1 then
    Writeln('Функция main без аргументов.');
   if b2=1 then
    Writeln('Функция main с аргументами.')
  end;

 Writeln('Определений: ', a1);
 Writeln('Включений файлов: ', a2);
 Writeln('Условных компиляций: ', a3);
 Writeln('Меток: ', c);
 yywrap:=TRUE
End;
Begin
 a1:=0; a2:=0; a3:=0; b1:=0; b2:=0; c:=0;
 yylex;
End.
```

Оператор yywrap:=TRUE в функции yywrap указывает, что лексический анализатор должен завершить работу. Если необходимо продолжить работу анализатора для чтения данных из нового файла, нужно указать yywrap:=TRUE , предварительно осуществив операции закрытия и откры-

тия файлов. В этом случае, анализатор продолжит чтение и обработку входного потока. Однако, если yywrap не возвращает TRUE, то это приводит к бесконечному циклу.

6. ГЕНЕРАТОР ТАБЛИЦ КЛЮЧЕВЫХ СЛОВ

6.1. Утилита Kwtbl

Kwtbl – утилита генерации таблиц ключевых слов для последующего использования их в лексических анализаторах, таких как yylex, сгенерированных TP Lex. Ниже описан модифицированный вариант утилиты kwtbl.

Утилита Kwtbl на основе файла описания ключевых слов генерирует выходной файл (на языке Pascal), содержащий таблицу ключевых слов и функцию поиска в этой таблице.

Командная строка вызова утилиты kwtbl:

kwtbl kwtbl-file[.tbl] [output-file[.pas]],

где kwtbl-file.tbl – файл описания ключевых слов; output-file.pas – выходной файл.

В языках с большим количеством ключевых слов часто для наиболее эффективного анализа зарезервированных слов используют общий образец распознавания, а их организовывают в виде таблицы ключевых слов и, используя функцию поиска в таблице ключевых слов, определяют код для соответствующего ключевого слова.

Большое количество описаний ключевых слов (в виде регулярных выражений в разделе правил) могут вызывать переполнение TP Lex, в таких случаях полезной оказывается утилита Kwtbl.

Утилита Kwtbl читает список пар (ключевое слово, код) из входного файла kwtbl-file.tbl, сортирует ключевые слова в алфавитном порядке и создает две таблицы-константы: одна содержит строки ключевого слова, а другая – соответствующие ключевым словам коды. К этим таблицам мож-

но обращаться через процедуру поиска в таблице ключевых слов, которая добавляется в конец выходного файла, создаваемого утилитой Kwtbl. Программный код для процедуры поиска находится в отдельном файле kwtbl.cod, который должен присутствовать или в текущем каталоге, или в каталоге, откуда была запущена утилита kwtbl. Без этого файла утилита kwtbl не генерирует процедуру поиска в таблице ключевых слов.

Рассмотрим типичный язык программирования с идентификаторами и некоторыми зарезервированными словами, которые по своей структуре напоминают идентификаторы.

Соответствующие правила lex-программы могли бы выглядеть следующим образом:

```
…                                …
begin                            return( _BEGIN_ );
do                               return( _DO_ );
enddo                            return( _ENDDO_ );
else                             return( _ELSE_ );
end                              return( _END_ );
if                               return( _IF_ );
endif                            return( _ENDIF_ );
output                           return( _OUTPUT_ );
read                             return( _READ_ );
then                             return( _THEN_ );
while                            return( _WHILE_ );
...                              ...
[A-Za-z]+    return( ID );       return( ID );
…                                …
```

Другой вариант – использовать утилиту kwtbl для представления ключевых слов в виде таблицы с последующим их поиском. Описание

ключевых слов во входном файле kwtbl-file.tbl для утилиты kwtbl будет выглядеть следующим образом:

```
…         …
begin     return( _BEGIN_ );
do        return( _DO_ );
enddo     return( _ENDDO_ );
else      return( _ELSE_ );
end       return( _END_ );
if        return( _IF_ );
endif     return( _ENDIF_ );
output    return( _OUTPUT_ );
read      return( _READ_ );
then      return( _THEN_ );
while     return( _WHILE_ );
…         ...
```

Каждое входное ключевое слово должно находиться на отдельной строке, и ключевое слово, и соответсвующий ему код должны быть разделены пробелом(ами) (и/или символом(ами) табуляции). Пустые строки игнорируются. Нет необходимости сортировать входные ключевые слова, т.к. утилита kwtbl делает это автоматически. Кодом для ключевого слова может быть любая символьная последовательность, которая является константой типа integer. На основе описания таблицы ключевых слов утилита kwtbl создает два массива констант:

