

КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ
ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ
И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Кафедра теоретической кибернетики

А.М. ГУСЕНКОВ, Н.А. ПРОКОПЬЕВ

СПЕЦИАЛИЗИРОВАННЫЕ ЯЗЫКИ
ОБРАБОТКИ ИНФОРМАЦИИ

Учебное пособие

Казань – 2024

УДК 004.43+811.93
ББК 32.973.2

*Принято на заседании кафедры теоретической кибернетики
Протокол № 8 от 29 апреля 2024 года*

*Принято на заседании учебно-методической комиссии института
вычислительной математики и информационных технологий
Протокол № 10 от 29 мая 2024 года*

Рецензенты:

кандидат физико-математических наук,
доцент кафедры теоретической кибернетики КФУ **Н.Р. Бухараев**;
кандидат физико-математических наук,
доцент кафедры теоретической кибернетики КФУ **Р.Б. Ахтямов**

Гусенков А.М., Прокопьев Н.А. Специализированные языки обработки информации: учебное пособие / А.М. Гусенков, Н.А. Прокопьев. – Казань: Казанский федеральный университет, 2024. –182 с.

В учебном пособии представлены генераторы лексических (lex) и синтаксических (yacc) анализаторов, а также основы теории конечных автоматов и формальных грамматик. Генератор lex по описанию лексем на языке регулярных выражений строит детерминированный конечный автомат. Генератор yacc по описанию входной грамматики языка строит МП-автомат, реализующий синтаксический анализ. В пособии рассмотрены генераторы для языков C и C++ (lex, yacc), Java (JFlex, CUP), Python (пакеты PLY и SLY). Представлены расширенные примеры реализации анализаторов с использованием указанных пакетов в реальном практическом применении.

Учебное пособие представляет собой основную часть лекционных специальных курсов «Специализированные языки обработки информации», «Языки программирования и методы трансляции», «Математическая лингвистика» и предназначено для использования в качестве учебного и справочного материала студентами математических специальностей.

© Гусенков А.М., Прокопьев Н.А., 2024

© Казанский федеральный университет, 2014

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
ГЛАВА 1. ЗАДАЧА РАЗБОРА ФОРМАЛЬНОГО ЯЗЫКА	9
1.1. Формальные языки	9
1.2. Регулярные языки	10
1.3. Автоматы	11
1.4. Формальные грамматики	12
1.5. Анализаторы контекстно-свободных языков	18
ГЛАВА 2. ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ LEX	21
2.1. Регулярные выражения	21
2.1.1. Первичные элементы	22
2.1.2. Унарные операции	23
2.1.3. Бинарные операции	23
2.1.4. Метасимволы	24
Примеры регулярных выражений	25
2.2. Структура входного файла	26
2.2.1. Секция определений	26
2.2.2. Секция правил	28
2.2.2.1. Встроенные переменные	29
2.2.2.2. Встроенные функции	30
2.2.2.3. Встроенные макрооперации	32
2.2.2.4. Альтернативные правила	32
2.2.2.5. Активные правила	35
2.2.3. Секция подпрограмм	36
2.2.4. Примеры входных файлов генератора lex	37
2.3. Использование генератора lex	39
ГЛАВА 3. ГЕНЕРАТОР СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ YACC	41
3.1. Структура входного файла	43
3.2. Секция определений	43
3.3. Секция подпрограмм	46

3.4.	Секция правил	46
3.4.1.	Правила	47
3.4.2.	Рекурсивные правила	49
3.4.3.	Действия	50
3.4.4.	Использование в действиях псевдопеременных	52
3.5.	Конфликтные ситуации при грамматическом разборе	56
3.6.	Структура информационного файла u.output	64
3.7.	Обработка ошибок при грамматическом разборе	67
3.8.	Стандартные функции	70
3.9.	Пример входного файла	71
ГЛАВА 4. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ГЕНЕРАТОРОВ YACC И LEX		74
4.1.	Использование генераторов flex и bison	75
4.2.	Работа с flex и bison на ОС Windows	77
4.3.	Пример грамматики языка SQL	78
4.4.	Формат обмена данными MapInfo MID/MIF	89
ГЛАВА 5. ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ JFLEX		107
5.1.	Регулярные выражения	107
5.2.	Структура входного файла	108
5.2.1.	Секция код пользователя	108
5.2.2.	Секция опции и определения	108
5.2.3.	Секция лексические правила	112
5.2.4.	Встроенные переменные и методы	114
5.3.	Примеры входных файлов	114
5.4.	Использование генератора JFlex	117
ГЛАВА 6. ГЕНЕРАТОР СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ CUP		118
6.1.	Структура входного файла	119
6.1.1.	Секция задания пакета и спецификаций import	120
6.1.2.	Секция код пользователя	120
6.1.3.	Секция список терминальных и нетерминальных символов	121
6.1.4.	Секция описания приоритетов	122

6.1.5. Секция грамматики	122
6.2. Пример входного файла	124
ГЛАВА 7. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ JFLEX И JAVA CUP	126
8.1. MiniJava – игрушечный язык	126
8.2. Калькулятор	133
8.3. Разбор C	136
ГЛАВА 8. ГЕНЕРАТОР АНАЛИЗАТОРОВ PLY	148
8.1. Установка и использование PLY	149
8.2. Настройка лексического анализатора lex.py	149
8.3. Состояния лексического анализатора lex.py	153
8.4. Запуск лексического анализатора lex.py	154
8.5. Пример лексического анализатора lex.py	155
8.6. Настройка синтаксического анализатора yacc.py	156
8.7. Запуск синтаксического анализатора yacc.py	158
8.8. Обработка конфликтов синтаксического анализатора yacc.py	159
8.9. Пример синтаксического анализатора yacc.py	159
ГЛАВА 9. ГЕНЕРАТОР АНАЛИЗАТОРОВ SLY	161
9.1. Установка и использование SLY	161
9.2. Настройка лексического анализатора Lexer	162
9.3. Запуск лексического анализатора Lexer	165
9.4. Пример лексического анализатора Lexer	166
9.5. Настройка синтаксического анализатора Parser	166
9.6. Запуск синтаксического анализатора Parser	169
9.7. Обработка конфликтов синтаксического анализатора Parser	170
9.8. Пример синтаксического анализатора Parser	170
9.9. Язык JSON	171
9.10. Язык STL	174
9.11. Язык f-string	178
СПИСОК ОСНОВНОЙ ЛИТЕРАТУРЫ	180
СПИСОК ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ	181

ВВЕДЕНИЕ

Значительная часть создаваемых программ, рассчитанных на определенную структуру входной информации, явно или неявно включает в себя некоторую процедуру синтаксического анализа. В задачах, где пользователю при задании входной информации предоставляется относительная свобода в отношении сочетания и последовательности структурных элементов, синтаксический анализ достаточно сложен. При решении подобных задач существенную поддержку могут оказать сервисные программы, осуществляющие построение синтаксических (грамматических) анализаторов на основе заданных сведений о синтаксической структуре входной информации.

Обширной областью, где наиболее явно видна необходимость нетривиального грамматического анализа, а, следовательно, и его автоматизации, является область создания компиляторов и интерпретаторов специализированных языков для решения задач преобразования информации, конвертирования баз данных, создания пакетных редакторов, описания сценариев поиска информации, моделирования, управления технологическими процессами и многих других.

В настоящее время существует большое количество программ автоматизации синтаксического анализа, так называемых “разборщиков” (parser). Такие программы имеются во всех современных операционных системах, а многие встроены непосредственно в языки программирования. Генерацию кода такие “разборщики” выполняют на различные языки программирования (С, С++, Java, XML и др.), но принцип их работы основан на одной и той же схеме.

В данной работе рассматривается классический “разборщик” – генераторы yacc и lex, которые были первыми средствами автоматизации построения анализаторов и появились в среде операционной системы UNIX. Генерация кода и все примеры рассматриваются на языке программирования С.

Дополнительную информацию можно получить в документации, кроме того обширная информация о генераторах yacc и lex имеется в Internet.

Исходная программа, написанная на некотором языке программирования, представляет собой последовательность символов. Компилятор превращает эту последовательность символов в последовательность битов – объектный код, а интерпретатор непосредственно выполняет команды. В этом процессе можно выделить две наиболее трудоемкие стадии программирования и отладки – лексический анализ и синтаксический анализ.

Работа лексического анализатора состоит в том, чтобы сгруппировать определенные терминальные символы в единые синтаксические объекты, называемые лексемами. Лексический анализатор – это транслятор, входом которого служит последовательность символов, представляющая исходную программу, а выходом – последовательность лексем. Этот выход образует вход синтаксического анализатора.

Синтаксический анализатор исследует последовательность лексем и устанавливает, удовлетворяет ли она структурным условиям, заданным правилами грамматики языка. Входные лексеммы синтаксического анализатора являются для него терминальными символами его грамматики.

Генератор lex строит лексический анализатор по описанию лексем на языке регулярных выражений. Результатом работы генератора lex является программа на языке Си, в которой построен детерминированный конечный автомат, реализующий функцию лексического анализа.

Генератор синтаксических анализаторов yacc по описанию входной грамматики языка строит конечный автомат с магазинной памятью в виде программы на языке С.

Генератор yacc обрабатывает широкий класс контекстно-свободных грамматик – LALR(1)-грамматики. LALR(1)-грамматики, являясь подмножеством LR(1)-грамматик, допускают при построении таблиц разбора сокращение

общего числа состояний за счет объединения идентичных состояний. Синтаксические анализаторы, создаваемые с помощью уасс, реализуют так называемый LALR(1)-разбор, являющийся модификацией одного из основных методов разбора "снизу вверх" – LR(k)-разбора (буквы L(ef) и R(igh) в обоих сокращениях означают соответственно чтение входных символов слева направо и использование правостороннего вывода. Индекс в скобках показывает число предварительно просматриваемых лексических единиц.

Любой разбор по принципу "снизу вверх" (или восходящий разбор) состоит в попытке приведения всей совокупности входных данных (входной цепочки) к так называемому "начальному символу грамматики".

Пользователь уасс должен описать структуру своей входной информации (грамматику) как набор правил. Грамматические правила описываются в терминах некоторых исходных конструкций, которые называются лексическими единицами, или лексемами. Имеется возможность задавать лексемы непосредственно (литерально) или употреблять в грамматических правилах имя лексем.

Генератор уасс обеспечивает автоматическое построение лишь процедуры грамматического анализа. Однако, действия по обработке входной информации обычно должны выполняться по мере распознавания на входе тех или иных допустимых грамматических конструкций. Поэтому наряду с заданием грамматики входных текстов уасс предусматривает возможность описания для отдельных конструкций семантических процедур (действий) с тем, чтобы они были включены в программу грамматического разбора.

ГЛАВА 1. ЗАДАЧА РАЗБОРА ФОРМАЛЬНОГО ЯЗЫКА

Задача разбора формального языка является одной из двух взаимосвязанных задач теории формальных языков и грамматик, лежащей в основании математической лингвистики. Задача заключается в том, чтобы для произвольной цепочки символов установить, является ли она правильной конструкцией для некоторого языка, в случае утвердительного ответа разделить цепочку на отдельные синтаксические элементы, в случае отрицательного – указать информацию об ошибках в данной цепочке.

Алгоритмы решения данной задачи находят применение в компиляторах и интерпретаторах при предварительном анализе исходного текста программы на предмет ошибок и при выполнении программного кода, в создании языков разметки, в автоматном программировании и в частичном анализе естественных языков.

В данной главе рассмотрены базовые определения, даны некоторые теоремы из теории автоматов, теории формальных языков и грамматик, относящиеся к данной задаче, в общем виде описаны алгоритмы разбора языка.

1.1. Формальные языки

Алфавит – это множество атомарных символов. Пример: $A_1 = \{ 'a', 'b', 'c', 'd' \}$, $A_2 = \{ '0', '1', '&' \}$, $A_3 = \{ \text{'формула'}, \text{'число'}, \text{'знак'} \}$.

Язык – это множество слов (цепочек символов) из некоторого алфавита. Если язык L содержит всевозможные слова из символов алфавита A , то он обозначается как $L = A^*$ при наличии пустого слова в языке, и $L = A^+$ если все слова в языке – непустые.

Основные способы задания языков: перечисление всех слов из языка, задание множества слов в виде порождающей функции, задание формального ав-

томата, распознающего язык, регулярные выражения, формальные грамматики, форма Бэкуса-Наура.

Для языков определены все стандартные операции над множествами (объединение, пересечение, дополнение), а также следующие:

1. L_1L_2 – конкатенация, содержит слова вида vw , где v – слово из L_1 , w – слово из L_2 ,
2. L_1/L_2 – правое отношение, содержит такие слова v , для которых существует такое слово w из L_2 , что vw принадлежит L_1 ,
3. L_1^* – замыкание Клини, содержит слова вида $w_1w_2w_3\dots w_n$, где w_i из L_1 , $n \geq 0$, включает в себя пустое слово ϵ ,
4. L_1^R – обращение, содержит обращенные слова из языка L_1 ,
5. L_1+L_2 – смешение языков, содержит слова вида $v_1w_1v_2w_2\dots v_nw_n$, где v_i из L_1 , w_i из L_2 , $n > 0$.

1.2. Регулярные языки

Регулярный язык – это множество слов, которое может быть выражено с помощью регулярных выражений. Подробно синтаксис регулярных выражений описан в главах, посвященных lex анализатору.

Регулярный язык в алфавите A определяется следующими рекурсивными свойствами:

1. Пустое множество, множество из пустого слова, множество слов, состоящих из одного символа алфавита A являются регулярными в этом алфавите.
2. Если язык является регулярным в алфавите A , то множество всевозможных конкатенаций его слов является регулярным в этом алфавите.
3. Если два языка являются регулярными в алфавите A , то их объединение является регулярным языком в этом алфавите, множество из всевозможных конкатенаций пар слов из этих языков является регулярным языком в этом алфавите.

4. Никакие другие множества слов не являются регулярными языками в алфавите A .

У регулярных языков есть свойство, полезное для доказательства нерегулярности некоторого языка – лемма о накачке.

Пусть L регулярный язык. Тогда существует целое $p \geq 1$ зависящее только от L , такое что любая строка w из L длины по меньшей мере p (p называется «длиной накачки») может быть записана как $w = xyz$, так что:

1. $|y| \geq 1$
2. $|xy| \leq p$
3. для всех $n \geq 0$, $xy^n z \in L$

Неформально, лемма говорит о таком факте, что если язык регулярный, то для любого слова из этого языка существует подстрока из этого слова, многократное повторение которой создает новое слово из этого же языка.

1.3. Автоматы

Формальный конечный автомат – это пятерка $A = (S, X, Y, \sigma, \lambda)$, где:

S – конечное множество состояний автомата,

X – конечный алфавит входных символов,

Y – конечный алфавит выходных символов,

$\sigma: S \times X \rightarrow S$ – функция переходов по состояниям,

$\lambda: S \times X \rightarrow Y$ – функция выходов.

В каждый момент времени автомат находится в одном из состояний из S . Из данного множества выделяется начальное состояние s_i , в котором автомат находится в начале работы. Автомат с выделенным начальным состоянием называется инициальным. Если при этом множество состояний S конечно, то автомат называется конечным.

На вход в автомат поступает последовательность символов, которую он обрабатывает по одному символу за шаг работы. При этом автомат переходит в

новое состояние в соответствии с функцией переходов и выводит некоторый символ в соответствии с функцией выходов. Если функция переходов определена однозначно (по одной паре состояние-символ возможен переход только в одно из состояний), то автомат называется детерминированным, иначе – недетерминированным.

Задача порождения языка для конечных автоматов подразумевает построение автомата, который при запуске выводит слова из некоторого языка.

Задача распознавания языка для конечных автоматов подразумевает построение автомата, который при получении на вход слов из некоторого языка переходит в завершающее состояние, а при получении слов не из этого языка – заикливается.

Автоматы являются эквивалентными, если порождаемые или распознаваемые ими языки совпадают. Теорема о детерминизации утверждает, что для любого конечного автомата может быть построен эквивалентный ему детерминированный автомат при возможном увеличении количества состояний, при этом количество состояний в худшем случае растет экспоненциально относительно роста количества состояний исходного автомата.

Язык называется автоматным, если существует конечный автомат, распознающий этот язык.

Замечание: любой конечный язык является автоматным языком.

Теорема Клини. Класс регулярных языков совпадает с классом автоматных языков.

1.4. Формальные грамматики

Грамматика – это четверка $G = (T, N, P, s)$, где:

T – алфавит терминальных символов,

N – алфавит нетерминальных символов (**T** и **N** не пересекаются),

P – набор правил перехода вида $\langle \text{левое слово} \rangle \rightarrow \langle \text{правое слово} \rangle$, где левое слово непустое и содержит хотя бы один нетерминал, правое слово – любое из терминалов и нетерминалов,

s – начальный символ грамматики из набора нетерминалов.

Вывод – это последовательность слов из терминалов и нетерминалов, начинающаяся со стартового символа грамматики, в которой каждое следующее слово получено из предыдущего путем применения одного из правил. Последнее слово вывода состоит полностью из терминалов и является словом из некоторого языка. Таким образом, множество всех выводов из грамматики задает некоторый язык.

Пример: Грамматика языка формул.

$$\begin{aligned}
 T &= \{ '0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '+', '-', '*', '/', '(', ')' \}, \\
 N &= \{ \text{'ФОРМУЛА'}, \text{'ЗНАК'}, \text{'ЧИСЛО'}, \text{'ЦИФРА'} \}, \\
 P &= \{ \\
 &\quad \text{'ФОРМУЛА'} \rightarrow \text{'ФОРМУЛА'} \text{'ЗНАК'} \text{'ФОРМУЛА'}, \\
 &\quad \text{'ФОРМУЛА'} \rightarrow \text{'ЧИСЛО'}, \\
 &\quad \text{'ФОРМУЛА'} \rightarrow (\text{'ФОРМУЛА'}), \\
 &\quad \text{'ЗНАК'} \rightarrow '+' \mid '-' \mid '*' \mid '/', \\
 &\quad \text{'ЧИСЛО'} \rightarrow \text{'ЦИФРА'}, \\
 &\quad \text{'ЧИСЛО'} \rightarrow \text{'ЧИСЛО'} \text{'ЦИФРА'}, \\
 &\quad \text{'ЦИФРА'} \rightarrow '0' \mid '1' \mid '2' \mid '3' \mid '4' \mid '5' \mid '6' \mid '7' \mid '8' \mid '9', \\
 &\} \\
 s &= \text{'ФОРМУЛА'}
 \end{aligned}$$

Знак | в описании правил означает «или» – варианты правых слов, выводимых из одного и того же левого слова.

Для формальных грамматик предлагается иерархия Хомского, основанная на различных ограничениях, накладываемых на правила перехода.

Тип 0: неограниченные. Соответствуют обычному определению грамматики без дополнительных ограничений.

Тип 1: контекстно-зависимые и неукорачивающие. В контекстно-зависимых все правила имеют вид $uAv \rightarrow uvv$, где A – символ из N , u, v – слова

из $(TUN)^*$, w – слово из $(TUN)^+$ для неукорачивающих КЗ-грамматик, w из $(TUN)^*$ для укорачивающих.

Тип 2: контекстно-свободные. Все правила имеют вид $A \rightarrow w$, где A – символ из N , w – слово из $(TUN)^+$ для неукорачивающих КС-грамматик, w из $(TUN)^*$ для укорачивающих.

Тип 3: регулярные. Все правила имеют вид $A \rightarrow Va$ или $A \rightarrow a$ для леволinéйных грамматик, $A \rightarrow aB$ или $A \rightarrow a$ для праволинейных. A, B – символы из N , w – символ из T .

Класс языков, порождаемый грамматикой типа N , полностью содержит класс, порождаемый грамматикой типа $N+1$.

Теорема: класс языков, задаваемых регулярными грамматиками эквивалентен классу автоматных языков.

Таким образом, класс регулярных языков, класс автоматных языков и класс языков, задаваемых регулярными грамматиками, совпадают.

Пример: Грамматика

$N = \{ 'S', 'A' \},$
 $T = \{ 'a', 'b', 'c' \},$
 $s = 'S',$
 $P = \{$
 $S \rightarrow aS$
 $S \rightarrow bA$
 $A \rightarrow e$
 $A \rightarrow cA$
 $\}$

Такая грамматика задает тот же язык, что описывается регулярным выражением a^+bc^* или распознается автоматом (рис. 1).

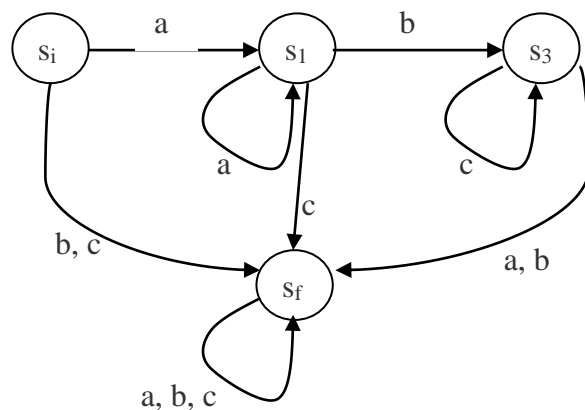


Рис. 1. Автомат распознавания регулярной грамматики

Применительно к синтаксическому анализу языков программирования и языков разметки наиболее широко используются контекстные грамматики, поэтому рассмотрим класс порождаемых ими языков подробнее.

Для класса контекстно-свободных языков существует лемма о накачке, позволяющая доказать непринадлежность некоторого языка к этому классу.

Пусть L – контекстно-свободный язык. Тогда существуют такие натуральные числа k и n , что если слово u принадлежит языку L и имеет длину не меньше n , то

1. слово u можно представить в виде $u = txvuw$, где длина цепочки xvu не превосходит k , и либо x , либо u – непустое слово;
2. для всякого натурального числа $m > 0$ цепочка tx^mvy^mw принадлежит языку L .

Неформально эта лемма говорит о том факте, что в любом слове из контекстно-свободного языка можно выделить 5 подстрок так, что многократное повторение 2 и 4 подстрок приводит к созданию новых слов, принадлежащих этому же языку.

Понятия о левом и правом выводе грамматики.

Если при построении вывода цепочки при каждом применении правила заменяется самый левый нетерминальный символ, то такой вывод называется

левым выводом. Если при построении вывода, всегда заменяется самый правый нетерминальный символ промежуточной цепочки, то вывод называется правым выводом.

Имеется взаимно однозначное соответствие между множеством всех левых выводов цепочек языка, множеством всех правых выводов и множеством всех деревьев выводов.

Множество всех выводов составляет абстрактное синтаксическое дерево грамматики.

КС-грамматика называется **однозначной** или **детерминированной**, если всякая выводимая терминальная цепочка имеет только одно дерево вывода (соответственно только один левый и только один правый вывод). Язык, заданный детерминированной грамматикой, называется детерминированным.

Пример: Варианты вывода из грамматики формул.

1. ФОРМУЛА →

ЧИСЛО →

ЧИСЛО ЦИФРА →

ЦИФРА ЦИФРА →

5 ЦИФРА →

59

2. ФОРМУЛА →

ФОРМУЛА ЗНАК ФОРМУЛА →

(ФОРМУЛА) ЗНАК ФОРМУЛА →

(ФОРМУЛА ЗНАК ФОРМУЛА) ЗНАК ФОРМУЛА →

(ЧИСЛО ЗНАК ФОРМУЛА) ЗНАК ФОРМУЛА →

(ЧИСЛО ЗНАК ЧИСЛО) ЗНАК ФОРМУЛА →

(ЧИСЛО ЗНАК ЧИСЛО) ЗНАК ЧИСЛО →

(ЦИФРА ЗНАК ЧИСЛО) ЗНАК ЧИСЛО →

(ЦИФРА ЗНАК ЧИСЛО ЦИФРА) ЗНАК ЧИСЛО →

(ЦИФРА ЗНАК ЦИФРА ЦИФРА) ЗНАК ЧИСЛО →

(ЦИФРА ЗНАК ЦИФРА ЦИФРА) ЗНАК ЦИФРА →

(4 ЗНАК ЦИФРА ЦИФРА) ЗНАК ЦИФРА →

(4 + ЦИФРА ЦИФРА) ЗНАК ЦИФРА →

(4 + 5 ЦИФРА) ЗНАК ЦИФРА →

(4 + 56) ЗНАК ЦИФРА →

$(4 + 56) * \text{ЦИФРА} \rightarrow$
 $(4 + 56) * 9$

Пример: КС-языки

1. Язык палиндромов $L = \{w \in T \mid w = wR\}$

$T =$ некоторый алфавит $\{\alpha \in T\}$

$N = \{S\}$

$S = S$

$P = \{$

$S \rightarrow \alpha S \alpha$

$S \rightarrow \alpha$

$S \rightarrow \epsilon$

$\}$

2. Язык арифметических выражений, грамматика которого была приведена ранее

3. Язык $L = \{a^n b^n \mid n \geq 0\}$

$T = \{a, b\}$

$N = \{S, A\}$

$P = \{$

$S \rightarrow aA$

$A \rightarrow Sb$

$S \rightarrow \epsilon$

$\}$

Пример: Не КС-языки

1. Язык: $L = \{a^n b^n c^n, n \geq 1\}$ – по лемме о накачке можно показать, что нельзя многократно повторять две подстроки из слов языка, не выходя за его пределы, так как необходимо одновременное повторение трех частей.

2. Язык: $L = \{a^n b^n c^k, n, k \geq 0, k \leq n\}$ – данному языку принадлежат слова из языка $\{a^n b^n c^n, n \geq 1\}$, а значит не для любого слова из языка возможно применение накачки.

3. Язык: $L = \{a^{n^2}\}$.

1.5. Анализаторы контекстно-свободных языков

Для разбора КС-языков есть два основных направления анализаторов: LL(k) и LR(k) анализаторы (k – количество предварительно просматриваемых лексем из входного потока).

LL(k) анализатор – нисходящий рекурсивный синтаксический анализатор. Анализирует поток символов слева направо и строит левый вывод для анализа потока. Промежуточные данные сохраняются в стек. Анализатор нисходящий, так как строит дерево разбора от корня синтаксического дерева к листьям. Грамматика, допускающая разбор таким анализатором без возврата потока назад называется LL(k) грамматикой.

Преимуществом LL(k) анализа является удобство написания программ для разбора языка вручную, без использования каких-либо генераторов. Существенным недостатком является достаточно узкий класс языков, допускающих такой разбор.

Примеры генераторов анализаторов: ANTLR, Coco/R, JavaCC. Пример языка, допускающего LL(k) разбор: язык арифметических формул. Пример языка, реализованного с таким типом разбора: Pascal.

LR(k) анализатор – восходящий рекурсивный синтаксический анализатор, анализирует поток символов слева направо и строит правый вывод для анализа потока. Анализатор восходящий, так как разбор начинается с листьев синтаксического дерева разбора и восходит к его корню. Грамматика, допускающая разбор таким анализатором, называется LR(k) грамматикой. Класс таких грамматик шире, чем класс LL(k) грамматик.

Недостатком является сложность разработки такого анализатора, поэтому на практике используются генераторы анализаторов.