```
Const
  nkws = 11;   { number of different keywords }
  kwsize = 6; { maximum size of keywords }

kwtbl : array [ 1..nkws ] of String[ kwsize ] = (
```

```
'BEGIN', 'DO',
'ELSE', 'END',
'ENDDO', 'ENDIF',
'IF', 'OUTPUT',
'READ', 'THEN',
'WHILE'
                                        );
kwcod : array [ 1..nkws ] of Integer = (
_BEGIN_, _DO_,
_ELSE_, _END_,
_ENDDO_, _ENDIF_,
_IF_, _OUTPUT_,
_READ_,  _THEN_,
_WHILE_                                 );
```

К этим таблицам можно обращаться через функцию поиска в таблице ключевых слов, названную kwlookup в стандартной версии файла kwtbl.cod:

function kwlookup(kw : String; var code : Integer) : Boolean;

Функция kwlookup проверяет, находится ли ключевое слово, заданное символьной строкой kw, в таблице ключевых слов и возвращает его код в переменную code. Если ключевое слово найдено в таблице ключевых слов, то функция kwlookup возвращает значение true, иначе – false.

Таким образом, используя утилиту генерации таблицы ключевых слов, указанную выше, lex-программу можно привести к следующему виду:

```
…                          …
[A-Za-z]+                  if  kwlookup( yytext, code ) then
                             return( code)
```