Анализ потока состоит из процесса сдвига-свертки (shift-reduce):

1. Программа анализатора читает последовательно символы входной строки до тех пор, пока не накопится цепочка, совпадающая с правой частью какого-нибудь из правил. Рассмотренные символы переносим в стек (операция сдвиг).
2. Далее все символы совпадающей цепочки извлекаются из стека и на их место помещается нетерминал, находящийся в левой части этого правила (операция свертка).

Соответственно, при построении управляющей таблицы анализатора таких грамматик могут возникнуть конфликты сдвига-свертки (невозможность однозначно решить, считывать ли символы далее в стек или сворачивать цепочку по какому-либо правилу) или свертки-свертки (невозможность однозначно выбрать, по какому правилу свернуть цепочку). Конфликты, в сущности, означают, что построенная грамматика недетерминированная и ее желательно привести к детерминированному виду.

Подтипы LR(k) анализаторов.

Частный случай – **LR(0)** разбор редко используется на практике ввиду узкого класса разбираемых грамматик, однако является основой построения SLR(1) и LALR(1) анализаторов.

SLR(1) – разбирает более широкий класс грамматик, чем LR(0). В целом преимущество данного типа анализатора состоит в том, что в нем возможно автоматическое устранение некоторых конфликтов сдвига-свертки, появляющихся в LR(0). На практике используется редко. Например, выражения с унарными и бинарными операциями разбираются с помощью SLR(1) анализа.

LALR(1) – эффективное расширение для SLR(1), которое позволяет разбирать большинство LR(k) языков, также устраняет некоторые конфликты сдвига-свертки, проявляющиеся в SLR(1).

Пример разбора некоторой строки в грамматике языка формул приведен в таблице 1.

Таблица 1.

Пример разбора строки в грамматике языка формул

Операция	Стек
На вход приходит строка: '(4 + 56) * 9'	Инициализирован пустой стек []
Сдвиг '('	['(']
Сдвиг '4'	['(', '4']
Свертка по правилу ЦИФРА	['(', ЦИФРА]
Свертка по правилу ЧИСЛО	['(', ЧИСЛО]
Свертка по правилу ФОРМУЛА	['(', ФОРМУЛА]
Сдвиг '+'	['(', ФОРМУЛА, '+']
Свертка по правилу ЗНАК	['(', ФОРМУЛА, ЗНАК]
Сдвиг '5'	['(', ФОРМУЛА, ЗНАК, '5']
Свертка по правилу ЦИФРА	['(', ФОРМУЛА, ЗНАК, ЦИФРА]
Свертка по правилу ЧИСЛО	['(', ФОРМУЛА, ЗНАК, ЧИСЛО]
На этом этапе возможны и свертка по правилу ФОРМУЛА, и сдвиг, по умолчанию сдвиг имеет приоритет так как необходимо просмотреть символы далее.	
Сдвиг '6'	['(', ФОРМУЛА, ЗНАК, ЧИСЛО, '6']
Свертка по правилу ЦИФРА	['(', ФОРМУЛА, ЗНАК, ЧИСЛО, ЦИФРА]
Свертка по правилу ЧИСЛО	['(', ФОРМУЛА, ЗНАК, ЧИСЛО]
Свертка по правилу ФОРМУЛА	['(', ФОРМУЛА, ЗНАК, ФОРМУЛА]
Свертка по правилу ФОРМУЛА	['(', ФОРМУЛА]
Сдвиг ')'	['(', ФОРМУЛА, ')']
Свертка по правилу ФОРМУЛА	[ФОРМУЛА]
Сдвиг '*'	[ФОРМУЛА, '*']
Свертка по правилу ЗНАК	[ФОРМУЛА, ЗНАК]
Сдвиг '9'	[ФОРМУЛА, ЗНАК, '9']
Свертка по правилу ЦИФРА	[ФОРМУЛА, ЗНАК, ЦИФРА]
Свертка по правилу ЧИСЛО	[ФОРМУЛА, ЗНАК, ЧИСЛО]
Свертка по правилу ФОРМУЛА	[ФОРМУЛА, ЗНАК, ФОРМУЛА]
Свертка по правилу ФОРМУЛА	[ФОРМУЛА]
Финальный символ в стеке является стартовым нетерминалом грамматики, значит, введенное выражение принадлежит к языку, описываемому этой грамматикой.	

ГЛАВА 2. ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ LEX

Генератор lex строит лексический анализатор, задача которого – распознать из входного потока символов очередную лексему. Список лексем, который должен распознавать конкретный лексический анализатор, задается регулярными выражениями в секции правил входного файла генератора lex. По содержимому входного файла, генератор lex строит детерминированный конечный автомат в виде программы на языке C.

Генератор лексических анализаторов может применяться для построения различных преобразователей текстовой информации (конверторов), создания пакетных редакторов, реализации распознавателя директив в диалоговой программе и т.д.

Однако, наиболее важное применение лексического анализатора – это использование его в компиляторе или интерпретаторе специализированного языка. Здесь лексический анализатор выполняет функцию ввода и первичной обработки данных. Он распознает из входного потока лексемы и передает их синтаксическому анализатору (в качестве терминальных символов грамматического разбора).

Лексический анализатор может не только выделять лексемы, но и выполнять над ними 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, чтобы в дальнейшем обращаться к нему не по имени, а по адресу в таблице и т.д. Все такие преобразования записываются на языке программирования C.

2.1. Регулярные выражения

Регулярное выражение является шаблоном, определяющим некоторое множество последовательностей символов.

Регулярное выражение применяется как шаблон к последовательности символов текстового файла или ко входному потоку символов. Фрагмент текста считается соответствующим регулярному выражению, если он входит в множество последовательностей символов, определяемых этим выражением.

Синтаксически регулярное выражение – это слово, т.е. последовательность символов, не содержащая разделителей (пробелов, символов табуляции и т. д.). При построении регулярных выражений используются метасимволы (символы, имеющие внутри выражений специальный смысл).

Регулярное выражение строится из первичных элементов, которые могут соединяться знаками операций. Первичный элемент обозначает одиночный символ, описывая множество символов, которому он должен принадлежать. Операции над первичными элементами определяют последовательности символов и позволяют строить более сложные регулярные выражения.

2.1.1. Первичные элементы

C – где C – произвольный символ, не являющийся метасимволом; определяет множество, состоящее из одного символа C .

$[...]$ – определяет множество, состоящее из символов, перечисленных в квадратных скобках; внутри квадратных скобок допустима конструкция из трех соседних символов $X-Y$, где X и Y – произвольные символы; такая конструкция определяет множество, состоящее из символов, коды которых попадают в интервал между кодами символов X и Y (символы X и Y также включаются в это множество).

$[^...]$ – определяет множество, состоящее из символов, не перечисленных в квадратных скобках (отрицание множества, определенного предыдущим первичным).

- – определяет множество, состоящее из всех символов, кроме символа перевода строки.

2.1.2. Унарные операции

Пусть R_{exp} – произвольное регулярное выражение.

$R_{exp}?$ – определяет необязательное вхождение фрагмента, соответствующего регулярному выражению R_{exp} , т.е. повторение фрагмента 0 или 1 раз.

R_{exp}^* – определяет n -кратное ($n \geq 0$) последовательное повторение фрагмента, соответствующего регулярному выражению R_{exp} .

R_{exp}^+ – определяет n -кратное ($n > 0$) последовательное повторение фрагмента, соответствующего регулярному выражению R_{exp} .

$R_{exp}\{n1, n2\}$ – определяет n -кратное ($n1 \leq n \leq n2$) последовательное повторение фрагмента, соответствующего регулярному выражению R_{exp} .

$^R R_{exp}$ – определяет фрагмент, соответствующий регулярному выражению R_{exp} и стоящий в начале строки.

$R_{exp}\$$ – определяет фрагмент, соответствующий регулярному выражению R_{exp} и стоящий в конце строки.

(R_{exp}) – эквивалентно R_{exp} .

2.1.3. Бинарные операции

Пусть R_{exp1} и R_{exp2} – произвольное регулярное выражение.

$R_{exp1}|R_{exp2}$ – определяет фрагмент, соответствующий либо выражению R_{exp1} , либо выражению R_{exp2} .

Rexp1/Rexp2 – определяет фрагмент, соответствующий выражению Rexp1, если за ним следует фрагмент, соответствующий выражению Rexp2.

Rexp1Rexp2 – определяет последовательность фрагментов, из которых первый соответствует выражению Rexp1, а второй – выражению Rexp2.

Унарные операции старше бинарных. Бинарные операции имеют одинаковый приоритет и выполняются слева направо. Порядок выполнения операций может быть изменен с помощью круглых скобок.

2.1.4. Метасимволы

К метасимволам относятся следующие символы:

“ \ | / ^ \$? . * + - () [] { } ”

Любой метасимвол можно использовать как обычный символ, сняв с него специальный смысл посредством экранирования. Для экранирования используется символ \. Обратная наклонная черта снимает специальный смысл у непосредственно следующего за ней символа. Кроме того, в последовательности символов, заключенной в кавычки, все символы являются обычными и не несут специального смысла.

Для того чтобы использовать в регулярном выражении пробелы и символы табуляции, их нужно экранировать или заключать в кавычки. Не обязательно экранировать метасимволы и промежутки внутри квадратных скобок, поскольку экранирование там подразумевается. Символ минус имеет специальный смысл только внутри квадратных скобок.

В регулярных выражениях можно использовать неграфические символы. Произвольный неграфический символ записывается в виде \xxx, где xxx – восьмеричное представление кода символа.

Некоторые символы имеют следующее обозначение:

- `\n` – символ перевода строки;
- `\t` – горизонтальная табуляция;
- `\b` – шаг назад;
- `\\` – обратная наклонная черта.

Примеры регулярных выражений

`[a-z0-9_]`

– определяет одиночный символ, который может быть либо строчной буквой, либо цифрой, либо символом подчеркивания;

`abc\+`

– определяет множество, состоящее из одной последовательности символов – `abc+`;

`abc+`

– определяет бесконечное множество последовательностей, начинающихся с `ab` и далее содержащих любое количество символов `c`;

`[+-]?[0-9]+`

– определяет множество целых чисел, возможно начинающихся со знака;

`[+-]?0|[1-9][0-9]*`

– определяет множество целых чисел, возможно начинающихся со знака и не содержащих ведущих нулей;

`[a-zA-Z][a-zA-Z0-9]*`

– определяет множество идентификаторов, т.е. последовательностей символов, состоящих из букв и цифр и начинающихся с буквы;

`[a-c]+b/xу`

– определяет пустое множество, т.к. символ `b` всегда будет распознаваться регулярным выражением в квадратных скобках;

`^[01]{1,7}\/`

– определяет множество последовательностей из не более чем семи нулей и единиц, стоящих в начале строки, если за ней следует обратная наклонная черта.

2.2. Структура входного файла

Входной файл генератора lex имеет следующий формат:

секция определений

%%

секция правил

%%

секция подпрограмм

Секции определений и подпрограмм могут отсутствовать. Два подряд идущих символа %% являются разделителями секций и должны располагаться с первой позиции в отдельной строке. Минимальный входной файл для lex:

%%

Нет описаний, нет правил и нет подпрограмм. Будет сгенерирован анализатор, копирующий входную последовательность символов на выход без изменений.

2.2.1. Секция определений

Секция определений может содержать:

- список состояний лексического анализатора,
- определения имен регулярных выражений,
- фрагменты текста программы на языке C.

Список состояний лексического анализатора задается следующей директивой:

```
%start name1 name2 ...
```

где name1 name2 ... – имена состояний лексического анализатора, разделенные пробелами.

Директива %start должна начинаться с первой позиции строки. Имена состояний используются в секции правил и все они обязательно должны быть описаны в директиве %start.

Определения имен регулярных выражений имеют следующий формат:

```
Name Rexp
```

где Name – имя;

Rexp – регулярное выражение.

Имя Name должно начинаться с первой позиции строки. Имена регулярных выражений можно использовать в секции правил. В любом регулярном выражении секции правил имя регулярного выражения из секции определений будет заменяться на определяемое им регулярное выражение. Определение имен регулярных выражений обычно используют для сокращения записи регулярных выражений секции правил.

Фрагменты текста программы на языке C задаются двумя способами:

```
%{
```

```
строки
```

```
фрагмента
```

```
программы
```

```
%}
```

Директивы %{ и %} записываются в отдельных строках, начиная с первой позиции.

Другой способ заключается в записи текста фрагмента программы, начиная не с первой позиции строки.

Все такие фрагменты без изменений размещаются в начале программы, построенной генератором lex, и будут внешними для любой функции сгенерированной программы.

Обычно фрагменты текста программы содержат определения глобальных переменных, массивов, структур, внешних переменных и т.д., а также операторы препроцессора.

2.2.2. Секция правил

В секции правил описываются лексемы, которые должен распознавать лексический анализатор, и действия, выполняемые при распознавании каждой лексемы.

Общий формат правила:

`<States>Rexp Action`

где States – имена состояний лексического анализатора;

Rexp – регулярное выражение;

Action – действие.

Каждое правило должно располагаться в отдельной строке и начинаться с первой позиции этой строки. Конструкция `<States>` является необязательной, т. е. правило может начинаться с выражения Rexp. Имена состояний заключаются в угловые скобки и разделяются запятыми. Действие Action должно отделяться от выражения Rexp по крайней мере одним пробелом.

Выражение Rexp правила определяет лексему, которую должен распознавать лексический анализатор. Если заданы состояния States, то распознавание

лексемы по данному правилу производится только тогда, когда анализатор находится в одном из указанных состояний.

Действие Action задается в виде фрагмента программы на языке C и выполняется всякий раз, когда применяется данное правило, т.е. когда входная последовательность символов принадлежит множеству, определяемому регулярным выражением *Regex* данного правила.

Действие может состоять из одного оператора языка C или содержать блок операторов C, заключенный в фигурные скобки.

Кроме операторов языка C, в действии можно использовать встроенные переменные, функции и макрооперации анализатора *lex*.

2.2.2.1. Встроенные переменные

ytext[]

– одномерный массив (последовательность символов), содержащий фрагмент входного текста, удовлетворяющего регулярному выражению и распознанного данным правилом;

yyleng

– целая переменная, значение которой равно количеству символов, помещенных в массив *ytext*.

Встроенные переменные позволяют определить конкретную последовательность символов, распознанных данным правилом. При применении правила анализатор *lex* автоматически заполняет значениями встроенные переменные. Эти значения можно использовать в действии примененного правила.

Пример правила:

```
[a-z]+ printf("%s",ytext);
```

Регулярное выражение правила определяет бесконечное множество последовательностей символов, состоящих из букв латинского алфавита. Данное правило применяется, когда из входного потока символов поступает конкретная последовательность символов, удовлетворяющих его регулярному выражению. Оператор языка C *printf* выводит в выходной поток эту последовательность символов.

2.2.2.2. Встроенные функции

yymore()

– В обычной ситуации содержимое **ytext** обновляется всякий раз, когда производится применение некоторого правила. Иногда возникает необходимость добавить к текущему содержимому **ytext** цепочку символов, распознанных следующим правилом. **yymore()** вызывает переход анализатора к применению следующего правила. Входная последовательность символов, распознанная следующим правилом, будет добавлена в массив **ytext**, а значение переменной **yylen** будет равно суммарному количеству символов, распознанными этими правилами.

yyless(n)

– Оставляет в массиве **ytext** первые *n* символов, а остальные возвращает во входной поток. Переменная **yylen** принимает значение *n*. Лексический анализатор будет читать возвращенные символы для распознавания следующей лексемы. Использование **yyless(n)** позволяет посмотреть правый контекст.

input()

– Выбирает из входного потока очередной символ и возвращает его в качестве своего значения. Возвращает ноль при обнаружении конца входного потока.

output(c)

– Записывает символ *c* в выходной поток.

unput(c)

– Помещает символ *c* во входной поток.

ywrap()

– Автоматически вызывается при обнаружении конца входного потока. Если возвращает значение 1, то лексический анализатор завершает свою работу, если 0 – входной поток продолжается текстом нового файла. По умолчанию `ywrap` возвращает 1. Если имеется необходимость продолжить ввод данных из другого источника, пользователь должен написать свою версию функции `ywrap()`, которая организует новый входной поток и возвратит 0.

Пример: Входной файл.

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

Входной файл генератора `lex` содержит одно правило. Анализатор распознает строки символов, заключенные 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\"эюя"`. Сначала будет распознана цепочка `"абв\"` и, так как последним символом в этой цепочке будет символ `"\"`, выполнится вызов `yymore()`. В результате повторного применения

правила к цепочке "абв\ будет добавлено "эюя, и в **ytext** мы получим: "абв\ "эюя, что и требовалось.

2.2.2.3. Встроенные макрооперации

ECHO

– Эквивалентно `printf(“%s”,ytext);` . Печать в выходной поток содержимого массива **ytext**.

BEGIN st

– Перевод анализатора в состояние с именем *st*.

BEGIN 0

– Перевод анализатора в начальное состояние.

REJECT

– Переход к следующему альтернативному правилу. Последовательность символов, распознанная данным правилом, возвращается во входной поток, затем производится применение альтернативного правила.

2.2.2.4. Альтернативные правила

Регулярное выражение, входящее в правило, определяет множество последовательностей символов.

Два правила считаются альтернативными, если определяемые ими два множества последовательностей символов имеют непустое пересечение, либо существуют такие две последовательности из этих множеств, начальные части которых совпадают.

Примеры: Альтернативные правила.

1. Регулярное выражение

SWITCH

определяет единственную последовательность символов SWITCH, а регулярное выражение

[A-Z]⁺

определяет бесконечное множество последовательностей символов, в том числе и SWITCH.

2. Регулярное выражение

INT

определяют последовательность INT, которая является подпоследовательностью последовательности INTEGER, определяемой регулярным выражением

INTEGER

3. Регулярное выражение

AC⁺

определяет множество, являющееся пересечением множеств, определяемых выражениями

A[BC]⁺

A[CD]⁺

В лексическом анализаторе каждый входной символ учитывается один раз. Поэтому в ситуации, когда возможно применение нескольких правил, действует следующая стратегия:

1. Выбирается правило, определяющее самую длинную последовательность входных символов;
2. Если таких правил оказывается несколько, выбирается то из них, которое текстуально стоит раньше других.

Если требуется несколько раз обработать один и тот же фрагмент входной цепочки символов, то можно воспользоваться функцией `uiless` или макрооператором `REJECT`.

Примеры.

1. Предположим, что мы хотим подсчитать все вхождения цепочек **she** и **he** во входном тексте. Для этого мы могли бы написать следующий входной файл для lex:

```
%{
int s=0,h=0;
}%
%%
she      { s++;
           yless(1);
        }
he       h++;
```

Так как **she** включает в себя **he**, анализатор (без использования функции `yless(1)`) не распознает те вхождения **he**, которые включены в **she**, так как, прочитав один раз **she**, эти символы он не вернет во входной поток.

2. Рассмотрим следующий входной файл для lex:

```
%%
A[BC]+   { /* операторы
           обработки
           */
           REJECT;
        }
A[CD]+   { /* операторы
           обработки
           */
        }
```

Входная последовательность символов ACCB сначала будет распознана первым правилом, а затем первые три символа этой последовательности ACC будут распознаны вторым правилом.

2.2.2.5. Активные правила

Секция правил может содержать активные и неактивные правила. В распознавании входной последовательности символов принимают участие только активные правила. Все правила, в которых не указаны состояния (<States>) всегда являются активными. Правило, в котором заданы состояния, активно только тогда, когда анализатор находится в одном из перечисленных состояний.

Все имена состояний лексического анализатора, используемые в правилах, обязательно должны быть описаны директивой %start в секции определений.

Для перевода анализатора в некоторое состояние используется макрооперация BEGIN. Анализатор всегда находится только в одном состоянии. В начальном состоянии активны только правила, в которых отсутствуют состояния.

Действия в правилах Lex-программы выполняются, если правило активно, и если автомат распознает цепочку символов из входного потока как соответствующую регулярному выражению данного правила.

Любая последовательность входных символов, не соответствующая ни одному правилу, копируется в выходной поток без изменений. Можно сказать, что действие – это то, что делается вместо копирования входного потока символов на выход. Часто бывает необходимо не копировать на выход некоторую цепочку символов, которая удовлетворяет некоторому регулярному выражению. Для этой цели используется пустой оператор C, например:

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

Это правило игнорирует (запрещает) вывод пробелов, табуляций и символа перевода строки. Запрет выражается в том, что на указанные символы во входном потоке осуществляется действие ";" – пустой оператор языка C, и эти символы не копируются в выводной поток символов.

Пример.

```
%start COMMENT
COMM_BEGIN      "/"*
COM_END         "*/"
%%
{COM_BEGIN}    { ECHO;
                BEGIN COMMENT;
              }
<COMMENT>.      ECHO;
<COMMENT>\n     ECHO;
<COMMENT>{COM_END} { ECHO;
                    BEGIN 0;
                  }
.               ;
\n             ;
```

lex построит лексический анализатор, который выделяет комментарии в программе на языке C и записывает их в стандартный файл вывода. В случае распознавания начала комментария (`/*`) анализатор переходит в состояние COMMENT. При распознавании конца комментария (`*/`) анализатор переводится в начальное состояние.

Два последних правила позволяют игнорировать символы, не входящие в состав комментариев.

2.2.3. Секция подпрограмм

В секции подпрограмм размещаются функции, написанные на языке C, которые необходимы в конкретном лексическом анализаторе. Эти функции могут вызываться в действиях правил и, как обычно, передавать и возвращать значения аргументов.

Здесь же можно переопределить стандартные и встроенные функции лексического анализатора, дав им свою интерпретацию. Пользовательские версии этих функций должны быть согласованы между собой по выполняемым действиям и возвращаемым значениям.

Содержимое этой секции без изменений копируется в выходной файл, построенный генератором lex.

2.2.4. Примеры входных файлов генератора lex

Пример 1.

```
%%
[jJ][aA][nN][uU][aA][rR][yY]      printf("Январь");
[fF][eE][bB][rR][uU][aA][rR][yY]  printf("Февраль");
[mM][aA][rR][cC][hH]                printf("Март");
[aA][pP][rR][iI][lL]                printf("Апрель");
[mM][aA][yY]                          printf("Май");
[jJ][uU][nN][eE]                      printf("Июнь");
[jJ][uU][lL][yY]                      printf("Июль");
[aA][uU][gG][uU][sS][tT]             printf("Август");
[sS][eE][pP][tT][eE][mM][bB][eE][rR] printf("Сентябрь");
[oO][cC][tT][oO][bB][eE][rR]         printf("Октябрь");
[nN][oO][vV][eE][mM][bB][eE][rR]    printf("Ноябрь");
[dD][eE][cC][eE][mM][bB][eE][rR]    printf("Декабрь");
```

Генератор построит конечный автомат, который распознает английские наименования месяцев и выводит русские значения найденных английских слов. Все другие последовательности входных символов без изменений копируются в выходной поток.

Пример 2.

```
%start AA BB CC
%{
    /*
    *   Строится лексический анализатор,
    *   который распознает наличие
    *   включений файлов в Си-программе,
    *   условных компиляций,
    *   макроопределений,
    *   меток и головной функции main.
    *   Анализатор ничего не выводит, пока
```

```

*    осуществляется чтение входного
*    потока, а по его завершении
*    выводит статистику.
*/
%}
БУКВА          [A-ZA-Яa-za-я_]
ЦИФРА          [0-9]
ИДЕНТИФИКАТОР  {БУКВА}({БУКВА}|{ЦИФРА})*

int a1,a2,a3,b1,b2,c;
a1 = a2 = a3 = b1 = b2 = c = 0;
%%
^#                BEGIN AA;
^[ \t]*main       BEGIN BB;
^[ \t]*{ИДЕНТИФИКАТОР} BEGIN CC;
[ \t]+           ;
\n               BEGIN 0;
<AA>define        { a1++; }
<AA>include       { a2++; }
<AA>ifdef         { a3++; }
<BB>”(“.”)”      { b1++; }
<BB>”()”         { b2++; }
<CC>”:”         { c++; }
.                ;
%%
yywrap(){
    if( b1 == 0 && b2 == 0 )
        printf("В программе отсутствует функция main.\n");
    if( b1 >= 1 && b2 >= 1 ){
        printf("Многократное определение функции main.\n");
    } else {
        if(b1 == 1 )
            printf("Функция main с аргументами.\n");
        if( b2 == 1 )
            printf("Функция main без аргументов.\n");
    }
    printf("Включений файлов: %d.\n",a2);
    printf("Условных компиляций: %d.\n",a3);
    printf("Определений: %d.\n",a1);
    printf("Меток: %d.\n",c);
    return(1);
}

```

Оператор **return(1)** в функции **ywrap** указывает, что лексический анализатор должен завершить работу.

Пример 3.