```
                                   else
                                     return( ID );
  …                                …
```

Это значительно уменьшает рабочие таблицы TP Lex и ускоряет процесс лексического анализа.

7. ОБЗОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ АВТОМАТИЗИРОВАННОЙ РАЗРАБОТКИ ТРАНСЛЯТОРОВ

7.1. Пакеты для разработки компиляторов

COCKTAIL – набор генераторов почти для всех фаз работы компилятора: REX – генератор сканеров; LALR и ELL – генератор синтаксических анализаторов для LALR(1) и LL(1); AST – генератор для абстрактных синтаксических деревьев; AG – генератор атрибутных вычислителей; PUMA – инструмент преобразования, основанный на сопоставлении образцов. (***http://www.first.gmd.de/cocktail/***).

ELI – пакет предоставляет решение для большинства задач, возникающих при создании языка программирования, – начиная со структурного анализа (решаемого с помощью средств типа LEX и YACC), через анализ имен, типов и значений к сохранению структур данных трансляции и получения выходного текста (***http://www.cs.colorado.edu/~eliuser/***).

GENTLE – интегрированная система, перекрывающая весь спектр задач по конструированию компиляторов. Gentle поддерживает распознавание языка, определение абстрактных синтаксических деревьев, интеллектуальный обход дерева, выбор оптимального кода для микропроцессоров и простой несинтаксический анализ для трансляции «источник – источник» (***http://www.first.gmd.de/gentle/***).

PCCTS – набор средств, помогающих в создании программ распознавания языка и трансляторов. Он состоит из трех инструментов: ANTLR – генератор парсеров, который функционирует подобно Yacc, но основан

на predicated LL(k); DLG – простой генератор лексических анализаторов в духе Lex; SORCERER – генератор синтаксических анализаторов, позволяющий программисту определять структуру данных дерева через грамматику. (***http://www.antlr.org/pccts133.html***).

7.2. Генераторы лексических и синтаксических анализаторов

ACCENT – компилятор компиляторов, не накладывающий никаких ограничений на грамматики. Никакой адаптации к специфическим методам синтаксического анализа, таким как LL(k) или LALR(k), не требуется. Поддерживает расширенную БНФ. (***http://www.combo.org/accent/***).

AFLEX & AYACC – аналогичны юниксовым инструментам Lex и Yacc, но написаны на Ада и генерируют на выходе программу на Ада. (***http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html***)

ANAGRAM – генератор синтаксических анализаторов LALR с возможностью автоматического восстановления после синтаксических ошибок (***http://www.parsifalsoft.com/***).

BTYACC – модифицированная версия Yacc, поддерживающая автоматический откат и семантическое устранение неоднозначности для разбора неоднозначных грамматик (***http://www.siber.com/btyacc/***).

BYACC (Berkeley Yacc) – свободно распространяемый генератор синтаксических анализаторов LALR(1). Он сделан как можно более совместимым с AT&T Yacc (***ftp://ftp.cs.berkeley.edu/ucb/4bsd/byacc.tar.Z***).

BISON – GNU версия Yacc (***http://www.combo.org/lex_yacc_page***).

BISON/EIFFEL – генератор синтаксических анализаторов Bison с возможностью вывода в формате Eiffel.

(***http://www.spin.ch/~kalberer/bison/index.htm***)

COGENCEE – генератор компиляторов для Delphi, разработанный на основе Coco (***http://www.cocolsoft.com.au/cogen.htm***).

COCO (Coco/R) – генератор анализаторов, работающих по методу рекурсивного спуска и связанных с ними сканеров по атрибутным грамматикам (***http://cs.ru.ac.za/homes/cspt/cocor.htm***).

DEPOT4 – генератор нисходящих распознавателей, поддерживающий описание в стиле, схожем с синтаксически управляемой схемой перевода. Язык описания основан на РБНФ. Depot4 предназначен для использования неспециалистами при создании проблемно-ориентированных языков (***http://www.math.tu-dresden.de/wir/depot4/Depot4.html***).

EAG – компилятор компиляторов, основанный на Extended Affix Grammars (***ftp://hades.cs.kun.nl/pub/eag/***).

FLEX – GNU версия генератора сканнеров Lex.

(***http://www.combo.org/lex_yacc_page***)

FLEX/EIFFEL – версия генератора лексических анализаторов Flex. (***http://www.spin.ch/~kalberer/flex/index.htm***)

GOBO EIFFEL LEX & YACC – реализации Lex и Yacc для Eiffel. (***http://www.gobosoft.com/eiffel/gobo/***)

HAPPY – система генерации парсеров для Haskell, аналогичная инструменту Yacc для C. Как и Yacc, она берет файл, содержащий БНФ спецификацию грамматики, и производит модуль на Haskell, содержащий синтаксический анализатор этой грамматики. (***http://www.haskell.org/happy/***)

LEX – классический генератор лексических анализаторов AT&T, поставляемый с Unix (***http://www.combo.org/lex_yacc_page***).

LLGEN – инструмент для создания эффективного анализатора на основе рекурсивного спуска из ELL(1) грамматики. Грамматика может быть неоднозначная или более общая, чем ELL(1): LLgen предоставляет как статические, так и динамические средства для разрешения неоднозначности (***http://www.cs.vu.nl/~ceriel/LLgen.html***).