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
^\n          ;
[ \t]*      ;
[A-Za-z]    { yylval = yytext[yytextlen-1] - 'a';
              return(LETTER);
            }
[0-9] +     { int i;
              yylval = yytext[0] - '0';
              for(i=1; i< yytextlen; i++)
                yylval = yylval *10 + yytext[i] - '0';
              return(DIGIT);
            }
```

Лексический анализатор распознает однобуквенные идентификаторы и целые положительные числа и возвращает номера типов этих лексем, определенных в файле **y.tab.h**. Во внешнюю переменную **yylval** помещаются следующие значения: для идентификатора – порядковый номер в английском алфавите; для числа – значение в двоичном представлении.

2.3. Использование генератора lex

Вызов выполнения генератора lex имеет вид:

lex Lfile

где **Lfile** – имя входного файла, построенного в соответствии с требованиями структуры входного файла lex.

В результате выполнения этой команды `lex` создаст лексический анализатор в виде текста программы на языке C и поместит его в файл со стандартным именем **lex.yy.c**

Файл **lex.yy.c** содержит две основных функции и несколько вспомогательных. Основными являются две следующие функции:

yylex()

– содержит разделы действий всех правил, которые определены пользователем;

yylook()

– реализует детерминированный конечный автомат, который осуществляет разбор входного потока символов в соответствии с регулярными выражениями правил входного файла генератора `lex`.

Для получения выполняемой программы лексического анализатора (загрузочного модуля) необходимо выполнить компиляцию программы **lex.yy.c** и скомпоновать ее с программами из стандартной библиотеки генератора `lex`. Это выполняется следующей командой:

cc lex.yy.c -ll

В результате выполнения этой команды будет создан выполняемый файл (загрузочный модуль) со стандартным именем **a.out**.

Головная функция **main** из стандартной библиотеки генератора `lex` имеет вид:

```
main(){
    yylex();
    exit(0);
}
```

Разработчик лексического анализатора имеет возможность подключить собственную функцию **main()** вместо библиотечной. Для этого достаточно поместить текст собственной функции **main()** в секцию подпрограмм входного файла `lex`. Наличие сгенерированного текста программы в файле **lex.yy.c** дает программисту дополнительные возможности для внесения корректив в работу лексического анализатора.

ГЛАВА 3. ГЕНЕРАТОР СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ УАСС

Генератор синтаксических анализаторов уасс по описанию входной грамматики языка строит конечный автомат с магазинной памятью в виде программы на языке С.

Генератор уасс обрабатывает широкий класс контекстно-свободных грамматик – LALR(1)-грамматики. LALR(1)-грамматики, являясь подмножеством LR(1)-грамматик, допускают при построении таблиц разбора сокращение общего числа состояний за счет объединения идентичных состояний, различающихся только набором символов-следователей (символов, которые могут следовать после применения одного из правил вывода, если разбор по этому правилу проходил через данное состояние). Другие грамматики являются неоднозначными для принятого в уасс метода разбора и вызовут конфликты.

Синтаксические анализаторы, создаваемые с помощью уасс, реализуют так называемый LALR(1)-разбор, являющийся модификацией одного из основных методов разбора "снизу вверх" – LR(k)-разбора (буквы L(eft) и R(ight) в обоих сокращениях означают соответственно чтение входных символов слева направо и использование правостороннего вывода. Индекс в скобках показывает число предварительно просматриваемых лексических единиц).

Любой разбор по принципу "снизу вверх" (или восходящий разбор) состоит в попытке приведения всей совокупности входных данных (входной цепочки) к так называемому "начальному символу грамматики".

В каждый момент грамматического разбора анализатор находится в некотором состоянии, определяемом предысторией разбора, и в зависимости от очередной лексемы предпринимает то или иное действие для перехода к новому состоянию. Различают два типа действий: **сдвиг**, т.е. чтение следующей входной лексемы, и **свертку**, т.е. применение одного из правил подстановки для замещения нетерминалом последовательности символов, соответствующей правой части правила. Работа уасс по генерации процедуры грамматического ана-

лиза заключается в построении таблиц, которые для каждого из состояний определяют тип действий анализатора и номер следующего состояния в соответствии с каждой из входных лексем.

Пользователь **уасс** должен описать структуру своей входной информации (**грамматику**) как набор **правил**. Грамматические правила описываются в терминах некоторых исходных конструкций, которые называются лексическими единицами, или **лексемами**. Имеется возможность задавать лексеммы непосредственно (литерально) или употреблять в грамматических правилах имя лексеммы.

Лексеммой называется цепочка (последовательность) символов, которую удобно рассматривать как единый синтаксический объект.

Набор лексем определяется разработчиком лексического анализатора. Для синтаксического анализатора все лексеммы считаются терминальными символами грамматики.

Генератор **уасс** обеспечивает автоматическое построение лишь процедуры грамматического анализа. Однако, действия по обработке входной информации обычно должны выполняться по мере распознавания на входе тех или иных допустимых грамматических конструкций. Поэтому наряду с заданием грамматики входных текстов **уасс** предусматривает возможность описания для отдельных конструкций семантических процедур (**действий**) с тем, чтобы они были включены в программу грамматического разбора. В зависимости от характера пользовательских семантических процедур (интерпретация распознанного фрагмента входного текста, генерация фрагмента объектного кода, отметка в справочной таблице или форматирование вершины в дереве разбора) генерируемая с помощью **уасс** программа будет обеспечивать кроме анализа тот или иной вид обработки входного текста, в частности, его компиляцию или интерпретацию.

3.1. Структура входного файла

Входной файл генератора уасс имеет следующий формат:

```
секция определений
%%
секция правил
%%
секция подпрограмм
```

Секции определений и подпрограмм могут отсутствовать. Два подряд идущих символа %% являются разделителями секций и должны располагаться с первой позиции в отдельной строке. Минимальный входной файл для уасс:

```
%%
правило
```

Пробелы, символы табуляции и перевода строки игнорируются, недопустимо лишь появление их в именах. Комментарий, ограниченный символами "/*" в начале и "*/" в конце, может находиться между любыми двумя разделителями в любой секции входного файла.

3.2. Секция определений

Секция определений может содержать:

- список терминальных символов грамматики (лексем),
- определение начального символа (аксиомы) грамматики,
- определение приоритетов и порядка выполнения операций,
- фрагменты текста программы на языке С.

Все строки секции определений, начинающиеся с символа %, интерпретируются как директивы уасс.

Терминальные символы грамматики (лексемы) определяются директивой:

```
%token name number ...
```

где

name – имя терминального символа (лексемы);

number – целое не отрицательное число, являющееся номером типа лексемы `name`. `number` является необязательным и может быть опущен. Многоточие указывает, что конструкция `name number` может повторяться.

Часто, для улучшения читабельности входного файла, каждую лексему определяют отдельной директивой `%token`.

Все номера типов лексем должны быть уникальными. По умолчанию (когда отсутствует `number`), yacc присваивает всем лексемам номера типов следующим образом:

- для одиночного символа (литерала) номером типа лексемы считается числовое значение кода символа из кодировочной таблицы;
- специальная лексема `error`, зарезервированная для обработки ошибок, получает следующий по порядку номер после кодировочной таблицы;
- лексемы, обозначенные именами, в соответствии с очередностью их объявления в директивах `%token` получают последующие номера.

Для каждого имени лексемы yacc генерирует оператор препроцессора:

```
#define <имя_лексемы> <номер_типа>
```

По этой причине имена лексем не должны совпадать с ключевыми словами языка C.

Аксиома грамматики определяется директивой:

```
%start name
```

где **name** – имя нетерминального символа

В случае отсутствия директивы `%start`, аксиомой грамматики считается нетерминал, указанный в левой части первого правила из секции правил.

Приоритеты и порядка выполнения операций задаются директивами:

```
%left name ...
```

```
%right name ...
```

`%nonassoc name ...`

где **name ...** – имена лексем, разделенные пробелами.

Директива `%left` задает левую ассоциативность (операции выполняются слева направо). Директива `%right` задает правую ассоциативность (операции выполняются справа налево). Директива `%nonassoc` определяет неассоциативные операции (операции, которые могут встречаться в выражении только один раз). Все лексем, указанные в одной директиве, имеют одинаковые приоритет и ассоциативность. Последовательность директив задает операции в порядке возрастания приоритета.

Пример: Задание приоритетов лексем.

```
%token OR AND NOT
%right '='
%left OR
%left AND
%left NOT
%left '+' '-'
%left '*' '/'
/* самый низкий приоритет имеет лексема "=",
   самый высокий - лексемы "*" и "/"
*/
```

Фрагменты текста программы на языке C размещаются следующим образом:

```
{
    строки
    фрагмента
    программы
}
```

Директивы `{` и `}` записываются в отдельных строках, начиная с первой позиции.

Все такие фрагменты без изменений размещаются в начале программы, построенной генератором уасс, и будут внешними для любой функции сгенерированной программы.

Обычно фрагменты текста программы содержат определения глобальных переменных, массивов, структур, внешних переменных и.т.д., а также операторы препроцессора.

3.3. Секция подпрограмм

В секции подпрограмм размещаются функции, написанные на языке С, которые необходимы в конкретном синтаксическом анализаторе. Эти функции могут вызываться в действиях правил и, как обычно, передавать и возвращать значения аргументов.

Здесь же можно переопределить стандартные функции синтаксического анализатора, дав им свою интерпретацию. Пользовательские версии этих функций должны быть согласованы между собой по выполняемым действиям и возвращаемым значениям.

Содержимое этой секции без изменений копируется в выходной файл, построенный генератором уасс.

3.4. Секция правил

В данной секции с помощью набора грамматических правил определяются все конструкции языка, которые должен разбирать синтаксический анализатор, и действия, выполняемые при грамматическом разборе.

3.4.1. Правила

Правила, определяющие синтаксический вид конструкции, задаются следующим образом:

L: R;

где **L** – левая часть правила – имя нетерминального символа;

R – правая часть правила – последовательность терминальных и нетерминальных символов (допустима и пустая последовательность).

Терминальными символами считаются все имена, определенные директивами %token секции определений (лексемы) и одиночные символы, заключенные в апострофы (литералы). Последовательность элементов правой части правила (**R**) может разделяться пробелами.

При грамматическом разборе последовательность (**R**) в результате применения правила заменяется нетерминальным символом (**L**).

Все нетерминальные символы должны быть определены, т.е. имя каждого из них должно появиться в левой части хотя бы одного правила.

Правила с общей левой частью можно задавать в сокращенной записи, без повторения левой части, используя для разделения альтернативных определений символ "|". Например, три следующих правила

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

можно записать в следующем эквивалентном виде:

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

Примеры.

Во всех последующих примерах имена терминальных символов будем давать прописными буквами, а имена нетерминальных символов – строчными.

1. Правило

```
assign: VAR '=' expr
```

может определять нетерминал **assign** (присваивание) как цепочку из трех символов: нетерминала **var** (переменная), терминального символа '=' и нетерминала **expr** (выражение).

2. Правила

```
ident:  LETTER  |  
       ident  LETTER  |  
       ident  DIGIT
```

рекурсивно описывают нетерминал **ident** (идентификатор) как последовательность букв (**LETTER**) и цифр (**DIGIT**) начинающихся с буквы.

3. Правила

```
number: sign number;  
sign:   '+' |  
       '-' ;
```

определяют число (**number**), которое может быть записано со знаком плюс или минус.

Сочетание пустого правила с другими правилами, определяющими тот же нетерминальный символ, является одним из способов указать на необязательность вхождения соответствующей конструкции.

3.4.2. Рекурсивные правила

Правила часто описывают некоторую конструкцию рекурсивно, т.е. правая часть может рекурсивно включать определяемый нетерминальный символ.

Различают леворекурсивные правила вида:

```
<имя_нетерминала> : <имя_нетерминала>  
<многократно_повторяемый_фрагмент>;
```

и праворекурсивные вида:

```
<имя_нетерминала> :  
<многократно_повторяемый_фрагмент>  
<имя_нетерминала>;
```

Пара правил:

```
list: ITEM |  
      list ITEM
```

используется для определения списковых конструкций (последовательностей элементов **ИТЕМ**). Если элементы списка должны, например, разделяться символом запятая, то правила могут выглядеть так:

```
list: ITEM |  
      list ‘,’ ITEM
```

Заметим, что в каждом из этих случаев первое правило (не содержащее рекурсии) будет применено только для первого элемента списка, а второе (рекурсивное) – для всех последующих. Эти правила записаны с левой рекурсией.

Допускаются оба вида рекурсивных правил, однако при использовании правил с правой рекурсией увеличивается объем анализатора и его рабочей памяти.

Нетерминальные символы, связанные с последовательностями или списками разнородных элементов, могут описываться произвольным числом рекурсивных и не рекурсивных правил. Например, группа правил

```
идентификатор: буква |  
                '$' |  
                идентификатор буква |  
                идентификатор цифра |  
                идентификатор '_' ;
```

описывает идентификатор как последовательность букв, цифр и символов "_", начинающуюся с буквы или символа "\$". Следует обратить внимание на то, что рекурсивные правила не имеют смысла, если для определяемого ими нетерминала не задано ни одного правила без рекурсии.

Для обеспечения возможности задания пустых списков или последовательностей в качестве не рекурсивного правила используется пустое:

```
list :          |  
        list ITEM ;
```

Сочетание пустых и рекурсивных правил является удобным способом представления грамматик и ведет к большей их общности, однако, некорректное использование пустых правил может вызывать конфликтные ситуации из-за неоднозначности выбора нетерминала, соответствующего пустой последовательности.

3.4.3. Действия

При необходимости с любым правилом можно связать действие – набор операторов языка Си, которые будут выполняться при каждом распознавании конструкции во входном тексте.

Действие не является обязательным элементом правил. Правило описывает синтаксическую структуру фрагмента входного текста, а действие – семантику этого фрагмента.

Действие заключается в фигурные скобки и помещается вслед за правой частью правила, т.е. правило с действием имеет вид:

`<имя_нетерминального_символа>: правая часть {действие};`

Точка с запятой после правила с действием может опускаться.

При использовании сокращенной записи правил с общей левой частью следует иметь в виду, что действие может относиться только к отдельному правилу, а не к их совокупности. Следующий пример иллюстрирует задание действий в случае правил с общей левой частью.

```
statement: assign_stat |
           if_then_stat {printf("if_оператор\n");} |
           goto_stat { kgoto++;
                     printf("goto_оператор\n");
                     }
```

Существует возможность задания действий, которые будут выполняться по мере распознавания отдельных фрагментов правила. Действие в этом случае можно размещать после любого элемента правой части правила.

Например, в правиле

```
if_then_stat: if '(' expression ')' {действие1}
              then statement ';' {действие 2}
```

действия заданы с таким расчетом, чтобы при разборе строки вида

if (a>b) then x=a;

действие 1 выполнялось при нахождении правой круглой скобки, а действие 2 – по окончании разбора.

Пример входного файла генератора уасс для построения синтаксического анализатора, определяющего соответствие открывающих и закрывающих скобок.

```
%token SYM    /* любой символ, кроме скобки */
%token LB     /* левая скобка */
%token RB     /* правая скобка */
%start s     /* аксиома грамматики */
%%
s:  a    { printf("правильная строка");
        exit(0);
      }
a:  a a   |
    SYM |
    b
b:  LB RB   |
    LB a RB
%%
yylex()
{ char c;
  while ((c = getchar()) != EOF)
  { switch (c)
    { case '(' : return(LB);
      case ')' : return(RB);
      default  : return(SYM);
    }
  }
}
```

В приведенном примере работу лексического анализатора выполняет функция `yylex()`.

3.4.4. Использование в действиях псевдопеременных

Для обеспечения связи между действиями, а также между действиями и лексическим анализатором синтаксический анализатор поддерживает специальный стек, в котором сохраняются значения лексем и нетерминальных символов. Значение лексем автоматически попадает в стек после ее распознавания лексическим анализатором. После применения каждого правила (свертки)

вычисляется значение нетерминала, стоящего в левой части правила, и помещается в вершину стека. Значения элементов правой части примененного правила перед этим выталкиваются из стека. Заметим, что таким образом к моменту свертки любого правила все значения нетерминалов в правой части оказываются вычисленными в результате свертки.

Значения терминальных символов, попадающих в стек по мере их распознавания, по умолчанию, формируются следующим образом:

- одиночный символ (литерал) – числовой код символа во внутреннем представлении;
- имя лексемы – номер типа лексемы.

Значение, которое получает нетерминальный символ в результате свертки правила, будем называть результирующим значением правила. По умолчанию, результирующее значение правила равно значению первого компонента правой части правила.

Описанный выше механизм предоставляет следующие возможности:

- использовать в действиях, осуществляемых после свертки правила, значение любого элемента его правой части;
- формировать в действиях результирующее значение правила.

Доступ к значениям компонент правой части правила обеспечивается набором встроенных псевдопеременных: $\$1, \$2, \dots$, где

$\$i$ – соответствует значению i -го элемента.

Элементы правой части правила нумеруются слева направо без различия лексем и нетерминальных символов.

Например, в правиле

```
Head:  name '(' list ')'
```

псевдопеременные $\$1, \$2, \$3, \4 относились бы соответственно к `name`, `'('`, `list`, `)'`.

Примеры.

expr: '(' expr ')' { \$\$=\$2; }

– Значение нетерминала expr, стоящего в левой части, будет равно значению нетерминала expr, стоящего в правой части правила.

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

– Значением нетерминала expr, стоящего в левой части правила, станет сумма ранее вычисленных значений двух других нетерминалов expr, стоящих в правой части правила.

alpha: beta

– Значение нетерминала alpha будет равно значению нетерминала beta, т. е. неявно выполнится присваивание: $$$=1 .

В записи правил допускает включение действий между любыми компонентами правой части правила, в том числе и перед первым компонентом.

Например, правило может быть записано так:

statement: IF '(' expr ')' { действие1 } THEN statement { действие2 }

При реализации семантики оператора “**IF**” можно в “действие1” посмотреть значение вычисленного выражения “**expr**”, используя псевдопеременную “\$3”, для того чтобы определиться, выполнять “**THEN**” – часть оператора или нет.

Любое правило, содержащее действия уасс приводит к общему формату правила.

Правило из предыдущего примера интерпретируется следующим образом:

statement: IF '(' expr ')' empty1 THEN statement empty2
empty1: { действие1 }

`empty2: { действие2 }`

где **empty1**, **empty2** – нетерминальные символы, добавленные генератором yacc.

Таким образом, в месте, где вставлено действие, при разборе осуществляется свертка по пустому правилу. При этом очередной элемент в стеке значений отводится для хранения значения неявно присутствующего "пустого" нетерминала. Следовательно, в нашем примере, значение компонента **"THEN"** находится в псевдопеременной **"\$6"**, **"\$5"** относится к значению **"empty1"**, а **"\$8"** – к значению **"empty2"**.

В действиях, находящихся внутри правила, с помощью псевдопеременных **\$i** доступны значения расположенных левее элементов, а также результаты предшествующих вставленных в тело действий. Результатом внутреннего действия (т.е. значением неявного нетерминала) является значение, присвоенное в этом действии псевдопеременной **\$\$**, при отсутствии такого присваивания результат действия не определен. Заметим, что присваивание значения псевдопеременной **\$\$** во внутренних действиях не вызывает предварительной установки значения нетерминала, стоящего в левой части правила: это значение в любом случае устанавливается только действием в конце правила или считается равным значению **\$1**.

В процедуре лексического анализа кроме выделения лексем можно предусмотреть некоторую обработку лексем и передачу грамматическому анализатору конкретных значений лексем, вместо передаваемых в вершину стека по умолчанию. С этой целью нужное значение должно быть присвоено внешней переменной целого типа с именем **yylval**. Если функция **yylex** находится в отдельном файле, то эта переменная должна быть объявлена:

```
extern int yyval;
```

Примером значения лексем могут служить числовое значение цифры, вычисленное значение константы, адрес идентификатора в таблице имен (по-

строение таблицы имен удобно осуществлять лексическим анализатором). Отметим, что, хотя значение `yylval` устанавливается с целью использования его в действиях, непосредственное обращение к переменной `yylval` в действии не имеет смысла (поскольку в `yylval` всегда находится значение последней выделенной лексемы). Доступ в действиях к значениям лексем осуществляется с помощью псевдопеременных `$I`.

Пример: Вычисление значения целого числа.

```
%token DIGIT
%%
CONST: DIGIT      |
CONST DIGIT { $$=$1*10+$2; }
%%
yylex()
{
    char c;
    if((c=getchar())>='0' && c<='9')
    {
        yylval = c-'0';
        return (DIGIT);
    }
}
```

Здесь при свертке по первому правилу нетерминал `CONST` получает значение первой цифры, присвоенное в функции `yylex` переменной `yylval`. При каждой свертке по второму правилу явно вычисляется значение нового нетерминала `CONST`. Функция `yylex()` присваивает переменной `yylval` значение введенного символа цифры во внутреннем (двоичном) представлении и возвращает номер типа лексемы `DIGIT`.

3.5. Конфликтные ситуации при грамматическом разборе

Заданная грамматика является неоднозначной, если существует входная строка, которая в соответствии с этой грамматикой может быть разобрана двумя или более различными способами.

Рассмотрим, например, набор правил, описывающих константное арифметическое выражение:

```
expr: CONST      | /*1*/  
     expr '+'expr | /*2*/  
     expr '-'expr | /*3*/  
     expr '*'expr | /*4*/  
     expr '/'expr; /*5*/
```

Описывая возможность построения выражения из двух выражений, соединенных знаком арифметической операции, правила неоднозначно определяют путь разбора некоторых входных строк. Так, строка вида:

25 * 12 - 124

допускает два пути разбора, приводящих к различным группировкам ее элементов:

(25 * 12) - 124 и **25 * (12 - 124)**

С точки зрения работы грамматического анализатора данная ситуация проявляется в неоднозначности выбора действия при вводе лексемы "-" в момент, когда разобранный часть строки приведена к виду $\text{expr} * \text{expr}$ (**25 * 12**). Два возможных действия анализатора состоят в следующем.

Можно ввести следующий символ и без применения правила подстановки перейти в новое состояние (выполнить сдвиг). Выбор сдвига приведет к тому, что в одном из следующих состояний ко второй части конструкции для приведения ее к expr будет применено правило (3), а затем вся полученная конструкция сведется к expr применением правила (4) (выполнится **25 * (12 - 124)**).

Можно сразу применить к конструкции $\text{expr} * \text{expr}$ правило (4), тем самым приведя ее к expr , и без ввода нового символа перейти в очередное состояние (выполнить свертку). Использование свертки в данном состоянии приведет к применению в дальнейшем правила (3) для свертывания оставшейся части конструкции в expr .

Неоднозначность такого рода будем называть конфликтом "сдвиг/свертка".

Возможен другой вид конфликта, состоящий в выборе между двумя возможными свертками; будем называть его конфликтом "свертка/свертка". Для примера подобного конфликта приведем грамматику, задающую десятичную и шестнадцатеричную форму записи константы:

```
const:  const_10 | /*1*/
        const_16 ; /*2*/
const_10: dec_sequence; /*3*/
const_16: hex_sequence 'x'; /*4*/
dec_sequence: digit | /*5*/
              dec_sequence digit; /*6*/
hex_sequence: digit | /*7*/
              ABCDEF | /*8*/
              hex_sequence digit | /*9*/
              hex_sequence ABCDEF; /*10*/
ABCDEF : 'A'|'B'|'C'|'D'|'E'|'F';
digit:   '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
```

При появлении на входе первой же десятичной цифры (если с нее начинается последовательность) после ее замены нетерминалом `digit` возникает конфликт между двумя возможными свертками: к нетерминалу `dec_sequence` в результате применения правила (5) и к нетерминалу `hex_sequence` с помощью правила (7). Заметим, что эта грамматика, в отличие от грамматики в предыдущем примере, не позволяет корректно разобрать какую-либо строку двумя способами и в принципе нетерминал `const` определен однозначно. Однако, алгоритм разбора с просмотром вперед на 1 символ не в состоянии правильно осуществить выбор нужного правила. Следовательно, в этом случае речь идет о неоднозначности грамматики по отношению к принятому в уасс методу разбора.

Поскольку вопрос о принципиальной неоднозначности грамматики формально неразрешим, будем в дальнейшем понимать под неоднозначностью невозможность для анализатора в некоторые моменты разбора выбрать очередное

действие. Каждая ситуация (т.е. появление в некотором состоянии некоторой входной лексемы), которая при разборе способна вызвать конфликт "сдвиг/свертка" или "свертка/свертка", выявляется уасс уже на этапе построения грамматического анализатора. При этом уасс выдает сообщение о числе выявленных конфликтных ситуаций обоих видов, а в выходной файл `u.output` (если он формируется) помещается подробное описание всех конфликтов. Однако, уасс не считает наличие конфликтов фатальной ошибкой грамматики и строит грамматический анализатор, заранее разрешая все возможные конфликты путем выбора в каждой ситуации единственного действия.

При работе уасс используются два способа разрешения конфликтов. Первый способ действует по умолчанию (т.е. при отсутствии специальной пользовательской информации) и заключается в применении следующих двух правил для выбора действия в конфликтных ситуациях:

- В случае **конфликта сдвиг/свертка** по умолчанию делается **сдвиг**.
- В случае **конфликта свертка/свертка** по умолчанию делается **свертка** по тому из конкурирующих правил, которое задано первым во входном файле генератора.

Грамматический анализатор, построенный с использованием этих правил, может не обеспечивать "правильного" с точки зрения пользовательской грамматики разбора. В частности, для первого из приведенных выше примеров разбор заключался бы в сворачивании арифметических выражений справа налево без учета приоритетов операций. Во втором примере в результате замены первой конструкции `digit` нетерминалом `dec_sequence` все числа, начинающиеся с цифры, разбирались бы как десятичные, а появление одной из букв от А до F или символа "x" в конце числа неверно трактовалось как ошибка во входном тексте.

Однако в ряде ситуаций описанный способ разрешения конфликтов приводит к нужному результату.

Например, рассмотрим фрагмент грамматики языка, описывающий условный оператор:

```
statement: IF '(' condition ')' THEN statement | /*1*/  
          IF '(' condition ')' THEN statement ELSE statement; /*2*/
```

Входная строка вида: **if(C1) then if(C2) then S1 else S2** вызвала бы при разборе конфликт сдвиг/свертка в момент просмотра лексемы **else**.

Выбор сдвига, осуществляемый по умолчанию, для данной грамматики приведет к следующему разбору:

```
if (C1) then {if(C2) S1 else S2}
```

В большинстве языков программирования принята именно эта интерпретация (каждый **else** относится к ближайшему предшествующему **if**).

В качестве рекомендации можно отметить, что применение принципа умолчания для конфликтов сдвиг/свертка приводит к положительному результату, если в грамматике принята право-ассоциативная интерпретация соответствующих конструкций и для них отсутствует понятие приоритета. Что касается конфликтов свертка/свертка, то стандартный способ их разрешения оказывается полезным только тогда, когда при любых конфликтах между данными двумя правилами справедлив выбор одного и того же правила.

В любом случае, если уасс сообщил о наличии конфликтных ситуаций, пользователь должен тщательно проанализировать содержательный смысл каждого конфликта и правильность выбранного уасс действия. Вся необходимая для этого информация содержится в файле `u.output`, структура которого будет рассмотрена ниже. Если оказалось, что конфликты разрешены неудовлетворительно, то грамматика должна быть перестроена или уточнена пользователем. В случае конфликтов свертка/свертка всегда требуется изменение самих грамматических правил; для конфликтов сдвиг/свертка есть возможность без

перестройки правил уточнить грамматику путем задания информации о приоритетах и ассоциативности лексем и правил.

В качестве примеров устранения конфликтов путем изменения правил приведем перестроенные варианты рассматривавшихся выше грамматик. Поскольку исходные конфликтные грамматики полностью удовлетворяют требованиям генерируемых ими языков, но содержат недостаточно информации для однозначного разбора, перестройка правил носит уточняющий характер.

Перестроенная грамматика константного арифметического выражения:

```
expr: expr1      |
      expr '+' expr1 |
      expr '-' expr1;
expr1: CONST     |
      expr1 '*' CONST |
      expr1 '/' CONST;
```

Ниже будет приведен также вариант грамматики, полученной из исходной, введением приоритетов (без перестройки правил).

Перестроенная грамматика для задания константы:

```
const:      const_10 |
           const_16;
const_10:   dec_sequence ;
const_16:   hex_sequence 'x' |
           dec_sequence 'x';
dec_sequence: digit |
             dec_sequence digit;
hex_sequence: ABCDEF |
             dec_sequence ABCDEF |
             hex_sequence ABCDEF |
             hex_sequence dec_sequence;
ABCDEF :   'A'|'B'|'C'|'D'|'E'|'F';
digit:     '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9';
```

Рассмотрим теперь второй способ разрешения конфликтов, базирующийся на задании пользователем информации о приоритетах и ассоциативности.

Пример грамматики константного выражения, уточненной заданием приоритетов:

```
%left '+' '-'
%left '*' '/'
%%
expr: CONST          |
     expr '+' expr   |
     expr '-' expr   |
     expr '*' expr   |
     expr '/' expr;
```

Приоритетное разрешение конфликтов сдвиг/свертка состоит в том, что с обоими действиями уасс ассоциирует приоритеты (со сдвигом – приоритет лексем, чтение которой вызывает данный конфликт, со сверткой – приоритет конкурирующего правила) и выбирает более приоритетное действие. В случае равенства приоритетов уасс руководствуется при выборе свойством ассоциативности. Приоритеты и ассоциативность отдельных лексем (явно) и правил (явно и неявно) задаются пользователем, все остальные приоритеты считаются неизвестными. уасс использует для разрешения конфликта данный способ, если известны приоритеты обоих конкурирующих действий. Поэтому для разрешения ряда конфликтов на приоритетной основе необходимо установить приоритеты участвующих в них лексем и правил.

Следует понимать, что задание приоритетов не ведет к устранению конфликтов и не делает грамматику однозначной. Но в отличие от конфликтов, разрешаемых уасс по принципу умолчания, пользователь получает здесь возможность управлять разрешением конфликтов. уасс, сообщая общее число конфликтов, не учитывает в нем конфликты, разрешенные в соответствии с информацией о приоритетах, и не включает в выходной файл u.output описания этих конфликтов.

Приоритеты и ассоциативность лексем задаются в секции определений. Приоритет правила автоматически определяется приоритетом последней лексем в теле правила. Если в секции деклараций для этой лексем не задан при-

оритет или если правая часть правила вообще не содержит лексем, то приоритет правила не определен. Этот принцип можно отменить явным заданием приоритета правила равным приоритету любой (имеющей приоритет) лексемы с помощью следующей директивы, помещенной вслед за правой частью правила (перед точкой с запятой или действием):

```
%prec <лексема>
```

Например, правилу:

```
expr: '-' expr %prec '*' ;
```

директива `%prec` придает приоритет лексемы `"*"` (лексема `"-"` при задании грамматики выражений часто используется для обозначения унарной и бинарной операций, имеющих разный приоритет; с помощью директивы `%prec` унарной операции можно приписать более высокий приоритет. Иногда, чтобы связать с правилом приоритет, не совпадающий с приоритетом ни одной лексемы, вводят псевдолексему, задав ей в секции деклараций уникальный приоритет, и приписывают приоритет псевдолексемы правилу. В примере грамматики настольного калькулятора, приводимом ниже, с операцией "унарный минус" связан приоритет псевдолексемы `UMINUS`.

Сформулируем теперь полностью используемые у нас **правила разрешения конфликтов сдвиг/свертка** на основе информации о **приоритетах** и **ассоциативности** (напомним, что конфликты свертка/свертка разрешаются только по принципу умолчания):

- Если для входной лексемы и правила заданы **приоритеты** и эти приоритеты **различны**, то выбирается действие с **большим приоритетом**. Большой приоритет правила вызывает свертку по нему, большой приоритет лексемы вызывает сдвиг.
- Если **приоритеты** заданы и **совпадают**, то принимается во внимание заданная одновременно с приоритетом **ассоциативность**: в случае **левой ассоци-**

ативности используется **свертка**, в случае **правой** – **сдвиг**. Отсутствие свойства ассоциативности (директива `%nonassoc`) в данном случае указывает на ошибку во входном тексте и анализатор воспримет вызвавшую данный конфликт лексему как ошибочную.

- Если **не задан приоритет** входной лексемы и/или приоритет правила, то действует принцип **разрешения конфликтов по умолчанию**, в результате чего выбирается **сдвиг**.

3.6. Структура информационного файла `y.output`

Основную часть данного файла составляет описание состояний построенного грамматического анализатора. Информация о каждом состоянии приводится в следующем порядке:

1. **Перечень соответствующих данному состоянию конфигураций грамматики** (конфигурация характеризуется определенным грамматическим правилом и позицией в его правой части, достигнутой к данному моменту разбора). Каждая конфигурация представляется правилом с отмеченной с помощью символа подчеркивания "`_`" распознанной частью (позицией конфигурации).

Например, конфигурация:

```
expr: expr +_expr
```

соответствует распознанной при разборе строки по указанному правилу последовательности символов `expr+`.

2. **Действия анализатора при вводе в качестве очередного просматриваемого символа каждой из лексем.**

Различные виды действий указываются следующим образом:

```
<лексема> сдвиг <номер_состояния>
```

– сдвиг при вводе данной лексемы в состояние с указанным номером;

<лексема> свертка <номер_правила>

– свертка при вводе лексемы по правилу с указанным номером;

<лексема> error

– выдача сообщения об ошибке во входных данных ("синтаксическая ошибка") и возврат из процедуры грамматического анализа (дальнейший разбор невозможен);

<лексема> ассерт

– возврат из процедуры грамматического анализа (успешное завершение разбора). Последняя из строк, описывающих действия анализатора, содержит вместо указания лексемы символ "." и сообщает действие, выполняемое анализатором для всех лексем, не перечисленных в данном состоянии. Часто эта строка имеет вид:

. error

и указывает, что все перечисленные лексемы в данном состоянии являются недопустимыми.

3. Перечень переходов для данного состояния. Каждый переход задается строкой

<имя_терминала> переход <номер_состояния>

сообщающей, в какое состояние перейдет анализатор после свертки указанного нетерминала, если его распознавание было начато из данного состояния.

Кроме того, описанию состояния может предшествовать информация о конфликтах обнаруженных уасс для этого состояния и разрешенных по принципу умолчания. Информация о конфликте содержит тип конфликта (свертка/свертка или сдвиг/сдвиг), конкурирующие действия анализатора (при этом для сдвига указывается номер состояния, для свертки – номер правила) и лексему, при появлении которой возникает данный конфликт. Узнать, какое из

возможных действий будет выполнено анализатором, можно из описания самого состояния.

Пример описания состояния:

```
8:Конфликт сдвиг/свертка (сдвиг 5,свертка 2) при +
Состояние 8
a:a_+a
a:a+a_ (2)
+ сдвиг 5
. свертка 2
```

Состоянию 8 здесь соответствуют две различные позиции, достигнутые при разборе по правилу

```
a: a '+' a
```

Конфликт между сверткой по этому правилу и сдвигом в состояние 5 при вводе лексемы "+" разрешен в пользу сдвига. Ввод остальных лексем вызывает свертку.

Пример нахождения конфликтов из практики. В начале файла u.output записано:

```
Состояние 54 конфликты: 1 сдвига/свертки
```

Для состояния 54:

```
State 54
```

```
3 commands: commands . command SEMICOLON
13 case: CASE expression commands .
```

```
error      сдвиг, и переход в состояние 4
DOCASE     сдвиг, и переход в состояние 5
IDEN       сдвиг, и переход в состояние 6
```

```
error      [вывод с использованием правила 13 (case)]
CASE       вывод с использованием правила 13 (case)
```

OTHERWISE	вывод с использованием правила 13 (case)
ENDCASE	вывод с использованием правила 13 (case)
command	переход в состояние 7
switchcase	переход в состояние 8
functioncall	переход в состояние 9
assignment	переход в состояние 10

Видим, что для лексемы `error` существует как возможность сдвига, так и возможность вывода. Значит, эта лексема и вызывает конфликт сдвига/свертки.

После описания состояний возможен ряд сообщений о несвернутых правилах (с указанием этих правил), т.е. о правилах, свертка по которым не будет произведена ни в одном из состояний. Наличие таких правил с большой вероятностью свидетельствует о некорректности грамматики.

В конце файла приводится информация статистического характера о количестве терминальных и нетерминальных символов, грамматических правил и состояний. Указывается число конфликтов каждого типа.

3.7. Обработка ошибок при грамматическом разборе

Если входной поток не удовлетворяет заданной грамматике, то грамматический анализатор в момент ввода лексем, делающей невозможным продолжение разбора, фиксирует ошибку.

Стандартной реакцией грамматического анализатора на ошибку является выдача сообщения ("синтаксическая ошибка") и прекращение разбора.

Сообщения об ошибках можно сделать более информативными, задав собственную версию функции `uerror()`. Однако, наиболее важная задача состоит в том, чтобы заставить анализатор в этом случае продолжать просмотр входного потока, в частности, для выявления последующих ошибок.

Применяемый уасс механизм восстановления основан на чтении и отбрасывании некоторого числа входных лексем; от пользователя требуется введение дополнительных грамматических правил, указывающих, в каких конструкциях синтаксические ошибки являются допустимыми (в отношении возможности восстановления). Одновременно эти правила определяют путь дальнейшего разбора для ошибочных ситуаций. Для указания точек допустимых ошибок используется зарезервированное с этой целью имя лексемы **error**.

Пример:

```
a:  b c d ;      | /*1*/  
    b c error;   /*2*/  
d:  d1 d2 d3;   /*3*/
```

Второе правило указывает путь разбора в случае, если при распознавании нетерминала *a* встретится ошибка после выделения элементов *b* и *c*.

Рассмотрим порядок работы анализатора при появлении во входном потоке ошибочной лексемы (т.е. лексемы, ввод которой в данном состоянии вызывает действие `error`).

Фиксируется состояние ошибки; вызывается функция `uerror()` для выдачи сообщения.

Путем обратного просмотра пройденных состояний, начиная с данного, делается попытка найти состояние, в котором допустима лексема `error`. Отсутствие такого состояния говорит о невозможности восстановления, и разбор прекращается.

Осуществляется возврат в найденное состояние.

Выполняется действие, заданное в этом состоянии для лексемы `error`. Очередной входной лексемой становится лексема, вызвавшая ошибку.

Разбор продолжается, но анализатор остается в состоянии ошибки до тех пор, пока не будут успешно прочитаны и обработаны три подряд идущие лек-

семь. При нахождении анализатора в состоянии ошибки, для предотвращения потока ложных сообщений, обработка ошибочной лексемы заключается в том, что сообщения об ошибке не выдается, а сама лексема игнорируется.

После обработки трех допустимых лексем считается, что восстановление произошло, и анализатор выходит из состояния ошибки.

Итак, грамматический анализатор, встретив ошибку, пытается найти ближайшую точку в грамматике, где разрешена лексема `error`. При этом сначала делается попытка возврата в рамках правила, по которому шел разбор в момент появления ошибочной лексемы, затем поиск распространяется на правила все более высокого уровня. В примере, приведенном в начале раздела, ввод недопустимой лексемы после того, как прочитана строка `b c d1 d2` вызовет возврат к состоянию, характеризующемуся конфигурациями:

```
a: b c_d;  
a: b c_error;
```

и продолжение разбора по правилу (2).

Часто правила, учитывающие возможность ошибки, задаются на уровне основных структурных единиц входного текста. Например, для пропуска в тексте ошибочных операторов может быть использовано правило

```
statement: error;
```

При этом восстановление из состояния ошибки произойдет после нахождения трех лексем, которые могут следовать после оператора, например, начинать новый оператор. Если точно распознать начало оператора невозможно, то ошибочное состояние может быть подавлено преждевременно, а обработка нового оператора начата с середины ошибочного, что, вероятно, приведет к повторному сообщению об ошибке (на самом деле не существующей). Учитывая это, более надежного результата следует ожидать от правил вида:

```
statement: error ';' ;
```

Здесь восстановление произойдет только после нахождения ";" и двух начальных лексем следующего оператора; все лексеммы после найденной ошибочной до ";" будут отброшены.

С правилами, включающими лексему `error`, могут быть связаны действия. С их помощью пользователь может самостоятельно обработать ошибочную ситуацию. Кроме обычных операторов, здесь можно использовать специальные операторы `yterror` и `yuclearin`, которые уасс на макроуровне расширяет в нужные последовательности. Оператор `yterror` аннулирует состояние ошибки. Таким образом, можно отменить действие принципа "трех лексем". Это помогает предотвратить маскирование новых ошибок в случаях, когда конец ошибочной конструкции распознается самим пользователем или однозначно определяется в правиле по меньшему числу лексем.

Оператор `yuclearin` стирает хранимую анализатором последнюю входную лексему, если поиск нужной точки для возобновления ввода обеспечивается в заданном пользователем действии.

Приведем общую форму правила с восстановительным действием

```
оператор : error { resynch();
                yuclearin;
                yterror;
            }
```

Предполагается, что пользовательская процедура `resynch()` просматривает входной поток до начала очередного оператора. Вызвавшая ошибку лексема, хранимая анализатором в качестве входной лексеммы, стирается, после этого гасится состояние ошибки.

3.8. Стандартные функции

Построенный с помощью уасс грамматический анализатор использует следующие функции:

yyparse() – функция синтаксического анализа; возвращает значение 0, если входной текст соответствует аксиоме грамматики, и 1 в противном случае. Эта функция строится генератором yacc на основе описания грамматики языка.

yylex() – функция лексического анализа; многократно вызывается функцией **yyparse()** для выделения лексем из входного потока символов; возвращает в качестве значения номер типа лексемы; эта функция обычно строится генератором lex.

main() – функция входа в грамматический анализатор; организует обращение к функции **yyparse()** и возвращает значение этой функции.

yyerror(s) – функция обработки ошибок; аргумент **s** – указатель на строку символов, содержащую сообщение об ошибке.

Обычно функции **yyparse()** и **yylex()** строятся генераторами yacc и lex, а функции **main()** и **yyerror(s)** имеются в стандартной библиотеке yacc. Однако, любые из них можно перепрограммировать, поместив, например, в секцию подпрограмм.

Библиотечные функции имеют следующий вид:

```
main()
{return (yyparse());}

#include <stdio.h>
yyerror(s) char *s; {
    fprintf(stderr, "%s\n", s);}
}
```

3.9. Пример входного файла

Ниже приведен полный входной файл для yacc, реализующий небольшой настольный калькулятор; калькулятор имеет 26 регистров, помеченных буквами от a до z, и разрешает использовать арифметические выражения, содержа-

шие операции +, -, *, /, % (остаток от деления), & (побитовое и), | (побитовое или) и присваивание. Как и в Си, целые числа, начинающиеся с 0, считаются восьмеричными, все остальные – десятичными.

В примере демонстрируются способы использования приоритетов для разрешения конфликтов, а также простые операции по восстановлению из состояния ошибки. Калькулятор работает в интерактивном режиме с построчным формированием выхода.

```
%token DIGIT
%token LETTER
%left '|' /* задание приоритетов */
%left '&' /* операций */
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* установка приоритета операции унарный ми
             нус */
%{ /* описания, используемые */
int base, regs[26]; /* в действиях */
%}
%% /* начало секции правил */
list:      |
           list stat '\n'      |
           list stat error '\n' {yyerror("синтаксическая ошибка"); }
stat:      expr {printf("%d\n",$1);} |
           LETTER '=' expr {regs[$1]=$3; }
expr:      '(' expr ')' { $$=$2; } |
           expr '+' expr { $$=$1+$3; } |
           expr '-' expr { $$=$1-$3; } |
           expr '*' expr { $$=$1*$3; } |
           expr '/' expr { $$=$1/$3; } |
           expr '%' expr { $$=$1%$3; } |
           expr '&' expr { $$=$1&$3; } |
           expr '|' expr { $$=$1|$3; } |
           '-' expr %prec UMINUS { $$= -$2; } |
           LETTER { $$=regs[$1]; } |
           number;
number:    DIGIT { $$=$1; base=10;
                if($1==0) base=8;
                } |
           number DIGIT { $$=base*$1+$2; }
```



```

%% /* начало секции программ */
/*
* Программа лексического анализа
* для строчных латинских букв возвращает LETTER,
* значение yylval от 0 до 25;
* для цифр - DIGIT, значение yylval от 0 до 9;
* остальные символы возвращаются непосредственно
*/
yylex()
{
    int c;
    while( (c=getchar()) == ' ' );
    if( c>='a' && c<='z' )
    { yylval = c - 'a';
      return(LETTER);
    }
    if( c>='0' && c<='9' )
    { yylval = c - '0';
      return(DIGIT);
    }
    return(c);
}

```

В приведенном примере функция `yylex()` записана в виде программы на языке Си.

Можно построить эту функцию генератором `lex`. Входной файл для `lex` приведен ниже.

```

%{
#include <y.tab.h>
extern int yylval;
%}
%%
[a-z]      { yylval = yytext[0] - 'a';
            return(LETTER);
          }
[0-9]      { yylval = yytext[0] - '0';
            return(DIGIT);
          }
[ \t]+    ;
\n        return('\n');
.         return(yytext[0]);

```

ГЛАВА 4. СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ ГЕНЕРАТОРОВ YACC И LEX

Вызов выполнения генератора yacc имеет вид:

yacc -vd Yfile

где **Yfile** – имя входного файла, построенного в соответствии с требованиями структуры входного файла yacc.

Флаги команды являются необязательными и имеют следующий смысл:

v – сформировать в файле **y.output** подробное описание грамматического анализатора;

d – сформировать в файле **y.tab.h** описание лексем и номеров их типов. Данный файл необходимо поместить в include секцию входного файла lex, если планируется использование лексем в секции правил.

В результате выполнения этой команды yacc создаст лексический анализатор в виде текста программы на языке C и поместит его в файл со стандартным именем **y.tab.c**.

Основной результат работы yacc – процедура **yyparse** и грамматические таблицы.

Для получения выполняемой программы синтаксического анализатора (загрузочного модуля) необходимо выполнить компиляцию программы **y.tab.c** и скомпоновать ее с программами из стандартной библиотеки генератора yacc.

В случае если функция **yulex()** реализована в виде программы на языке Си и помещена в секцию подпрограмм входного файла yacc, достаточно выполнить компиляцию:

cc y.tab.c -ly

где **-ly** – подключение стандартной библиотеки генератора уасс.

Если функция `yulex()` построена генератором `lex` и помещена в файл **lex.yy.c**, то необходимо выполнить следующую команду:

```
cc y.tab.c lex.yy.c -ly -ll
```

В результате выполнения этой любой из этих команд будет создан выполняемый файл (загрузочный модуль) со стандартным именем **a.out**.

Разработчик синтаксического анализатора имеет возможность подключить собственные функции `main()` и `yerror()` вместо библиотечных. Для этого достаточно поместить текст этих функции в секцию подпрограмм входного файла уасс.

Наличие сгенерированного текста программы в файле **y.tab.c** дает программисту дополнительные возможности для внесения корректив в работу синтаксического анализатора.

4.1. Использование генераторов `flex` и `bison`

Генераторы `flex` и `bison` являются совместимыми аналогами `lex` и `уасс`, разработанными в рамках проекта разработанными в рамках проекта GNU. Данные генераторы часто предустановлены в дистрибутивы систем на основе GNU, а значит, предоставляются во множестве дистрибутивов Linux.

Использование данных генераторов мало отличается от случая с `lex` и `уасс`.

Вызов выполнения генератора `flex` имеет вид:

```
flex Lfile
```

где **Lfile** – имя входного файла, построенного в соответствии с требованиями структуры входного файла flex.

В результате выполнения этой команды flex создаст лексический анализатор в виде текста программы на языке C и поместит его в файл со стандартным именем **lex.yy.c** аналогичный файлу в случае использования lex.

Для получения выполняемой программы лексического анализатора (загрузочного модуля) необходимо выполнить компиляцию программы **lex.yy.c** и скомпоновать ее с программами из стандартной библиотеки генератора lex. Это выполняется следующей командой:

```
cc lex.yy.c -lfl
```

где **-lfl** – подключение стандартной библиотеки генератора flex, аналог команды -ll для lex. В результате выполнения этой команды будет создан выполняемый файл (загрузочный модуль) со стандартным именем **a.out**.

Вызов выполнения генератора yacc имеет вид:

```
bison -vd Yfile
```

где **Yfile** – имя входного файла, построенного в соответствии с требованиями структуры входного файла bison.

Флаги команды являются необязательными и имеют тот же смысл, что и в случае с yacc.

В результате выполнения этой команды yacc создаст лексический анализатор в виде текста программы на языке C и поместит его в файл со именем **Yfile.tab.c** (зависящим от имени входного файла).

В случае если функция уulex() реализована в виде программы на языке Си и помещена в секцию подпрограмм входного файла bison, достаточно выполнить компиляцию:

```
cc Yfile.tab.c -ly
```

где **-ly** – подключение стандартной библиотеки генератора bison.

Если функция `yylex()` построена генератором flex и помещена в файл **lex.yy.c**, то необходимо выполнить следующую команду:

```
cc y.tab.c lex.yy.c -ly -lfl
```

В результате выполнения этой любой из этих команд будет создан выполняемый файл (загрузочный модуль) со стандартным именем **a.out**.

4.2. Работа с flex и bison на ОС Windows

Поскольку lex, yacc, flex и bison доступны в Unix-системах, использование их напрямую в Windows невозможно. Тем не менее, есть два варианта решения этой проблемы: использование аналогичных пакетов, собранных под mingw, что однако сопряжено с использованием иных команд для работы с данными инструментами, а также, возможно, с иным их поведением. Вторым и более предпочтительным вариантом является использование среды Cygwin, обеспечивающей Unix-подобное поведение и имеющей соответственные наборы необходимых программ.

Скачать установщик среды можно по ссылке <https://cygwin.com/install.html>. В установщике необходимо выбрать режим установки из интернета. Дождавшись загрузки и появления списка устанавливаемых пакетов, выбрать Devel/flex и Devel/bison. Также необходимо установить Devel/gcc-core для компиляции сгенерированных исходных файлов лексического и синтаксического анализаторов.

4.3. Пример грамматики языка SQL

```
/* symbolic tokens */

%union {
    int intval;
    double floatval;
    char *strval;
    int subtok;
}

%token NAME
%token STRING
%token INTNUM APPROXNUM

    /* operators */

%left OR
%left AND
%left NOT
%left <subtok> COMPARISON /* = <> < > <= >= */
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

    /* literal keyword tokens */

%token ALL AMMSC ANY AS ASC AUTHORIZATION BETWEEN BY
%token CHARACTER CHECK CLOSE COMMIT CONTINUE CREATE CURRENT
%token CURSOR DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT
DOUBLE
%token ESCAPE EXISTS FETCH FLOAT FOR FOREIGN FOUND FROM GOTO
%token GRANT GROUP HAVING IN INDICATOR INSERT INTEGER INTO
%token IS KEY LANGUAGE LIKE NULLX NUMERIC OF ON OPEN OPTION
%token ORDER PARAMETER PRECISION PRIMARY PRIVILEGES PROCEDURE
%token PUBLIC REAL REFERENCES ROLLBACK SCHEMA SELECT SET
%token SMALLINT SOME SQLCODE SQLERROR TABLE TO UNION
%token UNIQUE UPDATE USER VALUES VIEW WHENEVER WHERE WITH WORK

%%

sql_list:
```

```

        sql ';'      { end_sql(); }
    |    sql_list sql ';' { end_sql(); } ;

    /* schema definition language */
sql:    schema      ;

schema:
        CREATE SCHEMA AUTHORIZATION user
opt_schema_element_list;

opt_schema_element_list:
        /* empty */
    |    schema_element_list ;

schema_element_list:
        schema_element
    |    schema_element_list schema_element ;

schema_element:
        base_table_def
    |    view_def
    |    privilege_def ;

base_table_def:
        CREATE TABLE table '('
base_table_element_commalist ')' ;

base_table_element_commalist:
        base_table_element
    |    base_table_element_commalist ','
base_table_element ;

base_table_element:
        column_def
    |    table_constraint_def ;

column_def:
        column data_type column_def_opt_list ;

column_def_opt_list:
        /* empty */
    |    column_def_opt_list column_def_opt ;

```

```

column_def_opt:
    NOT NULLX
    | NOT NULLX UNIQUE
    | NOT NULLX PRIMARY KEY
    | DEFAULT literal
    | DEFAULT NULLX
    | DEFAULT USER
    | CHECK '(' search_condition ')'
    | REFERENCES table
    | REFERENCES table '(' column_commalist ')' ;

table_constraint_def:
    UNIQUE '(' column_commalist ')'
    | PRIMARY KEY '(' column_commalist ')'
    | FOREIGN KEY '(' column_commalist ')'
      REFERENCES table
    | FOREIGN KEY '(' column_commalist ')'
      REFERENCES table '(' column_commalist ')'
    | CHECK '(' search_condition ')' ;

column_commalist:
    column
    | column_commalist ',' column ;

view_def:
    CREATE VIEW table opt_column_commalist
    AS query_spec opt_with_check_option ;

opt_with_check_option:
    /* empty */
    | WITH CHECK OPTION ;

opt_column_commalist:
    /* empty */
    | '(' column_commalist ')' ;

privilege_def:
    GRANT privileges ON table TO grantee_commalist
    opt_with_grant_option ;

opt_with_grant_option:
    /* empty */
    | WITH GRANT OPTION ;

```



```

privileges:
    ALL PRIVILEGES
    |
    ALL
    |
    operation_commalist ;

operation_commalist:
    operation
    |
    operation_commalist ',' operation ;

operation:
    SELECT
    |
    INSERT
    |
    DELETE
    |
    UPDATE opt_column_commalist
    |
    REFERENCES opt_column_commalist ;

grantee_commalist:
    grantee
    |
    grantee_commalist ',' grantee ;

grantee:
    PUBLIC
    |
    user ;

/* cursor definition */
sql:
    cursor_def ;

cursor_def:
    DECLARE cursor CURSOR FOR query_exp
    opt_order_by_clause ;

opt_order_by_clause:
    /* empty */
    |
    ORDER BY ordering_spec_commalist ;

ordering_spec_commalist:
    ordering_spec
    |
    ordering_spec_commalist ',' ordering_spec ;

ordering_spec:
    INTNUM opt_asc_desc

```

```

        |      column_ref opt_asc_desc ;

opt_asc_desc:
        /* empty */
        |      ASC
        |      DESC ;

        /* manipulative statements */

sql:      manipulative_statement ;

manipulative_statement:
        close_statement
        |
        |      commit_statement
        |      delete_statement_positioned
        |      delete_statement_searched
        |      fetch_statement
        |      insert_statement
        |      open_statement
        |      rollback_statement
        |      select_statement
        |      update_statement_positioned
        |      update_statement_searched      ;

close_statement:
        CLOSE cursor ;

commit_statement:
        COMMIT WORK ;

delete_statement_positioned:
        DELETE FROM table WHERE CURRENT OF cursor ;

delete_statement_searched:
        DELETE FROM table opt_where_clause ;

fetch_statement:
        FETCH cursor INTO target_commalist ;

insert_statement:
        INSERT INTO table opt_column_commalist val-
ues_or_query_spec ;

```

```

values_or_query_spec:
    VALUES '(' insert_atom_commalist ')'
    |
    query_spec ;

insert_atom_commalist:
    insert_atom
    |
    insert_atom_commalist ',' insert_atom ;

insert_atom:
    atom
    |
    NULLX ;

open_statement:
    OPEN cursor ;

rollback_statement:
    ROLLBACK WORK ;

select_statement:
    SELECT opt_all_distinct selection
    INTO target_commalist
    table_exp ;

opt_all_distinct:
    /* empty */
    |
    ALL
    |
    DISTINCT ;

update_statement_positioned:
    UPDATE table SET assignment_commalist
    WHERE CURRENT OF cursor ;

assignment_commalist:
    |
    assignment
    |
    assignment_commalist ',' assignment ;

assignment:
    column '=' scalar_exp
    |
    column '=' NULLX ;

update_statement_searched:
    UPDATE table SET assignment_commalist
    opt_where_clause ;

```

```

target_commalist:
    target
    | target_commalist ',' target ;

target:
    parameter_ref ;

opt_where_clause:
    /* empty */
    | where_clause ;

    /* query expressions */

query_exp:
    query_term
    | query_exp UNION query_term
    | query_exp UNION ALL query_term ;

query_term:
    query_spec
    | '(' query_exp ')' ;

query_spec:
    SELECT opt_all_distinct selection table_exp ;

selection:
    scalar_exp_commalist
    | '*' ;

table_exp:
    from_clause
    opt_where_clause
    opt_group_by_clause
    opt_having_clause ;

from_clause:
    FROM table_ref_commalist ;

table_ref_commalist:
    table_ref
    | table_ref_commalist ',' table_ref ;

```

```

table_ref:
    table
    | table range_variable ;

where_clause:
    WHERE search_condition ;

opt_group_by_clause:
    /* empty */
    | GROUP BY column_ref_commalist ;

column_ref_commalist:
    column_ref
    | column_ref_commalist ',' column_ref ;

opt_having_clause:
    /* empty */
    | HAVING search_condition ;

    /* search conditions */

search_condition:
    | search_condition OR search_condition
    | search_condition AND search_condition
    | NOT search_condition
    | '(' search_condition ')'
    | predicate ;

predicate:
    comparison_predicate
    | between_predicate
    | like_predicate
    | test_for_null
    | in_predicate
    | all_or_any_predicate
    | existence_test ;

comparison_predicate:
    scalar_exp COMPARISON scalar_exp
    | scalar_exp COMPARISON subquery ;

between_predicate:
    scalar_exp NOT BETWEEN scalar_exp AND scalar_exp

```

```

        | scalar_exp BETWEEN scalar_exp AND scalar_exp ;

like_predicate:
    scalar_exp NOT LIKE atom opt_escape
    | scalar_exp LIKE atom opt_escape ;

opt_escape:
    /* empty */
    | ESCAPE atom ;

test_for_null:
    column_ref IS NOT NULLX
    | column_ref IS NULLX ;

in_predicate:
    scalar_exp NOT IN '(' subquery ')'
    | scalar_exp IN '(' subquery ')'
    | scalar_exp NOT IN '(' atom_commalist ')'
    | scalar_exp IN '(' atom_commalist ')' ;

atom_commalist:
    atom
    | atom_commalist ',' atom ;

all_or_any_predicate:
    scalar_exp COMPARISON any_all_some subquery ;

any_all_some:
    ANY
    | ALL
    | SOME ;

existence_test:
    EXISTS subquery ;

subquery:
    '(' SELECT opt_all_distinct selection table_exp
    ')' ;

    /* scalar expressions */

scalar_exp:
    scalar_exp '+' scalar_exp

```

```

| scalar_exp '-' scalar_exp
| scalar_exp '*' scalar_exp
| scalar_exp '/' scalar_exp
| '+' scalar_exp %prec UMINUS
| '-' scalar_exp %prec UMINUS
atom
column_ref
function_ref
>(' scalar_exp ') ;

```

scalar_exp_commalist:

```

| scalar_exp
| scalar_exp_commalist ',' scalar_exp ;

```

atom:

```

| parameter_ref
| literal
| USER ;

```

parameter_ref:

```

| parameter
| parameter parameter
| parameter INDICATOR parameter ;

```

function_ref:

```

| AMMSC '(' '*' ')'
| AMMSC '(' DISTINCT column_ref ')'
| AMMSC '(' ALL scalar_exp ')'
| AMMSC '(' scalar_exp ')' ;

```

literal:

```

| STRING
| INTNUM
| APPROXNUM ;

```

/* miscellaneous */

table:

```

| NAME
| NAME '.' NAME ;

```

column_ref:

```

NAME

```

```

|     NAME '.' NAME      /* needs semantics */
|     NAME '.' NAME '.' NAME ;

/* data types */

data_type:
    CHARACTER
|     CHARACTER '(' INTNUM ')'
|     NUMERIC
|     NUMERIC '(' INTNUM ')'
|     NUMERIC '(' INTNUM ',' INTNUM ')'
|     DECIMAL
|     DECIMAL '(' INTNUM ')'
|     DECIMAL '(' INTNUM ',' INTNUM ')'
|     INTEGER
|     SMALLINT
|     FLOAT
|     FLOAT '(' INTNUM ')'
|     REAL
|     DOUBLE PRECISION ;

/* the various things you can name */

column:      NAME ;

cursor:      NAME ;

parameter:   PARAMETER /* :name handled in parser */ ;

range_variable: NAME ;

user:        NAME ;

/* embedded condition things */
sql:         WHENEVER NOT FOUND when_action
|           WHENEVER SQLERROR when_action ;

when_action: GOTO NAME
|           CONTINUE ;
%%

```


4.4. Формат обмена данными MapInfo MID/MIF

Универсальный формат MapInfo MID/MIF является текстовым (ASCII) и позволяет сопоставлять различным графическим элементам настраиваемые данные.

Данные MapInfo хранятся в двух файлах: графическая информация содержится в файлах с расширением MIF (MapInfo Interchange Format), текстовая – в файлах с расширением MID. MIF-файл состоит из двух частей: заголовка и секции данных. Заголовок файла содержит информацию о том, как создавать таблицы MapInfo, секция данных содержит определения графических объектов.

Ниже представлен пример файла в формате MIF, а также парсер и лексический анализатор для разбора файлов в таком формате.

```
Version 2
Delimiter ","
CoordSys Earth Projection 8, 1001, "m", 51, 0, 1, 9500000, 0
  Bounds (1250718.46099, -10002137.4978) (17749281.539,
10002137.4978)
```

```
Columns 1
  SEG_I_D_ Decimal(16,6)
```

Data

```
Pline 11
  9494423.0000      6086811.5000
  9494468.0000      6086876.0000
  9494506.0000      6086926.0000
  9494550.0000      6086976.0000
  9494584.0000      6087015.5000
  9494624.0000      6087057.5000
  9494667.0000      6087106.0000
  9494709.0000      6087146.0000
  9494788.0000      6087218.5000
  9494845.0000      6087272.0000
  9494940.0000      6087339.0000
Pen (1,2,0)
```

Text

"УСЛОВНЫЕ ОБОЗНАЧЕНИЯ"

-0.46303 0.079249 1.694543 0.240865

Font ("Arial Cyr",2,0,0)

Text

"ТЕКТОНИЧЕСКАЯ КАРТА"

-0.503093 0.576512 1.93327 0.778532

Font ("Arial Cyr",1,0,0)

Point -1.661616 -0.030303

Symbol (46,255,8)

Point -1.661616 -0.232323

Symbol (46,16711680,8)

Text

"134"

-1.611111 0.010101 -1.392929 0.151515

Font ("Arial Cyr",2,0,0)

Rect -1.984848 -0.616162 -1.974747 -0.616162

Pen (1,2,0)

Brush (2,16777215,16777215)

Лексический анализ

```
%start index coords column colfld mult transf just labln
```

```
%{
```

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

```
#include "y.tab.h"
```

```
#include "glob.c"
```

```
int zzsect=0;
```

```
struct flstruct *cs;
```

```
extern struct flstruct hs[];
```

```
extern struct flstruct gs[];
```

```
extern char *tpfld;
```

```
int zzihs=0;
```

```
int zznfld=0;
```

```
int zzifld=0;
```

```
int zzics=0;
```

```
int zzigs=0;
```

```

int zzstr=0;
int zznobj=0;
int zzsobj=0;
int zzuid=0;
int zznid=0;
int zzobtype=0;
double zzdt[4]= {0.0,0.0,0.0,0.0};
int zzvers=0;
char zzd1m='\t';
char zztext[256];
extern FILE *ffh,*ffg,*ffd,*ffi;
extern char zzrazd;
extern struct formats ftbl[];
char *alltrim(char *s);
char *name_cor(char *s);
void gr_obj();
void ini_obj(int tp);
%}
empty    [ \t]*
letters  [A-Za-z]+
anynum   [+ -]?[0-9]+(\.[0-9]*(E[+-]?[0-9]+)?)?
name     [A-Za-z][A-Za-z_0-9]*
version  [Vv][Ee][Rr][Ss][Ii][Oo][Nn]
charset  [Cc][Hh][Aa][Rr][Ss][Ee][Tt]
delim    [Dd][Ee][Ll][Ii][Mm][Ii][Tt][Ee][Rr]
unique   [Uu][Nn][Ii][Qq][Uu][Ee]
index    [Ii][Nn][Dd][Ee][Xx]
coords   [Cc][Oo][Oo][Rr][Dd][Ss][Yy][Ss]
earth    [Ee][Aa][Rr][Tt][Hh]
proj     [Pp][Rr][Oo][Jj][Ee][Cc][Tt][Ii][Oo][Nn]
affine   [Aa][Ff][Ff][Ii][Nn][Ee]
units    [Uu][Nn][Ii][Tt][Ss]
bounds   [Bb][Oo][Uu][Nn][Dd][Ss]
nearth   [Nn][Oo][Nn][Ee][Aa][Rr][Tt][Hh]
layout   [Ll][Aa][Yy][Oo][Uu][Tt]
table    [Tt][Aa][Bb][Ll][Ee]
window   [Ww][Ii][Nn][Dd][Oo][Ww]
transf   [Tt][Rr][Aa][Nn][Ss][Ff][Oo][Rr][Mm]
colum    [Cc][Oo][Ll][Uu][Mm][Nn][Ss]
char     [Cc][Hh][Aa][Rr]
integ    [Ii][Nn][Tt][Ee][Gg][Ee][Rr]
small    [Ss][Mm][Aa][Ll][Ll][Ii][Nn][Tt]
decim    [Dd][Ee][Cc][Ii][Mm][Aa][Ll]

```

```

float    [Ff][Ll][Oo][Aa][Tt]
date     [Dd][Aa][Tt][Ee]
logic    [Ll][Oo][Gg][Ii][Cc][Aa][Ll]
data     [Dd][Aa][Tt][Aa]
none     [Nn][Oo][Nn][Ee]
point    [Pp][Oo][Ii][Nn][Tt]
symbol   [Ss][Yy][Mm][Bb][Oo][Ll]
line     [Ll][Ii][Nn][Ee]
pen      [Pp][Ee][Nn]
pline    [Pp][Ll][Ii][Nn][Ee]
mult     [Mm][Uu][Ll][Tt][Ii][Pp][Ll][Ee]
smooth   [Ss][Mm][Oo][Oo][Tt][Hh]
region   [Rr][Ee][Gg][Ii][Oo][Nn]
brush    [Bb][Rr][Uu][Ss][Hh]
center   [Cc][Ee][Nn][Tt][Ee][Rr]
arc      [Aa][Rr][Cc]
text     [Tt][Ee][Xx][Tt]
font     [Ff][Oo][Nn][Tt]
spacing  [Ss][Pp][Aa][Cc][Ii][Nn][Gg]
justify  [Jj][Uu][Ss][Tt][Ii][Ff][Yy]
left     [Ll][Ee][Ff][Tt]
right    [Rr][Ii][Gg][Hh][Tt]
angle    [Aa][Nn][Gg][Ll][Ee]
labelln  [Ll][Aa][Bb][Ee][Ll][ ]+[Ll][Ii][Nn][Ee]
simple    [Ss][Ii][Mm][Pp][Ll][Ee]
arrow    [Aa][Rr][Rr][Oo][Ww]
rect     [Rr][Ee][Cc][Tt]
rourect  [Rr][Oo][Uu][Nn][Dd][Rr][Ee][Cc][Tt]
ellips   [Ee][Ll][Ll][Ii][Pp][Ss][Ee]
nmtype   {char}|{integ}|{small}|{decim}|{float}|{date}|{logic}
%%
\15      { printf("0D !!!!!\n"); }
\32      { printf("EOF\n"); return(EOF); }
{version} { zzihs=1;
           return(yvers);
         }
{charset} { zzihs=2;
           return(ychset);
         }
{delim}   { zzihs=3;
           return(ydelim);
         }
{unique}  { BEGIN index;

```

```

        zzihs=4;
        return(yuniq);
    }
{index}      { BEGIN index;
              zzihs=5;
              return(yindex);
            }
<index>{anynum}(,{anynum})* { BEGIN 0;
                              strcpy(hs[zzih].val,yttext);
                              return(ynums);
                            }
{coords}     { BEGIN coords;
              printf("l coord\n");
              return(ycoords);
            }
<coords>\n   {}
<coords>({earth}|{nearth}) { strcpy(hs[6].val,yttext);
                              printf("l (n)earth\n");
                              if (yttext[0]=='e' ||
yyttext[0]=='E')
                                  return(yearth);
                                  else return(ynearth);
                              }
<coords>{layout}[ ]+{units} { strcpy(hs[6].val,yttext);
                              zzihs=30;
                              printf("l layout\n");
                              return(ylay);
                              }
<coords>{table}      { strcpy(hs[6].val,yttext);
                              zzihs=31;
                              printf("l table\n");
                              return(ytable);
                              }
<coords>{window}     { strcpy(hs[6].val,yttext);
                              zzihs=32;
                              printf("l window\n");
                              return(ywind);
                              }
<coords>{proj}      { zzihs=7;
                      printf("l proj\n");
                      return(yproj);
                    }
<coords>{affine}[ ]+{units} {

```

```

        zzihs=23;
        printf("l affine\n");
        return(yaff);
    }
<coords>{units} { zzihs=9;
                  printf("l units\n");
                  return(yunit);
                }
<coords>{bounds} { zzihs=19;
                  printf("l bounds\n");
                  return(ybound);
                }
<coords,transf>{anynum} { strcpy(hs[zzih].val,yytext);
                          printf("l coor num=(%s)\n",yytext);
                          return(ynum);
                        }
<coords,transf>, { zzihs++;
                  printf("l coor ,\n");
                  return(',')');
                }
<coords>\) { if (zzih==20) zzihs++;
            printf("l coor )\n");
            return(')');
          }
<coords>\"[^\"]*\" { strcpy(hs[zzih].val,yytext);
                  printf("l coor str=(%s)\n",yytext);
                  return(ystr);
                }
<column>{anynum} { BEGIN colfld;
                  strcpy(hs[37].val,yytext);
                  zzsect=1;
                  zznfld=strtol(yytext,NULL,10);
                  if ((cs=(struct flstruct *)
                      calloc(zznfld,sizeof(
                                  struct flstruct)))==NULL)
                    { printf("КГ еѣ в ГВ Ĩ -пвĚ\n");
                      exit(1);
                    }
                  if ((tpfld=(char
*)calloc(zznfld+1,sizeof(char)))==NULL)
                    { printf("КГ еѣ в ГВ Ĩ -пвĚ\n");
                      exit(1);
                    }
                }

```

```

        printf("1 colum num=(%s)\n",yytext);
        return(ynum);
    }
<colfld>{nmtype}({empty}\({anynum}({,}{anynum})?\))? {
    strcpy(cs[zzics].val,yytext);
    if (tolower(yytext[0])=='c' ||
        tolower(yytext[0])=='d' && tolow-
er(yytext[1])=='a')
        tpfld[zzics]='c';
    else tpfld[zzics]='n';
    zzics++;
    return(ytype);
}
^{empty}{data} { int i;
    BEGIN 0;
    zzsect=2;
    fprintf(ffh,"\n\"%s\"",hs[0].val);
    for (i=1;i<LHS;i++)
    { if (hs[i].tpfld=='c' && hs[i].val[0]!='')
        fprintf(ffh,"%c\"%s\"",zzrazd,hs[i].val);
        else fprintf(ffh,"%c%s",zzrazd,hs[i].val);
    }
    for (i=0;i<zznfld;i++)
        fprintf(ffh,"%c\"%s\"",zzrazd,cs[i].val);
    fprintf(ffh,"\n");
    fprintf(ffg,"\n");
    zzvers=hs[1].val[0];
    zzdlm=hs[3].val[1];
    printf("1 Vers=(%c) Delim=(%c) (%s)",
        zzvers, zzdlm, hs[3].val);

    free(cs);
    zzsect=2;
    printf("1 data\n");
    return(ydata);
}
<colfld>^{empty}{name} {

fprintf(ffh,"%c%s",zzrazd,alltrim(yytext));

fprintf(ffg,"%c%s",zzrazd,alltrim(yytext));
    printf("Name=(%s)\n",yytext);
    return(yname);
}

```

```

<just>{left}|{right}|{center} {
    BEGIN 0;
    strcpy(gs[18].val,yytext);
    printf("l just=(%s)\n",yytext);
    return(y1_c_r);
}
<labln>{simple}|{arrow} {
    BEGIN 0;
    strcpy(gs[20].val,yytext);
    zzsect=1;
    zzigs=4;
    printf("l labln=(%s)\n",yytext);
    return(ys_a);
}
^{empty}{none} { gr_obj();
    ini_obj(0);
    printf("l None begin\n");
    return(ynone);
}
^{empty}{point} { gr_obj();
    ini_obj(1);
    printf("l Point begin\n");
    return(ypoint);
}
^{empty}{line} { gr_obj();
    ini_obj(2);
    printf("l Line begin\n");
    return(yline);
}
^{empty}{pline} { gr_obj();
    ini_obj(3);
    printf("l Pline begin\n");
    return(ypline);
}
^{empty}{region} { gr_obj();
    ini_obj(4);
    printf("l Region begin\n");
    return(yregion);
}
^{empty}{arc} { gr_obj();
    ini_obj(5);
    printf("l Arc begin\n");
    return(yarc);
}

```



```

    }
^{\emptyset}{text} { gr_obj();
                    ini_obj(6);
                    printf("l Text begin\n");
                    return(ytext);
    }
^{\emptyset}{rect} { gr_obj();
                    ini_obj(7);
                    printf("l Rect begin\n");
                    return(yrect);
    }
^{\emptyset}{rourect} { gr_obj();
                       ini_obj(8);
                       printf("l Roundrect begin\n");
                       return(yrourect);
    }
^{\emptyset}{ellips} { gr_obj();
                       ini_obj(9);
                       printf("l Ellips begin\n");
                       return(yellips);
    }
{mult} { zzstr= -2;
         return(ymult);
    }
{pen} { zzsect=1;
        zzigs=6;
        printf("l pen\n");
        return(ypen);
    }
{smooth} { strcpy(gs[4].val, "1");
           printf("l smooth\n");
           return(ysmooth);
    }
{brush} { zzsect=1;
          zzigs=9;
          printf("l brush\n");
          return(ybrush);
    }
{symbol} { zzsect=1;
           zzigs=7;
           printf("l symbol\n");
           return(ysymbol);
    }
}

```

```

{transf}      { BEGIN transf;
                zzihs=33;
                printf("l transf\n");
                return(ytrans);
              }
{center}      { zzsect=1;
                zzigs=4;
                printf("l center\n");
                return(ycenter);
              }
{font}        { zzsect=1;
                zzigs=12;
                printf("l font\n");
                return(yfont);
              }
{spacing}     { zzsect=1;
                zzigs=17;
                printf("l spacing\n");
                return(yspacing);
              }
{justify}     { BEGIN just;
                printf("l justify\n");
                return(yjustify);
              }
{angle}       { zzsect=1;
                zzigs=19;
                printf("l angle\n");
                return(yangle);
              }
{labelln}     { BEGIN labln;
                zzsect=1;
                zzigs=20;
                printf("l labelln\n");
                return(ylabelln);
              }
{column}      { BEGIN column;
                printf("l column\n");
                return(ycolumn);
              }
{anynum}      { int i;
                printf("l Num=(%s) obtp=%d sect=%d
                zzigs=%d zzstr=%d\n", yytext, zzobtype, zzsect, zzigs, zzstr);
                switch (zzsect)

```

```

{ case 0:
    strcpy(hs[zzihs].val,yytext);
    break;
case 1:
    if (zzobtype==1)
    { strcpy(gs[zzihs].val,yytext);
      if (zzigs==7) zzigs=8;
      else if (zzigs==8) zzigs=6;
      else if (zzigs==6)
        { if (strlen(gs[12].val)==0)
          zzigs=12;
          else zzigs=13;
        }
      else if (zzigs==13) zzigs=19;
    }
    else
    { strcpy(gs[zzihs].val,yytext);
      zzigs++;
    }
    break;
case 2:
    switch (zzobtype)
    { case 1: /* Point */
      if (zzstr==0)
fprintf(ffd, "%d%c%d%c%d%c%s", zznobj, zzrazd, zzsobj, zzrazd,
                                             zzuid, zzrazd, yytext);
        else
fprintf(ffd, "%c%s%c%c\n", zzrazd, yytext, zzrazd, zzrazd);
          zzstr++;
          break;
      case 2: /* Line */
        if (zzstr==0)
fprintf(ffd, "%d%c%d%c%d", zznobj, zzrazd, zzsobj, zzrazd, zzuid);
          fprintf(ffd, "%c%s", zzrazd, yytext);
          if (zzstr==3) fprintf(ffd, "\n");
          zzstr++;
          break;
      case 3: /* Pline */
      case 4: /* Region */
        if (zzstr== -2)
          strcpy(gs[2].val,alltrim(yytext));
        else if (zzstr== -1)
          { if (strlen(gs[zzihs].val)>0)

```

```

        strcat(gs[zzigs].val,",");
        strcat(gs[zzigs].val,yytext);
        zznid=strtol(yytext,NULL,10);
        zzsobj++;
        zzuid=1;
    }
    else
zzdt[zzstr]=strtod(yytext,NULL);
        zzstr++;
        if (zzstr==4)
        {
fprintf(ffd,"%d%c%d%c%d",zznobj,zzrazd,zzsobj,zzrazd,zzuid);
            for (i=0;i<4;i++)
fprintf(ffd,"%c%f",zzrazd,zzdt[i]);
            fprintf(ffd,"\n");
            zzdt[0]=zzdt[2];
            zzdt[1]=zzdt[3];
            zzuid++;
            if (zzuid>=zznid)
            { zzstr=-1;
              zzigs=3;
            }
            else zzstr=2;
        }
        break;
    case 5: /* Arc */
    case 6: /* Text */
    case 7: /* Rect */
    case 8: /* Roundrect */
    case 9: /* Ellipse */
        if (zzstr==0)
fprintf(ffd,"%d%c%d%c%d",zznobj,zzrazd,zzsobj,zzrazd,zzuid);
            fprintf(ffd,"%c%s",zzrazd,yytext);
            if (zzstr==3)
            { fprintf(ffd,"\n");
              if (zzobtype==5 || zzobtype==8)
              { zzsect=1;
                zzigs=4;
              }
            }
            zzstr++;
            break;
    }
}

```

```

        }
        return(ynum);
    }
    { int fl=0;
      printf(" str = (%s)\n",yytext);
      if (zzobtype==1 || zzobtype==6)
      { if (zzigs==7)
        { strcpy(gs[12].val,yytext);
          zzigs=8;
        }
        else
        { if (zzigs==21)
          { strcpy(gs[zzigs].val,"");
            fl=strlen(zztext);
            strcat(zztext,yytext);
          }
          else
          { strcpy(gs[zzigs].val,yytext);
            zzigs++;
          }
        }
      }
      else strcpy(hs[zzihs].val,yytext);
      if (!fl) return(ystr);
    }
}

[\n]*      { printf("l10\n");
            return('\n');
}

[ \t]+    { /*printf("l1\n");*/ }
[^ \t]    { printf(" %c ",yytext[0]);
            return(yytext[0]);
}

%%
char *alltrim(char *s)
{ char *s1;
  int i;
  // printf("alltrim: (%s) ",s);
  for (i=0;i<strlen(s),s[i]!=' ';i++) ;
  s1=&(s[i]);
  for (i=strlen(s);i>0,s[--i]!=' '); ;
  s[i+1]='\0';
  // printf(" (%s)\n",s1);
  return(s1);
}

```

```

}
char *name_cor(char *s)
{ int i;
  printf("name_cor: (%s) ",s);
  for (i=0;i<strlen(s);i++)
    if (!isalpha(s[i]) && !isdigit(s[i]) && s[i]!='-') s[i]='_';
  printf(" (%s)\n",s);
  return(s);
}
/*****/
void gr_obj()
{ int i,fl=0;
  char c,c1;
  if (!zznobj) return;
  fprintf(ffg,"%d",zznobj);
  printf("l %s end\n",gs[1].val);
  for (i=1;i<LGS;i++)
  { if (i==21 && zzobtype==6) fprintf(ffg,"%c%s",zzrazd,zztext);
    else
      if (gs[i].tpfld=='c' && gs[i].val[0]!='')
        fprintf(ffg,"%c\"%s\"",zzrazd,gs[i].val);
      else fprintf(ffg,"%c%s",zzrazd,gs[i].val);
      strcpy(gs[i].val,"");
  }
  c=' ';
  for (i=0;i<zznflld;i++)
  { fprintf(ffg,"%c",zzrazd);
    if (feof(fffi) || (c=fgetc(fffi))==zzdlm || c=='\n')
    { if (tpfld[i]=='c') fprintf(ffg,"\"");
      continue;
    }
    if (tpfld[i]=='c')
      if (c=='') fl=1;
      else fprintf(ffg,"");
    fprintf(ffg,"%c",c);
    while (!feof(fffi) && (c=fgetc(fffi))!='\n' && (fl || c!=zzdlm))
    { fprintf(ffg,"%c",c);
      if (fl && c=='') fl=0;
      c1=c;
    }
    if (tpfld[i]=='c' && c1!='') fprintf(ffg,"");
  }
  fprintf(ffg,"\n");
}

```

```

}
/*****/
void ini_obj(int tp)
{ zobtype=tp;
  zzsect=2;
  strcpy(gs[1].val,alltrim(yytext));
  zznobj++;
  if (tp==3 || tp==4)
  { zzsobj=0;
    zzigs=3;
    if (tp==3) zzstr= -1;
    else zzstr= -2;
  }
  else
  { zzsobj=1;
    zzuid=1;
    zzstr=0;
  }
  if (tp==6) zzigs=21;
  strcpy(zztext,"");
}

```

Синтаксический анализ

```

%token yvers ychset ydelim yuniq yindex ynums ycoords yearth
%token ynearth ylay ytable ywind yproj yaff yunit ybound
%token ytrans ycolumn ydata ynum ystr yname ytype ypoint
%token ysymbol ynone yline ypen ypline ymult ysmooth yregion
%token ybrush ycenter yarc ytext yfont yspacing yjustify
%token yl_c_r yangle ylabelln ys_a yrect yrourect yellips
%{
#include <ctype.h>
/*#include "glob.c"*/
int flgrid= -1;
extern char zzrazd;
extern FILE *ffh,*ffg,*ffd;
%}
%%
mapfile:      '\n' |
              mif

mif:          yvers ynum { printf("Vers\n"); }
              inf_lines { printf("after ~INF_LINES\n"); }

```

```

        ycolumn ynum '\n' { printf("Colum\n"); }
        col_lines { printf("after ~COL_LINES\n"); }
ydata '\n' dat_lines { printf("after ~DAT_LINES\n"); }

```

```

inf_lines:  inf_line |
            inf_lines inf_line

```

```

inf_line:   '\n' { printf("EMPTY\n"); } |
            ychset ystr '\n' { printf("CharSet\n"); } |
            ydelim ystr '\n' { printf("Delim\n"); } |
            yuniq ynums '\n' { printf("Uniq\n"); } |
            yindex ynums '\n' { printf("Index\n"); } |
            ycoords coord_lns { printf("Coords\n"); } |
            ytrans ynum ',' ynum ',' ynum ',' ynum '\n'
            { printf("Trans\n"); }

```

```

coord_lns:  yearth proj_lns aff_ln bou_ln { printf("Earth\n"); } |
            ynearth aff_ln unit_ln bou_ln { printf("NonEarth\n"); } |
            ylay ystr { printf("Lay\n"); } |
            ytable ystr { printf("Tabl\n"); } |
            ywind ynum { printf("Wind\n"); }

```