LISA – генератор таблично-управляемых лексических анализаторов и LL(1) синтаксических анализаторов по регулярным выражениям и БНФ. LISA поддерживает атрибутные вычислители Кеннеди – Уорена и Катаямы (***http://marcel.uni-mb.si/nikolaj/sigplan.htm***).

MANGO – генератор синтаксических анализаторов, включенный в систему Self. Синтаксические анализаторы Mango автоматически создают дерево разбора, а не просто предоставляют ловушки для вызова низкоуровневых функций преобразования во время анализа.

(***http://www.cs.ucsb.edu/oocsb/self/papers/mango.html***)

MUSKOX – аннотирует классы в РБНФ для LR(k) грамматик. Он предоставляет наследование грамматик и переопределение правил. Также поддерживает множественные анализаторы, запись/воспроизведение журнала трассировки и т.д. (***http://www.mastersys.com/***).

MKS LEX & YACC – Lex-совместимый генератор лексических анализаторов и Yacc-совместимый генератор синтаксических анализаторов для PC (***http://www.mks.com/solution/ly/***).

NEWYACC – внешний интерфейс к Yacc. Он предоставляет надмножество Yacc с трансляциями, подключенными к грамматикам в дополнение к действиям. Трансляции схожи с простыми синтаксически управляемыми схемами перевода в том значении, что они преобразуют, переупорядочивают, выбирают, приращивают и повторяют входной поток символов соответствующим образом просматривая завешенное дерево разбора (***ftp://flubber.cs.umd.edu/src/***).

PCYACC – в основном используется для разработки встроенных языков в продуктах третьих фирм, использующих языки типа SQL или SGML. Он содержит поддержку наиболее распространенных языков в виде исходного кода (***http://www.abxsoft.com/pcyacc.htm***).

PRECC – генератор компиляторов с бесконечным заглядыванием вперед для контекстно-зависимых грамматик. Спецификации описываются в РБНФ с наследуемыми и синтезируемыми атрибутами.

(***http://www.comlab.ox.ac.uk/archive/redo/precc.html***)

PROGRAMMAR (ProGrammar Developer's Toolkit) – интегрированный набор инструментов и утилит для создания, тестирования и отладки синтаксических анализаторов. Programmar включает в себя объектно-ориентированный язык, визуальную среду разработки и интерактивный отладчик (***http://www.programmar.com/***).

RDP – компилирует атрибутные LL(1) грамматики, украшенные семантическими действиями языка С, в компиляторы на основе рекурсивного спуска (***http://www.dcs.rhbnc.ac.uk/research/languages/rdp.shtml***).

RE2C – инструмент для генерации основанных на С распознавателей по регулярным выражениям. Сгенерированный код не привязан к какой-либо конкретной входной модели. (***http://www.tildeslash.org/re2c/index.html)***

S/SL – является языком программирования для конструирования компиляторов. Он включает последовательность, повторение и выбор; ввод, сравнение и вывод токенов; вывод сообщений об ошибках; подпрограммы; вызов семантических операций. (***ftp://ftp.cs.toronto.edu/pub/ssl.tar.Z***)

SCANGEN, LLGEN, LALRGEN – генераторы лексических и LL(1) и LALR(1) анализаторов, представленные в книге Фишера и Лебланка «Crafting a Compiler».

(***ftp://ftp.csc.ncsu.edu/pub/compilers/crafting_compiler/tools***)

TP LEX AND YACC – генератор лексических и синтаксических анализаторов для Turbo Pascal.

(***http://www.musikwissenschaft.uni-mainz.de/~ag/tply/tply.html)***

VISUALPARSE++ – предоставляет визуальный интерфейс, позволяющий любому программисту интерактивно изучать и применять технологии лексического и синтаксического анализа.

(***http://www.sand-stone.com/prod02.htm***)

YACC – классический генератор синтаксических анализаторов AT&T, поставляемый с Unix (***http://www.combo.org/lex_yacc_page***).

YACC++ – не просто набор классов-оберток C++ вокруг вывода Lex и Yacc. Yacc++ и Language Objects Library являются объектно-ориентированной версией Lex и Yacc. Среди возможностей – грамматические классы с наследованием, регулярные выражения, эффективно интегрированные в LR анализ, решения для включения файлов, ключевые слова в подстроках, вложенные комментарии и т.д.

(***http://www.world.std.com/~compres/***)

7.3. Системы атрибутной грамматики

ELEGANT (Exploiting Lazy Evaluation for the Grammar Attributes of Non-Terminals) – возник как генератор компиляторов, основанный на атрибутных грамматиках, и превратился в полноценный язык программирования.

(***http://www.research.philips.com/generalinfo/special/elegant/elegant.html***)

FNC-2 – система атрибутной грамматики, основанная на строго нециклических АГ, выполняющая расширенную оптимизацию используемой памяти (***http://www-rocq.inria.fr/oscar/FNC-2/littlefnc2.html***).

OX – обычные спецификации Yacc и Lex могут быть расширены синтезируемыми и наследуемыми атрибутами, написанными в синтаксисе С. OX допускает наиболее общий класс атрибутных грамматик. Пользователь может определять проход дерева для постдекорации. (***ftp://ftp.cs.iastate.edu/pub/ox/***)

7.4. Средства преобразования