```

proj_lns:   |
            yproj ynum vk { printf("Proj1\n"); } |
            yproj ynum numlist vk { printf("Proj2\n"); }

```

```

numlist:   ',' vk |
            ',' vk ynum |
            ',' vk ystr |
            numlist ',' vk |
            numlist ',' vk ynum |
            numlist ',' vk ystr

```

```

aff_ln:    |
            yaff ystr numlist vk

```

```

bou_ln:    |
            ybound '(' ynum ',' ynum ')' '(' ynum ',' ynum
            ')' vk

```

```

unit_ln:   |
            yunit ystr vk

```



```

col_lines:  col_line |
            col_lines col_line

col_line:  yname ytype colsize '\n'

colsize:   |
            '(' ynum ')' |
            '(' ynum ',' ynum ')'

dat_lines: dat_line |
            dat_lines dat_line

dat_line:  ynone '\n' |
            ypoint ynum ynum vk sym_ln |
            yline ynum ynum ynum ynum vk pen_ln |
            ypline mult sects pen_ln smooth_ln |
            yregion ynum vk sects pen_ln brush_ln center_ln |
            yarc ynum ynum ynum ynum vk ynum ynum vk pen_ln |
            ytext vk ystr vk ynum ynum ynum ynum vk
                font_ln spac_ln just_ln angle_ln label_ln |
            yrect ynum ynum ynum ynum vk pen_ln brush_ln |
            yrourect ynum ynum ynum ynum vk ynum vk pen_ln
brush_ln  |
            yellips ynum ynum ynum ynum vk pen_ln brush_ln

sym_ln:   |
            ysymbol '(' ynum numlist ')' '\n' |
            ysymbol '(' ystr numlist ')' '\n'

pen_ln:   |
            ypen '(' pars ')' vk

pars:     ynum |
            pars ',' ynum

mult:     |
            ymult ynum '\n'

sects:    ynum vk |
            sects ynum vk

smooth_ln: |
            ysmooth vk

```

```

brush_ln:      |
               | ybrush '(' pars ')' vk
center_ln:    |
               | ycenter ynum ynum vk
font_ln:      |
               | yfont '(' ystr numlist ')' vk
spac_ln:      |
               | yspacing ynum vk
just_ln:      |
               | yjustify yl_c_r vk
angle_ln:     |
               | yangle ynum vk
label_ln:     |
               | ylabelln ys_a ynum ynum vk
vk:           |
               | '\n'

```

ГЛАВА 5. ГЕНЕРАТОР ЛЕКСИЧЕСКИХ АНАЛИЗАТОРОВ JFLEX

Генератор JFlex строит лексический анализатор, задача которого – распознать из входного потока символов очередную лексему. По содержимому входного файла, генератор JFlex строит детерминированный конечный автомат в виде программы на языке Java.

Генератор лексических анализаторов может применяться для построения различных преобразователей текстовой информации (конверторов), создания пакетных редакторов, реализации распознавателя директив в диалоговой программе и т.д.

5.1. Регулярные выражения

Регулярное выражение является шаблоном, определяющим некоторое множество последовательностей символов.

Регулярное выражение применяется как шаблон к последовательности символов текстового файла или к входному потоку символов. Фрагмент текста считается соответствующим регулярному выражению, если он входит в множество последовательностей символов, определяемых этим выражением.

Платформа Java использует шаблоны Regex, реализованные в стандартном пакете `java.util.regex`. Шаблоны Regex немного синтаксически отличаются от регулярных выражений, используемых генератором lex. Рассмотрим эти отличия.

- В Regex внутри квадратных скобок можно использовать операцию пересечения (`&&`) множеств символов, определяемых регулярным выражением.

Примеры:

`[a-z&&[def]]` определяет d, e, f

`[a-z&&[^bc]]` определяет от a до z, кроме b и c

`[a-z&&[^m-p]]` эквивалентно `[a-lq-z]`

- Отсутствует бинарная операция «/», позволяющая при распознавании задавать правый контекст.

5.2. Структура входного файла

Входной файл генератора JFlex имеет следующий формат:

```
код пользователя
%%
опции и определения
%%
лексические правила
```

Два подряд идущих символа %% являются разделителями секций и должны располагаться с первой позиции в отдельной строке.

5.2.1. Секция код пользователя

Содержимое этой секции без изменений копируется в начало выходного файла, построенного генератором JFlex.

5.2.2. Секция опции и определения

Секция может содержать:

- опции настройки генератора JFlex (JFlex директивы и Java код, включаемые в выходной текст, построенный генератором JFlex);
- список состояний лексического анализатора;
- определения имен регулярных выражений.

Каждая JFlex директив записывается в отдельной строке и начинается с символа «%». Директива может иметь один или более параметров.

Допустимо использование следующих директив:

`%class "classname"`

– Сообщает генератору JFlex имя основного класса генерируемого лексического анализатора. Сгенерированный код записывается в файл с именем "classname.java". Если директива `%class` отсутствует, то создается класс с именем "Yulex", который записывается в **файл** "Yulex.java".

`%implements "interface 1"[, "interface 2", ..]`

– Создает класс, реализующий заданный интерфейс.

`%extends "classname"`

– Создает подкласс класса "classname". Можно использовать только одну директиву `%extends`.

`%public`

– Добавляет модификатор `public` к описанию генерируемого класса.

`%final`

– Добавляет модификатор `final` к описанию генерируемого класса.

`%abstract`

– Создает абстрактный класс.

`%private`

– Создает все генерируемые методы и поля класса как `private`. Исключением является конструктор, пользовательский код в спецификации, и, если используется `%cup`, метод `next_token`.

`{`

`...`

`}`

– Код, заключённый между `{` и `}` копируется без изменений в генерируемый класс. Здесь вы можете определить свои собственные переменные и функции в генерируемом анализаторе.

```
%init{  
...  
%init}
```

– Код, заключённый между `%init{` и `%init}` копируется без изменений в конструктор генерируемого класса. Здесь переменные, описанные в директиве `{` и `}` могут быть инициализированы. Если присутствует более одной опции инициализации, то код объединяется в порядке поступления спецификаций.

```
%initthrow{  
"exception1" [, "exception2", ...]  
%initthrow}
```

или допустима запись в виде одной строки

```
%initthrow "exception1" [, "exception2", ...]
```

– Определяет исключения какого класса может выбрасывать конструктор генерируемого класса. Если исключений более одного, то все они будут добавлены к заголовку конструктора.

```
%ctorarg "type" "ident"
```

– Добавляет заданные аргументы в конструктор генерируемого анализатора. Если указано более одной такой директивы, аргументы добавляются в порядке их описания.

```
%scanerror "exception"
```

– Определяет исключения, которые будет обрабатывать генератор в случае его внутренней ошибки (по умолчанию `java.lang.Error`).

```
%buffer "size"
```

– Устанавливает размер буфера анализатора в байтах. По умолчанию размер буфера 16384 байт.

`%include "filename"`

– Вставляет в текст код из указанного файла.

`%function "name"`

– Определяет имя метода лексического анализатора. По умолчанию – `yulex`.

`%integer`

`%int`

– Каждая из этих деклараций указывает, что лексический анализатор в качестве результата будет возвращать значение типа `int`.

`%cup`

– Сообщает, что лексический анализатор будет использоваться синтаксическим анализатором.

Список состояний лексического анализатора задается следующей директивой:

`%s[tate] name1, name2 ...`

где **name1 name2 ...** – имена состояний лексического анализатора.

Примеры: Определение состояний.

```
%state STATE1
```

```
%state STATE3, XYZ, STATE_10
```

```
%state ABC STATE5
```

Определения имен регулярных выражений имеют следующий формат:

`Name = Rexp`

где **Name** – имя;

RegExp – регулярное выражение.

Имена регулярных выражений можно использовать в секции правил. В любом регулярном выражении секции правил имя регулярного выражения из секции определений будет заменяться на определяемое им регулярное выражение.

5.2.3. Секция лексические правила

В секции правил описываются лексемы, которые должен распознавать лексический анализатор, и действия, выполняемые при распознавании каждой лексемы.

Формат правила похож на описанный в генераторе lex и выглядит следующим образом:

```
<StatesList >   RegExp   Action
```

где **States** – имена состояний лексического анализатора;

RegExp – регулярное выражение;

Action – действие.

Конструкция <States> является необязательной, т. е. правило может начинаться с выражения RegExp. Имена состояний заключаются в угловые скобки и разделяются запятыми. Действие Action должно отделяться от выражения RegExp по крайней мере одним пробелом.

Выражение RegExp правила определяет лексему, которую должен распознавать лексический анализатор. Если заданы состояния States, то распознавание лексемы по данному правилу производится только тогда, когда анализатор находится в одном из указанных состояний.

Действие Action задается в виде фрагмента программного кода на языке Java и выполняется всякий раз, когда применяется данное правило, т.е. когда входная последовательность символов принадлежит множеству, определяемому регулярном выражением RegExp данного правила.

Полный синтаксис секции лексических правил в виде грамматики Бэкуса-Наура выглядит следующим образом:

```

LexicalRules ::= Rule+
Rule         ::= [StateList] ['^'] RegExp [LookAhead] Action
              | [StateList] '<<EOF>>' Action
              | StateGroup
StateGroup   ::= StateList '{' Rule+ '}'
StateList    ::= '<' Identifier (',' Identifier)* '>'
LookAhead    ::= '$' | '/' RegExp
Action       ::= '{' JavaCode '}' | '|'

RegExp       ::= RegExp '|' RegExp
              | RegExp RegExp
              | '(' RegExp ')'
              | ('!' | '~') RegExp
              | RegExp ('*' | '+' | '?')
              | RegExp "{" Number ["," Number] "}"
              | '[' ['^'] (Character|Character'-Character)* ']'
              | PredefinedClass
              | '{' Identifier '}'
              | '"' StringCharacter+ '"'
              | Character

PredefinedClass ::= '[:jletter:]'
                 | '[:jletterdigit:]'
                 | '[:letter:]'
                 | '[:digit:]'
                 | '[:uppercase:]'
                 | '[:lowercase:]'
                 | '.'

```

Кроме операторов языка Java, в действии можно использовать встроенные переменные и методы анализатора.

5.2.4. Встроенные переменные и методы

string yytex()

– возвращает фрагмент входного текста, удовлетворяющего регулярному выражению и распознанного данным правилом;

int yylength()

– возвращает количество символов фрагмент входного текста, удовлетворяющего регулярному выражению и распознанного данным правилом;

int yystate()

– возвращает текущее состояние лексического анализатора;

void yybegin(int lexicalState)

– переводит лексический анализатор в указанное состояние;

void yypushback(int number)

– возвращает во входной поток `number` символов.

5.3. Примеры входных файлов

Пример 1.

```
package Example;

import java_cup.runtime.*;
%%
%cup
%%
";"          { return new Symbol(sym.SEMI); }
```

```

"+"          { return new Symbol(sym.PLUS); }
"*"          { return new Symbol(sym.TIMES); }
"("          { return new Symbol(sym.LPAREN); }
")"          { return new Symbol(sym.RPAREN); }
"[0-9]+"     { return new Symbol(sym.NUMBER,
                                new Integer(yytext())); }
[ \t\r\n\f] { /* ignore white space. */ }
.            { System.err.println(
                "Illegal character: "+yytext()); }

```

Пример 2.

В примере строится лексический анализатор, на вход которого подается исходный текст программы на языке Java. Данный анализатор распознает некоторые синтаксические конструкции Java, а именно, некоторые ключевые слова, некоторые операторы, комментарии и два типа символов.

```

/* JFlex example: part of Java language lexer specification */
import java_cup.runtime.*;

/**
 * This class is a simple example lexer.
 */
%%
%class Lexer
%unicode
%cup
%line
%column
%{
    StringBuffer string = new StringBuffer();

    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }
    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
}%
LineTerminator = \r|\n|\r\n

```

```

InputCharacter = [^\r\n]
WhiteSpace     = {LineTerminator} | [ \t\f]

/* comments */
Comment = {TraditionalComment} | {EndOfLineComment} |
          {DocumentationComment}

TraditionalComment = "/*" [^*] ~"*/" | "/*" "*" + "/"
EndOfLineComment   = "//" {InputCharacter}* {LineTerminator}
DocumentationComment = "/*" {CommentContent} "*" + "/"
CommentContent      = ( [^*] | \*+ [^/*] ) *

Identifier = [:jletter:] [:jletterdigit:]*

DecIntegerLiteral = 0 | [1-9][0-9]*
%state STRING

%%
/* keywords */
<YYINITIAL> "abstract"      { return symbol(sym.ABSTRACT); }
<YYINITIAL> "boolean"       { return symbol(sym.BOOLEAN); }
<YYINITIAL> "break"        { return symbol(sym.BREAK); }
<YYINITIAL> {
  /* identifiers */
  {Identifier}              { return symbol(sym.IDENTIFIER); }

  /* literals */
  {DecIntegerLiteral}      { return symbol(sym.INTEGER_LITERAL); }
  \"                       { string.setLength(0); yybegin(STRING); }

  /* operators */
  "="                      { return symbol(sym.EQ); }
  "=="                     { return symbol(sym.EQEQ); }
  "+"                      { return symbol(sym.PLUS); }

  /* comments */
  {Comment}                { /* ignore */ }

  /* whitespace */
  {WhiteSpace}             { /* ignore */ }
}
<STRING> {
  \"                       { yybegin(YYINITIAL);

```

```

        return symbol(sym.STRING_LITERAL,
            string.toString());
    }
    [^\n\r\"\\]+ { string.append( yytext() ); }
    \\t          { string.append('\t'); }
    \\n          { string.append('\n'); }

    \\r          { string.append('\r'); }
    \\\"         { string.append('\\"'); }
    \\          { string.append('\\'); }
}
/* error fallback */
.|\\n          { throw new Error("Illegal character <"+
                        yytext()+">");
}

```

5.4. Использование генератора JFlex

Запуск FLex

java JFlex.Main <options> <inputfiles>

где **inputfiles** – входной файл в формате JFlex;

options – опции (см. документацию).

В результате будет сгенерирован файл лексического анализатора с именем <inputfile>.java (не забудьте добавить lib\JFlex.jar в CLASSPATH).

ГЛАВА 6. ГЕНЕРАТОР СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ CUP

Генератор синтаксических анализаторов CUP по описанию входной грамматики языка строит конечный автомат с магазинной памятью в виде программы на языке Java.

Генератор CUP обрабатывает широкий класс контекстно-свободных грамматик – LALR(1)-грамматики.

Синтаксические анализаторы, создаваемые с помощью CUP, реализуют так называемый LALR(1)-разбор, являющийся модификацией одного из основных методов разбора "снизу вверх" – LR(k)-разбора. Любой разбор по принципу "снизу вверх" (или восходящий разбор) состоит в попытке приведения всей совокупности входных данных (входной цепочки) к так называемому "начальному символу грамматики".

В каждый момент грамматического разбора анализатор находится в некотором состоянии, определяемом предысторией разбора, и в зависимости от очередной лексемы предпринимает то или иное действие для перехода к новому состоянию. Различают два типа действий: **сдвиг**, т.е. чтение следующей входной лексемы, и **свертку**, т.е. применение одного из правил подстановки для замещения нетерминалом последовательности символов, соответствующей правой части правила. Работа CUP по генерации процедуры грамматического анализа заключается в построении таблиц, которые для каждого из состояний определяют тип действий анализатора и номер следующего состояния в соответствии с каждой из входных лексем.

Пользователь CUP должен описать структуру своей входной информации (**грамматику**) как набор **правил**. Грамматические правила описываются в терминах некоторых исходных конструкций, которые называются лексическими единицами, или **лексемами**. Лексемой называется цепочка (последователь-

ность) символов, которую удобно рассматривать как единый синтаксический объект.

Набор лексем определяется разработчиком лексического анализатора. Для синтаксического анализатора все лексемы считаются терминальными символами грамматики.

Генератор CUP обеспечивает автоматическое построение лишь процедуры грамматического анализа. Однако, действия по обработке входной информации обычно должны выполняться по мере распознавания на входе тех или иных допустимых грамматических конструкций. Поэтому наряду с заданием грамматики входных текстов CUP предусматривает возможность описания для отдельных конструкций семантических процедур (**действий**) с тем, чтобы они были включены в программу грамматического разбора.

6.1. Структура входного файла

Входной файл генератора CUP имеет следующий формат:

- задание пакета и спецификации `import`
- код пользователя
- список терминальных и нетерминальных символов
- описание приоритетов
- грамматика

Секции должны следовать друг за другом в порядке, определенном структурой входного файла.

6.1.1. Секция задания пакета и спецификаций `import`

Спецификации `import` имеют тот же самый синтаксис и смысл, в котором они используются в программах на языке Java. Определение пакета должно предшествовать спецификациям `import`.

Объявление пакета показывает, в каком пакете будут сгенерированы классы `sum` и `parser`.

6.1.2. Секция код пользователя

Секция позволяет включать пользовательские переменные и методы непосредственно в код генерируемого синтаксического анализатора.

```
action code {: ... :};
```

– В файле синтаксического анализатора (`parser`) при генерации создаётся отдельный не `public` класс, который содержит все необходимое для выполнения встроенных пользовательских действий. Определение `action code` позволяет включать свой код в этот внутренний класс. Методы и переменные, которые используются в действиях грамматики, обычно помещаются именно сюда.

```
parser code {: ... :};
```

– Определение `parser code` позволяет включать свои переменные и методы в генерируемый класс `parser`. Это позволяет использовать свои методы для настройки синтаксического анализатора и/или переопределения существующих методов (например, метода обработки ошибок)

```
init with {: ... :};
```


– Здесь определяется код, который будет выполнен синтаксическим анализатором перед первым чтением лексемы. Обычно используется для инициализации сканера (лексического анализатора), такой как создание таблиц и структур данных, необходимых для выполнения семантических действий. Код будет положен в `void` метод внутри класса `parser`.

```
scan with {: ... :};
```

– Эта секция определяет, как `parser` будет запрашивать следующую лексему у сканера. Содержимое секции будет записано как тело метода, возвращающего объект типа `java_cup.runtime.Symbol`. Соответственно, код, записанный в этой секции, должен возвращать значение того же типа.

6.1.3. Секция список терминальных и нетерминальных символов

Секция используется для именованья и определения типов терминальных и нетерминальных символов, используемых в грамматике.

```
terminal classname name1, name2, ...;  
non terminal classname name1, name2, ...;  
terminal name1, name2, ...;  
или  
non terminal name1, name2, ...;
```

где *classname* определяет тип значения (не)терминала, может быть любым классом.

В качестве имён (не)терминалов нельзя использовать зарезервированные слова:

```
"code", "action", "parser", "terminal", "non", "nonterminal", "init", "scan", "with",  
"start", "precedence", "left", "right", "nonassoc", "import", "package"
```

6.1.4. Секция описания приоритетов

В данной секции задаются приоритеты и ассоциативность терминальных символов.

```
precedence left    terminal[, terminal...];  
precedence right  terminal[, terminal...];  
precedence nonassoc terminal[, terminal...];
```

где *terminal* – имя терминального символа.

Семантика приоритетов и ассоциативности имеет тот же самый смысл, что и в генераторе уасс. Пример:

```
precedence left  ADD, SUBTRACT;  
precedence left  TIMES, DIVIDE;
```

6.1.5. Секция грамматики

В данной секции определяется аксиома грамматики и набор грамматических правил, определяющий все конструкции языка, которые должен разбирать синтаксический анализатор, и действия, выполняемые при грамматическом разборе.

Аксиома грамматики определяется директивой:

```
start with name
```

где *name* – имя нетерминального символа

В случае отсутствия директивы *start*, аксиомой грамматики считается нетерминал, указанный в левой части первого правила секции грамматики.

Синтаксически каждое правило имеет следующий формат:

```
non-terminal ::= tnlst { : action : }
```

где **non-terminal** – имя нетерминального символа; **tnlist** – список терминальных и нетерминальных символов, разделенных пробелами; **action** – действие, выполняемое при распознавании данного правила.

Каждый элемент списка **tnlist** имеет следующий формат:

tn

или

tn:name

где **tn** – имя терминального или нетерминального символа;

name – имя переменной, значение которой можно использовать в действии данного правила. Значение этой переменной играет роль значения соответствующей компоненты (данного терминального или нетерминального символа) правой части правила. Результирующее значение правила помещается во встроенную переменную **RESULT**.

В описании грамматики можно использовать встроенный нетерминальный символ **error**, который имеет тот же смысл, что и в генераторе yacc.

Пример:

```
stmt ::= expr SEMI | while_stmt SEMI | if_stmt SEMI | ... |  
      error SEMI ;
```

Пример калькулятора, реализующего операции сложения и умножения:

```
package Example;  
import java_cup.runtime.*;  
parser code {:  
    public static void main(String args[]) throws Exception {  
        new parser(new Yylex(System.in)).parse();  
    }  
:}  
terminal SEMI, PLUS, TIMES, LPAREN, RPAREN;  
terminal Integer NUMBER;
```

```

non terminal expr_list, expr_part;
non terminal Integer expr;
precedence left PLUS;
precedence left TIMES;

expr_list ::= expr_list expr_part | expr_part;
expr_part ::= expr:e { : System.out.println(" = "+e+"); : }
SEMI;
expr      ::= NUMBER:n
           { : RESULT=n; : }
           | expr:l PLUS expr:r
           { : RESULT=new Integer(l.intValue() + r.intValue()); : }
           | expr:l TIMES expr:r
           { : RESULT=new Integer(l.intValue() * r.intValue()); : }
           | LPAREN expr:e RPAREN
           { : RESULT=e; : }
           ;

```

6.2. Пример входного файла

Ниже приведен полный входной файл для CUP, реализующий калькулятор:

```

// CUP specification for a simple expression evaluator
// (no actions)

import java_cup.runtime.*;

/* Preliminaries to set up and use the scanner. */
init with { : scanner.init(); : };
scan with { : return scanner.next_token(); : };

/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER;

/* Non terminals */
non terminal expr_list, expr_part;
non terminal Integer expr, term, factor;

/* Precedences */
precedence left PLUS, MINUS;

```

```
precedence left TIMES, DIVIDE, MOD;
precedence left UMINUS;
```

```
/* The grammar */
expr_list ::= expr_list expr_part |
           expr_part;
expr_part ::= expr SEMI;
expr      ::= expr PLUS expr
           | expr MINUS expr
           | expr TIMES expr
           | expr DIVIDE expr
           | expr MOD expr
           | MINUS expr %prec UMINUS
           | LPAREN expr RPAREN
           | NUMBER
           ;
```

ГЛАВА 7. СОВМЕШНОЕ ИСПОЛЬЗОВАНИЕ JFLEX И JAVA CUP

Запуск JavaCup

java -jar java-cup-11a.jar options inputfile

inputfile – входной файл в формате JavaCup;

options – опции (см. документацию).

В результате будет сгенерирован файл с именем `parser.java` и файл таблицы терминальных и не терминальных символов `sym.java` (не забудьте добавить классы JavaCup в CLASSPATH).

Пример bat-файла сборки и запуска синтаксического анализатора.

```
set JCUP_HOME=C:\JavaCup
set JFLEX_HOME=C:\JFlex
java -cp %JCUP_HOME%;%JFLEX_HOME% JFlex.Main minimal.lex
copy minimal.lex.java Yylex.java
java -cp %JCUP_HOME% java_cup.Main minimal.cup
rem компиляция
javac -cp %JCUP_HOME%;%JFLEX_HOME% -d . parser.java sym.java Yylex.java
rem запуск
java -cp %JCUP_HOME%;%JFLEX_HOME%;. Example.parser
```

minimal.lex – входной файл JFlex;

minimal.lex.java – выходной файл JFlex (для удобства затем копируется в файл с именем `Yylex.java`);

minimal.cup – входной файл JavaCup;

parser.java sym.java – выходные файлы JavaCup;

Example.parser – тестовый пример.

8.1. MiniJava – игрушечный язык

MiniJava – это маленький искусственный игрушечный язык, приведённый здесь для иллюстрации основных шагов создания компилятора с помощью CUP. Для языка имеется спецификация через грамматику ("MiniJava"). По сути,

он знает только ветвления, циклы, определения и базовый ввод/вывод. Целью является игрушечный транслятор ("MiniJVM"), для которого имеется симулятор (в данном примере не рассматривается). Типичный пример входного файла на этом языке выглядит следующим образом:

```
int x, y;
y = 10;
x = read();
write(x);
if (x>=0)
    write (y);
else
    write (x);
while (1-y <= -x) {
    y = x * (-y + 5);
}
```

Типичное использование дерева синтаксического разбора заключается в генерации кода. Это можно сделать вручную, создав классы синтаксического дерева в разделе действий для каждого альтернативного правила и разработав механизм прохождения этого дерева, например, с вызовами рекурсивных функций или шаблонами посещения.

Однако CUP даёт возможность автоматической генерации XML документа для дерева разбора, который затем может быть обработан XSLT/XQuery процессором, даже для генерации целевого кода (полный пример приведен в документации Java CUP)

Автоматический разбор напрямую в XML

В примере приведено прямое преобразование грамматики MiniJava в спецификации CUP-парсера. Java-парсер генерируется с помощью команды:

```
java -jar java-cup-11b.jar -locations -interface -parser Parser -xmlactions minijava.cup
```

```

/* Minijava Grammar */
import java_cup.runtime.ComplexSymbolFactory;
import java_cup.runtime.ScannerBuffer;
import java_cup.runtime.XMLElement;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;
import java.io.*;

import javax.xml.transform.*;
import javax.xml.transform.stream.*;
parser code {
    public Parser(Lexer lex, ComplexSymbolFactory sf) {
        super(lex,sf);
    }
    public static void main(String[] args) throws Exception {
        // initialize the symbol factory
        ComplexSymbolFactory csf = new ComplexSymbolFactory();
        // create a buffering scanner wrapper
        ScannerBuffer lexer = new ScannerBuffer(new Lexer(
            new BufferedReader(new FileReader(args[0])),csf));
        // start parsing
        Parser p = new Parser(lexer,csf);
        XMLElement e = (XMLElement)p.parse().value;
        // create XML output file
        XMLOutputFactory outFactory =
            XMLOutputFactory.newInstance();
        XMLStreamWriter sw = outFactory.createXMLStreamWriter(
            new FileOutputStream(args[1]));
        // dump XML output to the file
        XMLElement.dump(lexer,sw,e,"expr","stmt");

// transform the parse tree into an AST and a rendered HTML version
        Transformer transformer=TransformerFactory.newInstance()
            .newTransformer(new StreamSource(
                new File("tree.xsl")));
        Source text = new StreamSource(new File(args[1]));
        transformer.transform(text, new StreamResult(
            new File("output.xml")));
        transformer = TransformerFactory.newInstance()
            .newTransformer(new StreamSource(
                new File("tree-view.xsl")));
        text = new StreamSource(new File("output.xml"));
        transformer.transform(text, new StreamResult(

```