APP – препроцессор сопоставления образов алгебраических типов для С++ (***http://www.primenet.com/~georgen/ app.html***).

KIMWITU – система, поддерживающая конструирование программ, использующих деревья или термы в качестве основных структур данных. (***http://wwwtios.cs.utwente.nl/kimwitu/***)

MEMPHIS – инструмент, который поддерживает определение и обработку абстрактных синтаксических деревьев. Предкомпилятор расширяет С/С++: определения доменов описывают типы данных в грамматическом стиле. Операторы сравнения обрабатывают эти данные, используя метод сопоставления образцов (***http://www.combo. org/memphis***).

RIGAL – является языком конструирования компиляторов. Главными структурами данных являются атомы, списки и деревья. Управляющие структуры основаны на улучшенном методе сопоставления образцов. (***ftp://ftp.ida.liu.se/pub/labs/pelab/rigal/***)

TXL – язык трансформационного программирования. Основная парадигма TXL заключается в трансформации входных данных в выходные, используя набор правил преобразования, описывающих, как различные части входных данных должны изменяться на выходе.

(***http://www.thetxlcompany.com/***)

TXL-3 – реализация языка программирования TXL для Модула-3. (***ftp://ftp-i3.informatik.rwth-aachen.de/pub/Modula-3-Contrib/txl-3***)

7.5. Генерация кода

BEG – генератор генераторов кода, основанный на динамическом программировании. Генерирует локальные и глобальные распределители регистров (***http://www.first.gmd.de/beg/***).

IBURG – программа, генерирующая быстрые синтаксические анализаторы для cost-augmented грамматик.

(***http://www.cs.princeton.edu/software/iburg/***)

MBURG – генерирует анализаторы деревьев снизу вверх. На выходе создает программный код ISO Modula-2.

(***http://compilers.iecc.com/comparch/article/95-08-036***)

TWIG – преобразует схему спецификации дерева, состоящую из правил шаблон-действие со связанными издержками в функции C, которые могут вызываться для манипуляции входными деревьями.

(***http://www.acm.org/pubs/citations/journals/toplas/1989-11-4/p491-aho/***)

7.6. Анализ и оптимизация

BANE (Berkeley ANalysis Engine) – пакет инструментальных средств для создания программ анализа. BANE основан на ограничениях. Это означает, что анализ формулируется как система ограничений, генерируемых из текста программы. Решая эти ограничения, BANE вычисляет необходимую информацию (***http://bane.cs.berkeley.edu/***).

OMEGA – пакет инструментальных средств, в состав которого входят структуры и алгоритмы для анализа и преобразования научных программ (***http://www.cs.umd.edu/projects/omega/index.html***).

OPTIMIX – генератор оптимизаторов. Может быть использован для анализа и оптимизации программ. Входной язык основан на DATALOG и перезаписи графов (graph rewriting).

(***http://i44www.info.uni-karlsruhe.de/~assmann/optimix.html***)

PAG – генератор анализаторов программ. Поддерживает реализацию статических анализаторов программ. PAG генерирует эффективные анализаторы потока данных из кратких спецификаций.

(***http://www.absint.com/pag/***)

TINY – исследовательская реализация интерактивного инструмента для преобразования программ. Tiny определяет отношения зависимых

данных и в интерактивном режиме выполняет множество операций преобразования (***http://www.cse.ogi.edu/Sparse/tiny.html***).

7.7. Генераторы среды разработки

CENTAUR – с помощъю Centaur можно разрабатывать инструменты, необходимые для создания среды программирования: редакторы структур, отладчики, интерпретаторы и различные трансляторы.

(***http://www.inria.fr/croap/centaur/centaur.html***)

SYNTHESIZER GENERATOR – инструмент для создания чувствительных к языку сред редактирования и интерфейсов. Входные правила определяют абстрактный синтаксис языка, контекстно-зависимые связи, форматы отображения, конкретный входной синтаксис и преобразования «источник-источник».

(***http://www.grammatech.com/products/sg/lse-products.html***)

7.8. Инфраструктура, компоненты, инструменты

ACK (Amsterdam Compiler Kit) – является интегрированным набором программ, разработанных для того, чтобы упростить задачу создания кросс-платформенных компиляторов и интерпретаторов. Для каждого компилируемого языка должна быть написана программа для перевода исходного кода в промежуточный код. (***http://www.cs.vu.nl/vakgroepen/cs/ack.html***)

AISEE – автоматически вычисляет настраиваемый план графов, определенных в GDL (язык описания графов). Затем этот план отображается и может быть распечатан или изучен в интерактивном режиме. AISEE был разработан для визуализации внутренних структур данных, обычно находящихся в компиляторах (***http://www.absint.com/aisee/***).

ANDF (Architecture-Neutral Distribution Format) – технология, облегчающая разработку и распространение переносимого ПО на различных

аппаратных и программных платформах. (***http://www.gr.opengroup.org/andf/***)