```

new File("ast.html"));
}
:});

terminal SEMICOLON, COMMA, LPAR, RPAR, BEGIN, END, IF, ELSE
terminal WHILE, READ, WRITE, BUNOP, ASSIGN;
terminal TYPE, BINOP, UNOP, COMP, BBINOP, INTCONST;
terminal IDENT, STRINGCONST;
terminal BOOLCONST;

non terminal program, decllist, decl, stmtlist, identlist, stmt, expr, cond;

precedence left ELSE, UNOP, BINOP, BUNOP, BBINOP;

program ::= decllist:d stmtlist:s
;
decllist ::= decl:d decllist:d1
| /* empty */
;
stmtlist ::= stmtlist:s1 stmt:s
| /* empty */
;
decl ::= TYPE IDENT:identifier identlist:il SEMICOLON
;
identlist ::= identlist:il COMMA IDENT:identifier
| /* empty */
;
stmt ::= SEMICOLON
| BEGIN stmtlist:s1 END
| IDENT:lhs ASSIGN expr:rhs SEMICOLON
| IDENT:lhs ASSIGN READ LPAR RPAR SEMICOLON
| IDENT:lhs ASSIGN READ LPAR STRINGCONST:s RPAR SEMICOLON
| WRITE LPAR expr:e RPAR SEMICOLON
| WRITE LPAR STRINGCONST:s RPAR SEMICOLON
| IF LPAR cond:c RPAR stmt:s
| IF LPAR cond:c RPAR stmt:t ELSE stmt:e
| WHILE LPAR cond:c RPAR stmt:s
;
cond ::= BOOLCONST:c
| LPAR cond:c RPAR
| expr:e1 COMP:op expr:e2
| BUNOP cond:c
| cond:c1 BBINOP:op cond:c2

```

```

;
expr ::= IDENT:identifier
      | INTCONST:constant
      | LPAR expr:e RPAR
      | BINOP expr:e
      | expr:e1 BINOP:op expr:e2
;

```

Сгенерированный сканер

Вместо утомительного создания сканера вручную, воспользуемся генератором сканеров JFlex. Для MiniJava спецификация сканера будет выглядеть так:

```

// Technische Universitaet Muenchen
// Fakultaet fuer Informatik
// Michael Petter

import java_cup.runtime.Symbol;
import java_cup.runtime.ComplexSymbolFactory;
import java_cup.runtime.ComplexSymbolFactory.Location;

%%
%public
%class Lexer
%cup
%implements sym, minijava.Constants
%char
%line
%column

%{
    StringBuffer string = new StringBuffer();
    public Lexer(java.io.Reader in, ComplexSymbolFactory sf){
        this(in);
        symbolFactory = sf;
    }
    ComplexSymbolFactory symbolFactory;

    private Symbol symbol(String name, int sym) {
        return symbolFactory.newSymbol(name, sym,
            new Location(yyline+1,yycolumn+1,yychar),
            new Location(yyline+1,yycolumn+yylength(),
                yychar+yylength()));
    }
}

```

```

}

private Symbol symbol(String name, int sym, Object val) {
    Location left = new Location(yyline+1,yycolumn+1,yychar);
    Location right= new Location(yyline+1,yycolumn+yylength(),
                                yychar+yylength());
    return symbolFactory.newSymbol(name,sym,left,right,val);
}

private Symbol symbol(String name,int sym,Object val,int buflen) {
    Location left = new Location(yyline+1,yycolumn+yylength()-
                                buflen,yychar+yylength()-buflen);
    Location right= new Location(yyline+1,yycolumn+yylength(),
                                yychar+yylength());
    return symbolFactory.newSymbol(name, sym, left, right,val);
}

private void error(String message) {
    System.out.println("Error at line "+(yyline+1)+
                       ", column "+(yycolumn+1)+" : "+message);
}
%}

%eofval{
    return symbolFactory.newSymbol("EOF", EOF,
                                   new Location(yyline+1,yycolumn+1,yychar),
                                   new Location(yyline+1,yycolumn+1,yychar+1));
%eofval}

Ident = [a-zA-Z$_] [a-zA-Z0-9$_]*
IntLiteral = 0 | [1-9][0-9]*
BoolLiteral = true | false
new_line = \r|\n|\r\n;
white_space = {new_line} | [ \t\f]

%state STRING

%%

<YYINITIAL>{
/* keywords */
"int"      { return symbol("int",TYPE,new Integer( INTTYPE )); }
"if"       { return symbol("if",IF); }
"else"     { return symbol("else",ELSE); }
"while"    { return symbol("while",WHILE); }

```

```

"read"          { return symbol("read",READ); }
"write"         { return symbol("write",WRITE); }

/* names */
{Ident}         { return symbol("Identifier",IDENT, yytext()); }

/* bool literal */
{BoolLiteral}  { return symbol("Boolconst",BOOLCONST,
                             new Boolean(Boolean.parseBool(yytext()))); }

/* literals */
{IntLiteral}   { return symbol("Intconst",INTCONST,
                             new Integer(Integer.parseInt(yytext()))); }

/* separators */
\"              { string.setLength(0); yybegin(STRING); }
";"            { return symbol("semicolon",SEMICOLON); }
","            { return symbol("comma",COMMA); }
"("            { return symbol("(",LPAR); }
")"            { return symbol(")",RPAR); }
"{"            { return symbol("{",BEGIN); }
"}"            { return symbol("}",END); }
"="            { return symbol("=",ASSIGN); }
"+"            { return symbol("plus",BINOP,new Integer( PLUS )); }
"-"            { return symbol("minus",BINOP,new Integer( MINUS )); }
"*"            { return symbol("mult",BINOP, new Integer( MULT )); }
"/"            { return symbol("div",BINOP, new Integer( DIV  )); }
%"            { return symbol("mod",BINOP, new Integer( MOD  )); }
"<="           { return symbol("leq",COMP, new Integer( LEQ  )); }
">="           { return symbol("gtq",COMP, new Integer( GTQ  )); }
"=="           { return symbol("eq",COMP, new Integer( EQ   )); }
"!="           { return symbol("neq",COMP, new Integer( NEQ  )); }
"<"            { return symbol("le",COMP, new Integer( LE   )); }
">"            { return symbol("gt",COMP, new Integer( GT   )); }
"&&"           { return symbol("and",BBINOP,new Integer( AND  )); }
"||"           { return symbol("or",BBINOP,new Integer( OR   )); }
"!"            { return symbol("not",BUNOP); }

{white_space}  { /* ignore */ }

}

<STRING> {

```

```

\"          { yybegin(YYINITIAL);
              return symbol("StringConst",STRINGCONST,
                          string.toString(),string.length()); }
[^\n\r\"\\]+ { string.append( yytext() ); }
\\t        { string.append('\t'); }
\\n        { string.append('\n'); }
\\r        { string.append('\r'); }
\\\"       { string.append('\\"'); }
\\         { string.append('\\'); }
}

/* error fallback */
.|\\n      { error("Illegal character <"+ yytext()+>"); }

```

8.2. Калькулятор

Пример показывает основные принципы начала работы с CUP. Минимальная спецификация CUP выглядит подобным образом:

```

/* Simple +/-/* expression language; parser evaluates constant
expressions on the fly*/
import java_cup.runtime.*;

parser code {:
    // Connect this parser to a scanner!
    scanner s;
    Parser(scanner s){ this.s=s; }
:}

/* define how to connect to the scanner! */
init with {: s.init(); :};
scan with {: return s.next_token(); :};

/* Terminals (tokens returned by the scanner). */
terminal SEMI, PLUS, MINUS, TIMES, UMINUS, LPAREN, RPAREN;
terminal Integer NUMBER; // our scanner provides numbers as integers

/* Non terminals */
non terminal      expr_list;

```

```

non terminal Integer  expr; // used to store evaluated subexpressions

/* Precedences */
precedence left PLUS, MINUS;
precedence left TIMES;
precedence left UMINUS;

/* The grammar rules */
expr_list ::= expr_list expr:e SEMI  {: System.out.println(e);:}
           | expr:e SEMI             {: System.out.println(e);:}
;

expr      ::= expr:e1 PLUS  expr:e2  {: RESULT = e1+e2;      :}
           | expr:e1 MINUS expr:e2  {: RESULT = e1-e2;      :}
           | expr:e1 TIMES expr:e2  {: RESULT = e1*e2;      :}
           | MINUS expr:e          {: RESULT = -e;         :}
           | %prec UMINUS
           | LPAREN expr:e RPAREN  {: RESULT = e;         :}
           | NUMBER:n             {: RESULT = n;          :}
;

```

Основываясь на этом файле, вам нужно просто создать по нему Java-спецификацию с помощью команды:

```
java -jar java-cup-11b.jar -interface -parser Parser calc.cup
```

Кроме того, вам нужен простой сканер для создания токенов, которые будут обрабатываться вашим парсером. Далее показано, как вручную написать такой сканер, читающий символы из входного потока:

```

import java_cup.runtime.*;

class Main {
    public static void main(String[] argv) throws Exception{
        System.out.println("Please type your arithmetic expression:");
        Parser p = new Parser(new scanner());
        p.parse();
    }
}

public class scanner {

```

```

/* single lookahead character */
protected static int next_char;

/* we use a SymbolFactory to generate Symbols */
private SymbolFactory sf = new DefaultSymbolFactory();

/* advance input by one character */
protected static void advance() throws java.io.IOException
{ next_char = System.in.read(); }

/* initialize the scanner */
public static void init() throws java.io.IOException
{ advance(); }

/* recognize and return the next complete token */
public Symbol next_token() throws java.io.IOException
{
for (;;)
    switch (next_char)
    {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            /* parse a decimal integer */
            int i_val = 0;
            do {
                i_val = i_val * 10 + (next_char - '0');
                advance();
            } while (next_char >= '0' && next_char <= '9');
            return sf.newSymbol("NUMBER",sym.NUMBER,
                                new Integer(i_val));

        case ';': advance(); return sf.newSymbol("SEMI",sym.SEMI);
        case '+': advance(); return sf.newSymbol("PLUS",sym.PLUS);
        case '-': advance(); return sf.newSymbol("MINUS",sym.MINUS);
        case '*': advance(); return sf.newSymbol("TIMES",sym.TIMES);
        case '(': advance(); return
                                sf.newSymbol("LPAREN",sym.LPAREN);
        case ')': advance(); return
                                sf.newSymbol("RPAREN",sym.RPAREN);

        case -1: return sf.newSymbol("EOF",sym.EOF);

        default:

```

```

/* in this simple scanner we just ignore everything else */
        advance();
        break;
    }
}
};

```

Для создания калькулятора скомпилируйте все классы командой:

```
javac -cp java-cup-11b-runtime.jar:. *.java
```

Запустите калькулятор:

```
java -cp ../../dist/java-cup-11b-runtime.jar:. Main
```

8.3. Разбор C

Разбор C представляет некоторую сложность, т.к. его грамматика контекстно-зависима. Причина этого в заложенном в C механизме определения ключевых слов типов внутри программы через оператор `typedef`. Однако, с помощью нескольких строк пользовательского кода мы можем во время синтаксического разбора возвращать сообщения о введении ключевых слов типа обратно в сканер, сохраняя при этом отслеживание областей видимости внутри парсера.

```

import java.io.*;
import java.util.*;
import java_cup.runtime.*;
import java_cup.runtime.XMLElement.*;
import javax.xml.stream.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;

parser code {
    public void syntax_error(Symbol cur_token){
        System.err.println("Syntax error at "+cur_token);
    }
}

```



```

public static void newScope(){
    typenames.push(new HashSet<String>());
}
public static void deleteScope(){
    typenames.pop();
}
public static boolean lookupType(String name){
    for (HashSet<String> scope: typenames)
        if (scope.contains(name)) return true;
    return false;
}
public static void addType(String name){
    typenames.peek().add(name);
}
public static LinkedList<HashSet<String>> typenames =
    new LinkedList<HashSet<String>>();
public Parser(Lexer lex, ComplexSymbolFactory sf) {
    super(lex,sf);
}

public static void main(String args[]) {
    try {
        ComplexSymbolFactory csf =
            new ComplexSymbolFactory();
        // create a buffering scanner wrapper
        ScannerBuffer lexer = new ScannerBuffer(
new Lexer(new BufferedReader(new FileReader(args[0])),csf));
        // start parsing
        Parser p = new Parser(lexer,csf);
        System.out.println("Parser runs: ");
        newScope();
        XMLElement e = (XMLElement)p.parse().value;
        // create XML output file
        XMLOutputFactory outFactory =
            XMLOutputFactory.newInstance();
        XMLStreamWriter sw = outFactory.createXMLStreamWriter(
            new FileOutputStream(args[1]));
        // dump XML output to the file
        XMLElement.dump(lexer,sw,e); //,"expr","stmt");
// transform the parse tree into an AST and a rendered HTML version
        Transformer transformer = TransformerFactory.newInstance()
            .newTransformer(new StreamSource(new File("tree.xsl")));
        Source text = new StreamSource(new File(args[1]));
    }
}

```

```

        transformer.transform(text, new StreamResult(
                                new File("output.html")));
        System.out.println("Parsing finished!");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
};

```

```

terminal IDENTIFIER, CONSTANT, STRING_LITERAL, SIZEOF, PTR_OP, INC_OP,
    DEC_OP, LEFT_OP, RIGHT_OP, LE_OP, GE_OP,
    AND_OP, OR_OP, MUL_ASSIGN, DIV_ASSIGN, MOD_ASSIGN, ADD_ASSIGN,
    SUB_ASSIGN, LEFT_ASSIGN, RIGHT_ASSIGN,
    XOR_ASSIGN, OR_ASSIGN, TYPE_NAME, TYPEDEF, EXTERN, STATIC, AUTO,
    REGISTER, CHAR, SHORT, INT, LONG, SIGNED,
    UNSIGNED, FLOAT, DOUBLE, CONST, VOLATILE, VOID, STRUCT, UNION,
    ENUM, ELLIPSIS, CASE, DEFAULT, IF, ELSE,
    SWITCH, WHILE, DO, FOR, GOTO, CONTINUE, BREAK, RETURN, SEMI,
    CURLYL, CURLYR, COMMA, COLON, ASSIGN, PARAL,
    PARAR, SQUARED_L, SQUARED_R, POINT, ADDRESS, NOT, TILDE, AND_ASSIGN,
    EQ_OP, NE_OP, MINUS, PLUS, MUL, DIVIDE,
    MODULUS, LESS, GREATER, XOR, OR, COND;

```

```

non terminal translation_unit, primary_expression, postfix_expression,
    expression, assignment_expression;
non terminal unary_operator, type_name, cast_expression,
    multiplicative_expression, additive_expression;
non terminal shift_expression, equality_expression, and_expression,
    exclusive_or_expression;
non terminal logical_and_expression, logical_or_expression,
    conditional_expression, constant_expression;
non terminal declaration_specifiers, init_declarator_list,
    storage_class_specifier, type_specifier, type_qualifier;
non terminal init_declarator, declarator, struct_or_union_specifier,
    struct_declaration_list, struct_declaration;
non terminal initializer, specifier_qualifier_list,
    struct_declarator_list, struct_declarator, enum_specifier;
non terminal enumerator_list, enumerator, pointer, direct_declarator,
    parameter_type_list, identifier_list;
non terminal type_qualifier_list, parameter_declaration,
    abstract_declarator, direct_abstract_declarator;
non terminal initializer_list, statement, labeled_statement,
    compound_statement, selection_statement;

```

```

non terminal jump_statement, statement_list, expression_statement,
    external_declaration, function_definition;
non terminal assignment_operator, parameter_list, unary_expression,
    iteration_statement, declaration_list;
non terminal relational_expression, inclusive_or_expression,
    declaration;

```

```

precedence nonassoc ELSE;

```

```

start with translation_unit;

```

```

primary_expression ::= IDENTIFIER:ident
    | CONSTANT:constant
    | STRING_LITERAL:stringliteral
    | PARAL expression:e PARAR
    ;

```

```

postfix_expression ::= primary_expression:pe
    | postfix_expression:pe
        SQUAREDL expression:index SQUAREDRL
    | postfix_expression:pe PARAL PARAR
    | postfix_expression:pe PARAL expression:e PARAR
    | postfix_expression:pe POINT IDENTIFIER:id
    | postfix_expression:pe PTR_OP IDENTIFIER:id
    | postfix_expression:pe INC_OP:op
    | postfix_expression:pe DEC_OP:op
    ;

```

```

unary_expression ::= postfix_expression:pe
    | INC_OP:op unary_expression:ue
    | DEC_OP:op unary_expression:ue
    | unary_operator:uo cast_expression:ce
    | SIZEOF unary_expression:ue
    | SIZEOF PARAL type_name:tn PARAR
    ;

```

```

unary_operator ::=ADRESS
    | MUL:op
    | PLUS:op
    | MINUS:op
    | TILDE
    | NOT:op
    ;

```

```

cast_expression
    ::= unary_expression:ue
    | PARAL type_name:tn PARAR cast_expression:ce
    ;

multiplicative_expression ::= cast_expression:ce
    | multiplicative_expression:me MUL:op cast_expression:ce
    | multiplicative_expression:me DIVIDE:op cast_expression:ce
    | multiplicative_expression:me MODULUS:op cast_expression:ce
    ;

additive_expression ::= multiplicative_expression:me
    | additive_expression:ae PLUS:op multiplicative_expression:me
    | additive_expression:ae MINUS:op multiplicative_expression:me
    ;

shift_expression ::= additive_expression:ae
    | shift_expression:se LEFT_OP additive_expression:ae
    | shift_expression:se RIGHT_OP additive_expression:ae
    ;

relational_expression ::= shift_expression:se
    | relational_expression:re LESS:op shift_expression:se
    | relational_expression:re GREATER:op shift_expression:se
    | relational_expression:re LE_OP:op shift_expression:se
    | relational_expression:re GE_OP:op shift_expression:se
    ;

equality_expression ::= relational_expression:re
    | equality_expression:ee EQ_OP:op relational_expression:re
    | equality_expression:ee NE_OP:op relational_expression:re
    ;

and_expression ::= equality_expression:ee
    | and_expression:ae ADRESS equality_expression:ee
    ;

exclusive_or_expression ::= and_expression:ae
    | exclusive_or_expression:eo XOR and_expression:ae
    ;

inclusive_or_expression ::= exclusive_or_expression:eo

```

```

    | inclusive_or_expression:ioe OR exclusive_or_expression:oe
    ;

logical_and_expression ::= inclusive_or_expression:ioe
    | logical_and_expression:lae AND_OP:op inclusive_or_expression:ioe
    ;

logical_or_expression ::= logical_and_expression:lae
    | logical_or_expression:loe OR_OP:op logical_and_expression:lae
    ;

conditional_expression ::= logical_or_expression:loe
    | logical_or_expression:loe COND expression:e COLON
                                conditional_expression:ce
    ;

assignment_expression ::= conditional_expression:ce
    | unary_expression:ue assignment_operator:aop
                                assignment_expression:ae
    ;

assignment_operator ::=ASSIGN
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression ::= assignment_expression:ae
    | expression:e COMMA assignment_expression:ae
    ;

constant_expression ::=conditional_expression:ce
    ;

declaration ::=declaration_specifiers:ds SEMI
    | declaration_specifiers:ds init_declarator_list:idl {:
```

```

if (ds.toString().indexOf(">typedef<")>0) {
    for (XMLElement e: ((XMLElement)idl).selectById("identifier"))
        Parser.addType(((Terminal)e).value().toString());
    }
:} SEMI
;

```

```

declaration_specifiers ::=storage_class_specifier:scc
    | storage_class_specifier:scc declaration_specifiers:ds
    | type_specifier:ts
    | type_specifier:ts declaration_specifiers:ds
    | type_qualifier:tq
    | type_qualifier:tq declaration_specifiers:ds
;

```

```

init_declarator_list ::=init_declarator:id
    | init_declarator_list:id1 COMMA init_declarator:id
;

```

```

init_declarator ::=declarator:d
    | declarator:d ASSIGN initializer:i
;

```

```

storage_class_specifier ::= TYPEDEF:id
    | EXTERN:id
    | STATIC:id
    | AUTO:id
    | REGISTER:id
;

```

```

type_specifier ::=VOID:type
    | CHAR:type
    | SHORT:type
    | INT:type
    | LONG:type
    | FLOAT:type
    | DOUBLE:type
    | SIGNED:type
    | UNSIGNED:type
    | struct_or_union_specifier:su
    | enum_specifier:es
    | TYPE_NAME:type
;

```

```

struct_or_union_specifier ::= STRUCT:s IDENTIFIER:id CURLYL
struct_declaration_list:sd1 CURLYR
    | STRUCT:s CURLYL struct_declaration_list:sd1 CURLYR
    | STRUCT:s IDENTIFIER:id
    | UNION:u IDENTIFIER:id CURLYL
struct_declaration_list:sd1 CURLYR
    | UNION:u CURLYL struct_declaration_list:sd1 CURLYR
    | UNION:u IDENTIFIER:id
;

```

```

struct_declaration_list ::=struct_declaration:s
    | struct_declaration_list:s1 struct_declaration:s
;

```

```

struct_declaration ::=specifier_qualifier_list:sq
struct_declarator_list:sd SEMI
;

```

```

specifier_qualifier_list ::=type_specifier:ts specifi-
er_qualifier_list:sq
    | type_specifier:ts
    | type_qualifier:tq specifier_qualifier_list:sq
    | type_qualifier:tq
;

```

```

struct_declarator_list ::=struct_declarator:s
    | struct_declarator_list:s1 COMMA struct_declarator:s
;

```

```

struct_declarator ::=declarator:d
    | COLON constant_expression:ce
    | declarator:d COLON constant_expression:ce
;

```

```

enum_specifier ::=ENUM CURLYL enumerator_list:e1 CURLYR
    | ENUM IDENTIFIER:id CURLYL enumerator_list:e1 CURLYR
    | ENUM IDENTIFIER:id
;

```

```

enumerator_list ::=enumerator:e
    | enumerator_list:e1 COMMA enumerator:e
;

```

```

enumerator ::=IDENTIFIER:id
            | IDENTIFIER:id ASSIGN constant_expression:ce
            ;

type_qualifier ::=CONST:id
               | VOLATILE:id
               ;

declarator ::=pointer:p direct_declarator:direct
           | direct_declarator:direct
           ;

direct_declarator ::=IDENTIFIER:identifier
                  | PARAL declarator:d PARAR
                  | direct_declarator:dd SQUAREDL constant_expression:ce
SQUAREDRL
                  | direct_declarator:dd SQUAREDL SQUAREDRL
                  | direct_declarator:dd PARAL parameter_type_list:ptl
PARAR
                  | direct_declarator:dd PARAL identifier_list:il PARAR
                  | direct_declarator:dd PARAL PARAR
                  ;

pointer      ::=MUL:id
            | MUL:id type_qualifier_list:tql
            | MUL:id pointer:p
            | MUL:id type_qualifier_list:tql pointer:p
            ;

type_qualifier_list ::=type_qualifier:tq
                    | type_qualifier_list:tql type_qualifier:tq
                    ;

parameter_type_list ::=parameter_list:pl
                    | parameter_list:pl COMMA ELLIPSIS:id
                    ;

parameter_list ::=parameter_declaration:pd
               | parameter_list:pl COMMA parameter_declaration:pd
               ;

```



```

parameter_declaration ::=declaration_specifiers:ds declara-
tor:d
    | declaration_specifiers:ds abstract_declarator:ad
    | declaration_specifiers:ds
    ;

identifier_list ::=IDENTIFIER:id
    | identifier_list:idl COMMA IDENTIFIER:id
    ;

type_name ::=specifier_qualifier_list:sl
    | specifier_qualifier_list:sl abstract_declarator:ad
    ;

abstract_declarator ::=pointer:p
    | direct_abstract_declarator:dad
    | pointer:p direct_abstract_declarator:d
    ;

direct_abstract_declarator ::=PARAL:id abstract_declarator:ad PARAR
    | SQUAREDL:id SQUAREDRL
    | SQUAREDL:id constant_expression:ce SQUAREDRL
    | direct_abstract_declarator:dad SQUAREDL:id SQUAREDRL
    | direct_abstract_declarator:dad SQUAREDL:id
        constant_expression:ce SQUAREDRL
    | PARAL:id PARAR
    | PARAL:id parameter_type_list:ptl PARAR
    | direct_abstract_declarator:dad PARAL:id PARAR
    | direct_abstract_declarator:dad PARAL:id
        parameter_type_list:ptl PARAR
    ;

initializer ::=assignment_expression:ae
    | CURLYL initializer_list:il CURLYR
    | CURLYL initializer_list:il COMMA CURLYR
    ;

initializer_list ::=initializer:i
    | initializer_list:il COMMA initializer:i
    ;

statement ::=labeled_statement:ls
    | {: Parser.newScope(); :} compound_statement:cs

```

```

                                {: Parser.deleteScope(); :}
    | expression_statement:es
    | selection_statement:ss
    | iteration_statement:is
    | jump_statement:js
    ;

labeled_statement ::=IDENTIFIER:id COLON statement:s
    | CASE constant_expression:ce COLON statement:s
    | DEFAULT COLON statement:s
    ;

compound_statement ::=CURLYL CURLYR
    | CURLYL statement_list:s1 CURLYR
    | CURLYL declaration_list:d1 CURLYR
    | CURLYL declaration_list:d1 statement_list:s1 CURLYR
    ;

declaration_list ::=declaration:d
    | declaration_list:d1 declaration:d
    ;

statement_list ::=statement:s
    | statement_list:s1 statement:s
    ;

expression_statement ::=SEMI
    | expression:e SEMI
    ;

selection_statement ::=IF PARAL expression:e PARAR statement:s
    | IF PARAL expression:e PARAR statement:s1 ELSE statement:s2
    | SWITCH PARAL expression:e PARAR statement:s
    ;

iteration_statement ::=WHILE PARAL expression:e PARAR statement:s
    | DO statement:s WHILE PARAL expression:e PARAR SEMI
    | FOR PARAL expression_statement:es1 expression_statement:es2
      PARAR statement:s
    | FOR PARAL expression_statement:es1 expression_statement:es2
      expression:e PARAR statement:s
    ;

```

```

jump_statement ::=GOTO IDENTIFIER:id SEMI
    | CONTINUE SEMI
    | BREAK SEMI
    | RETURN SEMI
    | RETURN expression:e SEMI
    ;

translation_unit ::=external_declaration:ed
    | translation_unit:tu external_declaration:ed
    ;

external_declaration ::=function_definition:fd
    | declaration:d
    ;

function_definition ::=declaration_specifiers:ds
    declarator:d declaration_list:dl {: Parser.newScope(); :}
    compound_statement:cs  {: Parser.deleteScope(); :}
    | declaration_specifiers:ds
    declarator:d {: Parser.newScope(); :}
    compound_statement:cs  {: Parser.deleteScope(); :}
    | declarator:d declaration_list:dl {: Parser.newScope(); :}
    compound_statement:cs  {: Parser.deleteScope(); :}
    | declarator:d          {: Parser.newScope(); :}
    compound_statement:cs  {: Parser.deleteScope(); :}
    ;

```

ГЛАВА 8. ГЕНЕРАТОР АНАЛИЗАТОРОВ PLY

Генератор лексических и синтаксических анализаторов PLY (Python-Lex-Yacc) – это совместный порт генераторов LEX и YACC на язык Python организованный в виде пакета. Данный пакет содержит два основных модуля: `lex.py` и `yacc.py`.

Модуль `lex.py` используется для задачи разбора входного потока символов и распознавания лексических единиц (лексем) при помощи детерминированного конечного автомата, то есть для задачи токенизации. Правила разбора лексем задаются пользователем в виде регулярных выражений. К правилам разбора есть возможность добавлять программные действия, осуществляемые при успешном разборе очередной лексемы.

Модуль `yacc.py` предназначен для описания синтаксического анализатора, обрабатывающего класс контекстно-свободных грамматик LALR(1), то есть поддерживающих восходящий разбор.

В каждый момент грамматического разбора анализатор находится в некотором состоянии, определяемом предысторией разбора, и в зависимости от очередной лексемы предпринимает то или иное действие для перехода к новому состоянию. Различают два типа действий: **сдвиг**, т.е. чтение следующей входной лексемы, и **свертку**, т.е. применение одного из правил подстановки для замещения нетерминалом последовательности символов, соответствующей правой части правила.

Пользователь `yacc.py` должен описать структуру своей входной информации (**грамматику**) как набор **правил**. Грамматические правила описываются в терминах исходных конструкций, лексем, полученных при разборе с помощью модуля `lex.py`. Для синтаксического анализатора все лексем считаются терминальными символами грамматики. Для каждого правила есть возможность указать программные действия, осуществляемые при успешном разборе данного правила.

8.1. Установка и использование PLY

Несмотря на то, что PLY является пакетом Python, он не распространяется через пакетные менеджеры. Для установки необходимо скачать исходные коды из репозитория (<https://github.com/dabeaz/ply>), и скопировать в папку проекта. Для работы PLY требуется Python 3.6 или более новой версии, какие-либо пакеты не требуются.

Поскольку Python является интерпретируемым языком, а PLY написан полностью на Python, данный генератор лексических и синтаксических анализаторов на самом деле не генерирует файлы исходного кода, которые требуют дальнейшей компиляции, в отличие от рассмотренных ранее генераторов анализаторов для C/C++ и Java. Вместо этого пользователь должен создать собственный модуль Python с описанием правил лексического и синтаксического разбора (рассмотрены далее) и с использованием функций PLY для запуска.

8.2. Настройка лексического анализатора lex.py

В первую очередь необходимо определить список или кортеж лексем. Для этого используется переменная `tokens`. Каждая лексема в списке представляет собой строку, называемую типом лексем.