ARCHELON – набор инструментов, позволяющий быстро разработать компилятор ANSI C, ассемблер, сборщик или упаковщик для любого микроконтроллера или DSP приложения. (***http://www.archelon.com/retarg.html***)

BOEHM COLLECTOR – консервативный сборщик мусора Boehm-Demers-Weiser может быть использован как замена сборщика мусора для malloc в C или new в C++. Он также используется большим количеством реализаций языков программирования, использующих C, как промежуточный код (***http://www.hpl.hp.com/personal/Hans_Boehm/gc/***).

COSY – среда разработки компиляторов с акцентом на параллельном выполнении фаз компилятора (***http://www.ace.nl/products/cosy.htm***).

DMS – пакет DMS Reengineering позволяет выполнять анализ, трансляцию и/или рееиженеринг крупномасштабных программных систем, содержащих произвольные смешения языков. DMS может быть также использован для генерации проблемно-зависимого программного обеспечения (***http://www.semdesigns.com/Products/DMS/DMSToolkit.html***).

EDG (Edison Design Group) – предоставляет внешние интерфейсы к компиляторам для рынка OEM (***http://www.edg.com/***).

EEL (Executable Editing Library) – предоставляет абстракции, позволяющие создавать инструменты для анализа и модификации выполнимых файлов программ, не затрагивая конкретные наборы команд, форматы выполнимых файлов или последовательность удаления существующего кода и добавления внешнего (***http://www.cs.wisc.edu/~larus/eel.html***).

JFRONT (Jfront Rawjava) – является библиотекой C++ для синтаксического анализа исходного кода java (***http://www.jfront.com/rawjava/***).

LDL (Language Development Laboratory) – система поддерживающая разработку языка, создание интерпретаторов и генерирование наборов тестов (***http://www.informatik.uni-rostock.de/FB/Praktik/psuet/ldl/index.html***).

NULLSTONE – автоматизированный инструмент анализа производительности компилятора, использующий метод вопрос-ответ при создании тестов для измерения оптимизатора.

(***http://www.nullstone.com/iadex.htm***)

SAGE++ – предоставляет объектно-ориентированный набор средств для создания систем преобразования программ для языков Fortran 77, Fortran 90, C и C++ (***http://www.extreme.indiana.edu/sage/***).

SCORPION – является мета-средой, приспособленной к созданию сред разработки ПО. Scorpion использует IDL (Inteface Description Language), позволяющий описывать структуру графов, содержащих вспомогательные узлы (***ftp://ftp.cs.arizona.edu/scorpion/***).

SIC – основанный на Smalltalk интерактивный компилятор компиляторов, образовательный инструмент для визуализации методов компиляции.

(***http://inf2-www.informatik.unibw-muenchen.de/Research/Tools/SIC/***)

SPARK (Scanning, Parsing, and Rewriting Kit) – небольшая языковая среда, поддерживающая создание языковых процессоров в Python.

(***http://www.csr.uvic.ca/~aycock/python/***)

SUIF – инфраструктура, разработанная для поддержки совместных исследований оптимизации и параллелизации компиляторов. Независимо разработанные этапы компиляции работают совместно, используя для представления программ общий промежуточный формат.

(***http://suif.stanford.edu/suif/suif.html***)

TM – препроцессор, берущий шаблон кода и определения структур данных и генерирующий исходный код для произвольного языка программирования (***http://pds.twi.tudelft.nl/~reeuwijk/software/Tm/intro.html***).

TRIMARAN – инфраструктура для поддержки современных исследований проблем компиляции для архитектур ILP(Instruction Level Parallel). В настоящее время система ориентирована на архитектуру EPIC (Explicitly Parallel Instruction Computing), особенно для семейства процессоров HPL-PD. Система поддерживает исследование компиляции в машино-зависимых ILP методах трансляции, таких как эффективное использование предсказывания, уменьшение длины критического пути, планирование выполнения команд, распределение регистров и программное управление иерархией кэша (***http://www.trimaran.org/***).

ZEPHYR – философия Zephyr состоит в создании компилятора из частей. Части могут включать внешние интерфейсы, генераторы, оптимизаторы и клей, объединяющий все части вместе. Вы можете даже генерировать части автоматически из компактных спецификаций.

(***http://www.cs.virginia.edu/zephyr/***).

ЛИТЕРАТУРА

1. Ахо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. М.: Мир, 1978.

2. Льюис Ф., Розенкранц Д., Стринз Р. Теоретические основы проектирования компиляторов. М.: Мир, 1979.

3. Легалов А.И. Разработка трансляторов в учебном курсе. - http://www.nsu.ru/archive/conf/nit/96/sect7/node8.html.

4. Albert Graef. TP Lex and Yacc - The Compiler Writer's Tools for Turbo Pascal (Version 4.1, User Manual).- Department of Musicinformatics Johannes Gutenberg-University Mainz, April 1998.

5. John R. Levine, Tony Mason, Doug Brown. Lex & Yacc. – O'Reilly & Associates, 366 pages 2nd/updated edition, October 1992.

6. M. E. Lesk, E. Schmidt. LEXLexical Analyzer Generator, Unix Research System Programmer's Manual, Tenth Edition, Volume 2.

ПРИЛОЖЕНИЕ

Пример описания лексического анаизатора языка Pascal

```
%{
(* PASLEX.L: входной файл описания лексическия анализатор языка
Pascal*)
%}
%{
(*
* extensions: to ways to spell "external" and "->" ok for "^".
 *)
%}
%{
(* Note: Keywords are determined by scanning a keyword table, rather than in-
cluding the keyword patterns in the Lex source which is done in the original ver-
sion of this file. I prefer this method, because it makes the grammar itself more
readable (handling case-insensitive keywords in Lex is quite cumbersome, e.g.,
you will have to write something like [Aa][Nn][Dd] to match the keyword
`and'), and also produces a more (space-) efficient analyzer (184 states and 375
transitions for the keyword pattern version, against only 40 statesand 68 transi-
tions for the keyword table version).*)

procedure commenteof;
	begin
	writeln('unexpected EOF inside comment at line ', yylineno);
	end(*commenteof*);

function upper(str : String) : String;
	(* converts str to uppercase *)
	var i : integer;
```

```
        begin
        for i := 1 to length(str) do str[i] := upCase(str[i]); upper := str
        end(*upper*);

function is_keyword(id : string; var token : integer) : boolean;
        (* checks whether id is Pascal keyword; if so, returns corresponding
        token number in token *)
        const  id_len = 20;
        type    Ident = string[id_len];
        const
        (* table of Pascal keywords: *)
        no_of_keywords = 39;
        keyword : array [1..no_of_keywords] of Ident = (
        'AND',          'ARRAY',        'BEGIN',        'CASE',
        'CONST',        'DIV',          'DO',           'DOWNTO',
        'ELSE',         'END'           'EXTERNAL',     'EXTERN',
        'FILE',         'FOR',          'FORWARD',      'FUNCTION',
        'GOTO',         'IF',           'IN',           'LABEL',
        'MOD',          'NIL',          'NOT',          'OF',
        'OR',           'OTHERWISE',    'PACKED',       'PROCEDURE',
        'PROGRAM',      'RECORD',       'REPEAT',       'SET',
        'THEN',         'TO',           'TYPE',         'UNTIL',
        'VAR',          'WHILE',        'WITH');
        keyword_token : array [1..no_of_keywords] of integer = (
        _AND,           _ARRAY,         _BEGIN,         _CASE,
        _CONST,         _DIV,           _DO,            _DOWNTO,
        _ELSE,          _END,           _EXTERNAL,
_EXTERN                         (* EXTERNAL: 2 spellings (see above)! *)
        _FILE,          _FOR,           _FORWARD,       _FUNCTION,
```

```
        _GOTO,          _IF,            _IN,            _LABEL,
        _MOD,           _NIL,           _NOT,           _OF,
        _OR,            _OTHERWISE,     _PACKED,        _PROCEDURE,
        _PROGRAM,       _RECORD,        _REPEAT,        _SET,
        _THEN,          _TO,            _TYPE,          _UNTIL,
        _VAR,           _WHILE,         _WITH);
        var m, n, k : integer;
        begin
          id := upper(id);
           (* binary search: *)
          m := 1; n := no_of_keywords;
          while m<=n do
          begin
            k := m+(n-m) div 2;
            if id=keyword[k] then
              begin
                is_keyword := true;
                token := keyword_token[k];
                exit
              end
           else    if id>keyword[k] then
                     m := k+1
                   else
                     n := k-1
                   end;
              is_keyword := false
end(*is_keyword*);
%}
NQUOTE          [^']
```

```
%%
%{
var c  : char; kw : integer;
%}
[a-zA-Z]([a-zA-Z0-9])*    if is_keyword(yytext, kw) then
                                return(kw)
          else
                         return(IDENTIFIER);
":="                          return(ASSIGNMENT);
'({NQUOTE}|'')+'              return(CHARACTER_STRING);
":"                           return(COLON);
","                           return(COMMA);
[0-9]+                        return(DIGSEQ);
"."                           return(DOT);
".."                          return(DOTDOT);
"="                           return(EQUAL);
">="                          return(GE);
">"                           return(GT);
"["                           return(LBRAC);
"<="                          return(LE);
"("                           return(LPAREN);
"<"                           return(LT);
"-"                           return(MINUS);
"<>"                          return(NOTEQUAL);
"+"                           return(PLUS);
"]"                           return(RBRAC);
[0-9]+"."[0-9]+               return(REALNUMBER);
")"                           return(RPAREN);
";"                           return(SEMICOLON);
```

```
"/"                     return(SLASH);
"*"                     return(STAR);
"**"                     return(STARSTAR);
"->"                     |
"^"                      return(UPARROW);
"(*"                     |

"{"                     begin
                         repeat
                                   c := get_char;
                                  case c of
                                       '}' :    ;
                         '*' :    begin
                                                  c := get_char;
     if c=')' then
     exit
     else
     unget_char(c)
                                              end;
                                       #0 :    begin
                                                  commenteof;
     exit;
     end;
              end;
                    until false
          end;
[ \n\t\f]                ;
               .                    return(ILLEGAL);
```

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ………………………………………………………………………. 3

1 МОДЕЛИ ОПИСАНИЯ ГРАММАТИК ПРЕДМЕТНО-ОРИЕНТИРОВАННЫХ ЯЗЫКОВ ……………………………………………………………………….. 4

2 СТАДИИ РАБОТЫ ТРАСЛЯТОРА ………………………………………………. 20

3 ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ TP LEX……………………….. 32

4 ЯЗЫК ОПИСАНИЯ ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ LEX…………………… 38

5 МОДУЛЬ БИБЛИОТЕКИ LEXLIB…………………………………………….... 62

6 ГЕНЕРАТОР ТАБЛИЦ КЛЮЧЕВЫХ СЛОВ…………………………………….. 72

7 ОБЗОР ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ АВТОМАТИЗИРОВАННОЙ РАЗРАБОТКИ ТРАНСЛЯТОРОВ ………………………………………………… 76

ЛИТЕРАТУРА ………………………………………………………………… 88

ПРИЛОЖЕНИЕ ………………………………………………………………. 89

Электронное учебное издание

Александр Александрович **Рыбанов**
Ольга Викторовна **Свиридова**

ТЕОРИЯ ФОРМАЛЬНЫХ ЯЗЫКОВ И МЕТОДОВ ТРАНСЛЯЦИИ

Учебное пособие

Электронное издание сетевого распространения

Редактор Матвеева Н.И.

Темплан 2022 г. Поз. № 8.
Подписано к использованию 23.06.2022. Формат 60x84 1/16.
Гарнитура Times. Усл. печ. л. 5,9.

Волгоградский государственный технический университет.
400005, г. Волгоград, пр. Ленина, 28, корп. 1.

ВПИ (филиал) ВолгГТУ.
404121, г. Волжский, ул. Энгельса, 42а.