```
tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE',  
         'LPAREN', 'RPAREN')
```

Далее производится спецификация лексем. Каждая лексема определяется регулярным выражением, кроме того, при ее обработке можно добавить какое-либо действие и модифицировать возвращаемый токен. Если никакого действия не предполагается, то достаточно указать переменную с названием `t_тип_лексемы` и присвоить в качестве значения регулярное выражение:

```
t_PLUS = r'\+'
```

Обратите внимание, что имя, стоящее в переменной после префикса `t_`, должно в точности соответствовать одному из токенов.

Если необходимо произвести какие-либо действия при разборе лексемы, то необходимо определить функцию с именем `t_тип_лексемы`. Данная функция принимает на вход один параметр `t`, в который помещается экземпляр токена класса `LexToken`. Данный объект содержит поля:

`type` – соответствует типу лексемы (по умолчанию равен той части названия, что стоит после префикса `t_`)

`value` – исходный текст, которому соответствует разобранный лексема

`lineno` – номер строки, на которой произошел разбор

`lexpos` – позиция в строке, с которой произошел разбор.

Первой строчкой данной функции необходимо указать регулярное выражение, соответствующее данной лексеме в формате `docstring`. В конце функции необходимо вернуть объект `t`. Пример, в котором модифицируется значение объекта токена:

```
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Указание регулярного выражения в формате `docstring` накладывает ограничение – данное выражение не должно быть вычисляемым в процессе работы программы. Для указания вычисляемого выражения необходимо воспользоваться декоратором `@TOKEN`:

```
from ply.lex import TOKEN

digit = r'([0-9])'
nondigit = r'([_A-Za-z])'
identifier = fr'({nondigit}({digit}|{nondigit})*)'

@TOKEN(identifier)
def t_ID(t):
    ...
```

Для задания литералов используется переменная `literals` содержащая либо список символов, либо строку из символов. Символы-литералы – это лексемы, тип которых (`type`) равен их значению (`value`):

```
literals = [ '+', '-', '*', '/' ]
literals = "+-*/"
```

В результате такой спецификации лексем PLY производит единый конечный автомат, добавляя в него регулярные выражения для лексем в определенном порядке:

1. Сначала добавляются регулярные выражения лексем, специфицированных функциями в точности в том порядке, в котором они заданы в модуле.
2. Далее добавляются регулярные выражения лексем, специфицированных переменными, без действий, отсортированные в алфавитном порядке по убыванию регулярных выражений.
3. Далее добавляются простые регулярные выражения, соответствующие литералам.

При разборе регулярные выражения будут применяться в этом порядке. Иногда это может привести к не желаемому результату, например в случае разбора ключевых слов языка (такие конструкции как `for`, `while`, `if`, `class` и т.п.) конкурирующим регулярным выражением. К примеру при разборе идентификатора правилом `[a-zA-Z_][a-zA-Z_0-9]*` ключевые слова будут распознаны как идентификаторы. Для решения данной проблемы необходимо отделить ключевые слова, например следующим образом:

```
reserved = {
    'if' : 'IF',
    'then' : 'THEN',
    'else' : 'ELSE',
    'while' : 'WHILE',
    ...
}
```

```
tokens = ['LPAREN', 'RPAREN', ..., 'ID'] +
list(reserved.values())
```

```
def t_ID(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value, 'ID')
    return t
```

Данный код производит замену типа лексемы на тип, соответствующий ключевому слову, если такое есть, иначе же по умолчанию оставляет тип ID. В более сложных случаях можно воспользоваться заменой на основе регулярных выражений.

Для игнорирования лексем достаточно в специфицирующей функции не возвращать объект t, либо задать спецификацию в виде переменной с названием ignore_t_тип_лексемы:

```
def t_COMMENT(t):
    r'\#.*'
    pass
t_ignore_COMMENT = r'\#.*'
```

Для полного игнорирования отдельных символов входного потока задается переменная t_ignore, содержащая строку из таких символов:

```
t_ignore = '\s\t'
```

По умолчанию PLY не имеет функции подсчета номера текущей строки, ее можно задать следующим образом (обратите внимание на возможность обращения к объекту лексического анализатора lexer в объекте токена):

```
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
```

Обработка ошибок производится функцией t_error, которая вызывается в случае нахождения в потоке символа, не соответствующего ни одному правилу

лексического анализатора (обратите внимание на вызов функции skip(1) пропускающей данный символ):

```
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

8.3. Состояния лексического анализатора lex.py

При работе лексического анализатора может понадобиться разделение правил на группы по состояниям. Состояния определяются в переменной states в виде списка пар: (название, тип). Состояния бывают двух типов: включающие (inclusive) и исключаяющие (exclusive). Исключаяющие состояния при переходе в них полностью переписывают текущую группу регулярных выражений собственными, а включающие – добавляют к ней дополнительные регулярные выражения. По умолчанию, если состояния не указаны, анализатор находится в состоянии INITIAL, иначе – в первом указанном состоянии.

```
states = (
    ('base', 'exclusive'), # стартовое исключаящее состояние
    ('code', 'inclusive'), # включающее состояние code
)
```

Для назначения спецификации лексемы какому-либо или нескольким состояниям, необходимо добавить название состояния после префикса t_ в названии соответствующей переменной или функции. Если спецификация должна действовать в любом состоянии, то необходимо добавить ANY:

```
t_base_NUMBER = r'\d+' # Работает в base
t_code_ID = r'[a-zA-Z_][a-zA-Z0-9_]*' # Работает в code

def t_base_code_newline(t): ' # Работает в base и code
    r'\n'
    t.lexer.lineno += 1

t_ANY_ignore = " \t\n" # Игнорирует в любом состоянии

def t_code_error(t): # Обработка ошибок в code
```

```
pass
```

Для перехода в состояние можно использовать функцию `lexer.begin()`, либо оперировать со стеком состояний функциями `lexer.push_state()` и `lexer.pop_state()`:

```
def t_base_codestart(t):
    r'<code>'
    t.lexer.begin('code')

def t_base_codestart(t):
    r'<code>'
    t.lexer.push_state('code')

def t_code_codeend(t):
    r'</code>'
    t.lexer.pop_state()
```

8.4. Запуск лексического анализатора `lex.py`

Для сборки лексического анализатора необходимо в модуле, в котором задана спецификация лексем запустить функцию `lex()` модуля `lex.py`. В результате работы функции будет получен объект класса `Lexer`, который содержит две основные функции: `lexer.input(data)` и `lexer.token()`. Функция `input(data)` направляет входной поток (строку `data`) на вход анализатору, а функция `token()` производит одну итерацию разбора входного потока и возвращает лексему при успешном разборе либо `None` если достигнут конец потока:

```
import ply.lex as lex
... # Спецификация лексем
lexer = lex.lex()
with open("input.txt", "r") as f:
    data = f.read()
lexer.input(data)
t = lexer.token()
while(t is not None):
    print(t.type)
    t = lexer.token()
```

Существуют и другие возможности запуска, позволяющие более удобным образом организовать код анализатора. К примеру, спецификацию лексем можно поместить в отдельный модуль (для примера в модуль `tokrules.py`), тогда сборка лексического анализатора производится следующим образом:

```
import tokrules
lexer = lex.lex(module=tokrules)
```

Кроме того, можно поместить спецификацию в замыкание – то есть в функцию, производящую сборку лексического анализатора, либо в класс. Во втором случае необходимо добавить параметр `self` в функции спецификации лексем с действиями, а также методы доступа к объекту лексического анализатора. Данные примеры можно найти в официальной документации к PLY.

8.5. Пример лексического анализатора `lex.py`

Данный анализатор производит разбор арифметических выражений, состоящих из чисел, арифметических операторов и скобок:

```
import ply.lex as lex

tokens = ('NUMBER', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE',
         'LPAREN', 'RPAREN')

t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_LPAREN = r'\('
t_RPAREN = r'\)'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_newline(t):
    r'\n+'
```

```

t.lexer.lineno += len(t.value)

t_ignore = '\s\t'

def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)

lexer = lex.lex()
with open("input.txt", "r") as f:
    data = f.read()
lexer.input(data)
t = lexer.token()
while(t is not None):
    print(t.type)
    t = lexer.token()

```

8.6. Настройка синтаксического анализатора yacc.py

Настройка синтаксического анализатора начинается с определения лексем, аналогично настройке лексического анализатора lex.py в переменной tokens. Данная переменная может быть уже определена, если описание правил грамматического разбора находится в том же модуле/замыкании/классе, где и спецификация лексического анализатора, либо может быть импортирована:

```
from mylex import tokens
```

Эти лексемы будут терминалами при описании правил. Описание правил грамматического разбора состоит в задании функций с названиями вида p_название_функции. Данная функция принимает один параметр p, хранящий список считанных значений терминалов и нетерминалов соответствующих правилу. Первой строчкой в формате docstring задается правило контекстно-свободного разбора:

```

def p_expression_plus(p):
    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

```

В данном примере в правиле разбора слева от символа двоеточия стоит нетерминал `expression` (ему соответствует значение `p[0]`), а справа последовательность нетерминала `expression` (`p[1]`), терминала `PLUS` (лексемы, `p[2]`, соответствует `t.value` данной лексемы) и нетерминала `term` (`p[3]`). В качестве действия производится изменение значения левого нетерминала `expression`. В результате работы в стек будет помещено значение `p[0]`.

Отрицательные индексы для списка `p` имеют иное значение нежели обычно в Python, например `p[-1]` соответствует тому символу, что был до данного выражения.

Описания правил можно объединять с помощью символа `|`:

```
def p_expression(p):
    '''expression : expression PLUS term
                  | expression MINUS term'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
```

Символы-литералы в описании правила разбора должны заключаться в апострофы:

```
def p_binary_operators(p):
    '''expression : expression '+' term
                  | expression '-' term
    term          : term '*' factor
                  | term '/' factor'''
    if p[2] == '+':
        p[0] = p[1] + p[3]
    elif p[2] == '-':
        p[0] = p[1] - p[3]
    elif p[2] == '*':
        p[0] = p[1] * p[3]
    elif p[2] == '/':
        p[0] = p[1] / p[3]
```

Для обработки укорачивающего правила с пустым символом используется нетерминал `empty`:

```
def p_optitem(p):
    'optitem : item'
    '      | empty'
    ...
```

Для обработки произвольного набора ошибочных символов используется нетерминал error:

```
def p_statement_print_error(p):
    'statement : PRINT error SEMI'
    print("Syntax error in print statement. Bad expression")
```

Также, для обработки произвольной ошибки разбора, можно определить функцию p_error():

```
def p_error(p):
    print("Syntax error in input!")
```

По умолчанию стартовым нетерминалом назначается первый встреченный нетерминал в описании правил разбора. Для назначения другого нетерминала стартовым используется переменная start:

```
start = 'statement'
```

Для назначения приоритетов используется переменная precedence:

```
precedence = (
    ('nonassoc', 'LESSTHAN', 'GREATERTHAN'),
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVIDE'),
    ('right', 'UMINUS'),
)
```

8.7. Запуск синтаксического анализатора yacc.py

Для сборки синтаксического анализатора необходимо в модуле, в котором собран лексический анализатор и задана конфигурация правил, запустить функцию yacc() модуля yacc.py. В результате работы функции будет получен объект класса Parser, который содержит основную функцию parser.parse(s). Дан-

ная функция производит разбор входной строки *s*, используя ранее настроенный и собранный лексический анализатор:

```
import ply.lex as lex
import ply.yacc as yacc
... # Спецификация лексем
... # Конфигурация правил
lexer = lex.lex()
parser = yacc.yacc()
with open("input.txt", "r") as f:
    data = f.read()
result = parser.parse(data)
print(result)
```

8.8. Обработка конфликтов синтаксического анализатора yacc.py

При запуске синтаксического анализатора с опцией дебага (`yacc.yacc(debug=True)`) будет сгенерирован файл `parser.out` с описанием построенного автомата, разбирающего грамматику с описанием состояний, применяемых правил а также списком конфликтов свертки-свертки и сдвига-свертки. Данный файл аналогичен тому, что создает рассмотренный ранее генератор синтаксических анализаторов YACC. Рекомендации по разрешению конфликтов разбора те же самые (см. «Обработка ошибок при грамматическом разборе»).

8.9. Пример синтаксического анализатора yacc.py

Данный пример основан на лексическом анализаторе арифметических выражений, реализует вычисление данных выражений:

```
import ply.yacc as yacc
from calclex import tokens, lexer # импорт списка лексем и
анализатора

def p_expression_plus(p):
```

```

    'expression : expression PLUS term'
    p[0] = p[1] + p[3]

def p_expression_minus(p):
    'expression : expression MINUS term'
    p[0] = p[1] - p[3]

def p_expression_term(p):
    'expression : term'
    p[0] = p[1]

def p_term_times(p):
    'term : term TIMES factor'
    p[0] = p[1] * p[3]

def p_term_div(p):
    'term : term DIVIDE factor'
    p[0] = p[1] / p[3]

def p_term_factor(p):
    'term : factor'
    p[0] = p[1]

def p_factor_num(p):
    'factor : NUMBER'
    p[0] = p[1]

def p_factor_expr(p):
    'factor : LPAREN expression RPAREN'
    p[0] = p[2]

def p_error(p):
    print("Syntax error in input!")

lexer = lex.lex()
parser = yacc.yacc(lexer=lexer) # указание анализатора lexer
with open("input.txt", "r") as f:
    data = f.read()
result = parser.parse(data)
print(result)

```


ГЛАВА 9. ГЕНЕРАТОР АНАЛИЗАТОРОВ SLY

Генератор лексических и синтаксических анализаторов SLY (Sly Lex-Yacc) – это совместный порт генераторов LEX и YACC на язык Python организованный в виде пакета. Данный пакет содержит два основных класса: Lexer и Parser.

Класс Lexer используется для задачи разбора входного потока символов и распознавания лексических единиц (лексем) при помощи детерминированного конечного автомата, то есть для задачи токенизации. Правила разбора лексем задаются пользователем в виде регулярных выражений. К правилам разбора есть возможность добавлять программные действия, осуществляемые при успешном разборе очередной лексемы.

Класс Parser предназначен для описания синтаксического анализатора, обрабатывающего класс контекстно-свободных грамматик LALR(1), то есть поддерживающих восходящий разбор.

Данный пакет является развитием и продолжением пакета PLY, в его основе лежат те же принципы конфигурации анализаторов, однако он имеет иной, обратно не совместимый, интерфейс.

9.1. Установка и использование SLY

SLY не распространяется через пакетные менеджеры. Для установки необходимо скачать исходные коды из репозитория (<https://github.com/dabeaz/sly>), и скопировать в папку проекта. Для работы SLY требуется Python 3.6 или более новой версии, какие-либо пакеты не требуются.

Поскольку Python является интерпретируемым языком, а SLY написан полностью на Python, данный генератор лексических и синтаксических анализаторов на самом деле не генерирует файлы исходного кода, которые требуют дальнейшей компиляции, в отличие от рассмотренных ранее генераторов анализаторов для C/C++ и Java. Вместо этого пользователь должен создать соб-

ственный модуль Python с описанием правил лексического и синтаксического разбора (рассмотрены далее) и с использованием функций `SLY` для запуска.

9.2. Настройка лексического анализатора `Lexer`

Вся конфигурация лексического анализатора производится в классе, наследуемом от класса `Lexer`: В первую очередь необходимо определить множество лексем. Для этого используется поле `tokens`. Каждая лексема в множестве представляет собой идентификатор, называемый типом лексемы:

```
class MyLexer(Lexer):
    tokens = { ID, NUMBER, PLUS, MINUS, TIMES,
              DIVIDE, ASSIGN, LPAREN, RPAREN }
```

Далее производится спецификация лексем. Каждая лексема определяется регулярным выражением, кроме того, при ее обработке можно добавить какое-либо действие и модифицировать возвращаемый токен. Если никакого действия не предполагается, то достаточно указать поле с названием, соответствующим типу лексемы и присвоить в качестве значения регулярное выражение:

```
class MyLexer(Lexer):
    tokens = { PLUS }
    PLUS = r'\+'
```

Если необходимо произвести какие-либо действия при разборе лексем, то необходимо определить метод с именем, соответствующим имени лексем. Данная функция принимает на вход параметр `t`, в который помещается экземпляр токена класса `Token`. Данный объект содержит поля:

`type` – соответствует типу лексем (по умолчанию равен той части названия, что стоит после префикса `t_`)

`value` – исходный текст, которому соответствует разобранный лексема.

Данный метод необходимо декорировать при помощи `@_()` с указанием соответствующих регулярных выражений в качестве параметров декоратора. В кон-

це метода необходимо вернуть объект `t`. Пример, в котором модифицируется значение объекта токена:

```
class MyLexer(Lexer):
    tokens = { NUMBER }

    @_('d+')
    def NUMBER(self, t):
        t.value = int(t.value)
        return t
```

Есть возможность задания нескольких правил:

```
class MyLexer(Lexer):
    tokens = { NUMBER }

    @_('0x[0-9a-fA-F]+',
       'd+')
    def NUMBER(self, t):
        if t.value.startswith('0x'):
            t.value = int(t.value[2:], 16)
        else:
            t.value = int(t.value)
        return t
```

Другой вариант указания регулярного выражения, соответствующего методу – указание поля для данного типа лексемы с регулярным выражением:

```
class MyLexer(Lexer):
    tokens = { NUMBER }

    NUMBER = r'd+'
    def NUMBER(self, t):
        t.value = int(t.value)
        return t
```

Для задания литералов используется поле `literals` содержащее множество символов. Символы-литералы – это лексемы, тип которых (`type`) равен их значению (`value`):

```
literals = [ '+', '-', '*', '/' ]
literals = "+-*/"
```

В результате такой спецификации лексем SLY производит единый конечный автомат, добавляя в него регулярные выражения для лексем в том порядке, в котором они были определены в классе анализатора. В последнюю очередь добавляются регулярные выражения, соответствующие литералам.

При разборе регулярные выражения будут применяться в этом порядке. Иногда это может привести к не желаемому результату, например в случае разбора ключевых слов языка (такие конструкции как `for`, `while`, `if`, `class` и т.п.) конкурирующим регулярным выражением. К примеру при разборе идентификатора правилом `[a-zA-Z_][a-zA-Z_0-9]*` ключевые слова будут распознаны как идентификаторы. Для решения данной проблемы необходимо отделить ключевые слова, например следующим образом:

```
class MyLexer(Lexer):
    tokens = { ID, IF, ELSE, WHILE }

    ID = r'[a-zA-Z_][a-zA-Z_0-9]*'
    ID['if'] = IF
    ID['else'] = ELSE
    ID['while'] = WHILE
```

Данный код производит замену типа лексем на тип, соответствующий ключевому слову, если такое есть, иначе же по умолчанию оставляет тип `ID`.

Для игнорирования лексем необходимо задать спецификацию в виде поля с названием, начинающимся с префикса `ignore_`, либо метода с таким же префиксом, не возвращающего ничего. Для полного игнорирования отдельных символов входного потока задается поле `ignore`, содержащая строку из таких символов:

```
class MyLexer(Lexer):
    ignore = '\s\t'

    ignore_comment = r'\#.*'
    @_(r'\n+')
    def ignore_newline(self, t):
        pass
```

По умолчанию SLY не имеет функции подсчета номера текущей строки, ее можно задать следующим образом (обратите внимание на инкремент поля `lineno` класса анализатора):

```
class MyLexer(Lexer):
    @_(r'\n+')
    def ignore_newline(self, t):
        self.lineno += len(t.value)
```

Обработка ошибок производится методом `error`, который вызывается в случае нахождения в потоке символа, не соответствующего ни одному правилу лексического анализатора (обратите внимание на инкремент поля `index`, за счет чего производится пропуск данного символа):

```
class MyLexer(Lexer):
    def error(self, t):
        print("Illegal character '%s'" % t.value[0])
        self.index += 1
```

9.3. Запуск лексического анализатора `Lexer`

Для сборки лексического анализатора необходимо создать экземпляр класса анализатора и вызвать метод `tokenize(data)`. Данная функция направляет входной поток (строку `data`) на вход анализатору и возвращает объект-генератор, при итерировании которого производится разбор очередной лексемы входного потока:

```
from sly import Lexer
class MyLexer(Lexer):
    ... # Спецификация лексем

lexer = MyLexer()
with open("input.txt", "r") as f:
    data = f.read()
for tok in lexer.tokenize(data):
    print(tok)
```

9.4. Пример лексического анализатора Lexer

Данный анализатор производит разбор арифметических выражений, состоящих из чисел, арифметических операторов и скобок:

```
from sly import Lexer

class CalcLexer(Lexer):
    tokens = { ID, NUMBER, PLUS, MINUS, TIMES,
              DIVIDE, ASSIGN, LPAREN, RPAREN }

    ID = r'[a-zA-Z_][a-zA-Z0-9_]*'
    PLUS = r'\+'
    MINUS = r'\-'
    TIMES = r'\*'
    DIVIDE = r'\/'
    ASSIGN = r'\='
    LPAREN = r'\('
    RPAREN = r'\)'

    @_('d+')
    def NUMBER(self, t):
        t.value = int(t.value)
        return t

    ignore = ' \t'

lexer = CalcLexer()
with open("input.txt", "r") as f:
    data = f.read()
for tok in lexer.tokenize(data):
    print(tok)
```

9.5. Настройка синтаксического анализатора Parser

Конфигурация синтаксического анализатора производится в классе, наследуемом от класса Parser: Настройка синтаксического анализатора начинается с определения лексем, аналогично настройке лексического анализатора в

поле `tokens`. Данное поле может быть уже определено в классе лексического анализатора, в таком случае его необходимо импортировать:

```
from mylex import MyLexer

class MyParser(Parser):
    tokens = MyLexer.tokens
```

Эти лексемы будут терминалами при описании правил. Описание правил грамматического разбора состоит в задании методов с названиями, соответствующими левым нетерминалам правил. Данные методы принимают на вход параметр `p`, который хранит набор считанных значений терминалов и нетерминалов соответствующих правилам. Правила контекстно-свободного разбора задаются в декораторе `@_()`:

```
class MyParser(Parser):
    @_('expr PLUS term')
    def expr(self, p):
        return p[0] + p[2]
```

В данном примере строка в декораторе `@_()` представляет собой правую часть, а название метода – нетерминал левой части правила грамматики. Нетерминалу `expr` соответствует `p[0]`, терминалу `PLUS` лексеме соответствует `p[1]`, содержащее `t.value` данной лексемы, и нетерминалу `term` соответствует `p[2]`. В результате работы в стек будет помещено возвращаемое значение метода.

Иной способ обращения к элементам набора `p` – по их имени:

```
class MyParser(Parser):
    @_('expr PLUS term')
    def expr(self, p):
        return p.expr + p.term
```

Отрицательные индексы для набора `p` имеют иное значение нежели обычно в Python, например `p[-1]` соответствует тому символу, что был до данного выражения.

Можно задавать несколько правил:

```
class MyParser(Parser):
    @_('expr PLUS term',
       'expr MINUS term')
    def expr(self, p):
        if p[1] == '+':
            return return p.expr + p.term
        elif p[1] == '-':
            return return p.expr - p.term
```

Символы-литералы в описании правила разбора должны заключаться в кавычки:

```
class MyParser(Parser):
    @_('expr "+" term')
    def expr(self, p):
        return p.expr + p.term
```

Для обработки укорачивающего правила с пустым символом используется пустая строка правила:

```
class MyParser(Parser):
    @_('')
    def empty(self, p):
        pass

    @_('item',
       'empty')
    def optitem(self, p):
        ...
```

Для обработки произвольного набора ошибочных символов используется нетерминал error:

```
class MyParser(Parser):
    @_('PRINT error SEMI')
    def statement(self, p):
        print("Syntax error in print statement.")
```

Также, для обработки произвольной ошибки разбора, можно определить метод error():


```
class MyParser(Parser):
    def error(p):
        print("Syntax error in input!")
```

По умолчанию стартовым нетерминалом назначается первый встреченный метод нетерминал в описании правил разбора.

Для назначения приоритетов используется поле `precedence`:

```
class MyParser(Parser):
    precedence = (
        ('nonassoc', LESSTHAN, GREATERTHAN),
        ('left', PLUS, MINUS),
        ('left', TIMES, DIVIDE),
        ('right', UMINUS),
    )
```

9.6. Запуск синтаксического анализатора Parser

Для сборки синтаксического анализатора необходимо создать экземпляры классов лексического и синтаксического анализатора. Данный объект содержит метод `parse()`, принимающий на вход список лексем. В качестве списка можно передать результат работы функции `tokenize`:

```
from sly import Lexer, Parser
class MyLexer(Lexer):
    ... # Спецификация лексем

class MyParser(Parser):
    ... # Конфигурация правил

lexer = MyLexer()
parser = MyParser()
with open("input.txt", "r") as f:
    data = f.read()
result = parser.parse(lexer.tokenize(data))
print(result)
```

9.7. Обработка конфликтов синтаксического анализатора Parser

Если в классе синтаксического анализатора определить поле `debugfile = 'parser.out'`, то при его запуске будет сгенерирован файл `parser.out` с описанием построенного автомата, разбирающего грамматику с описанием состояний, применяемых правил а также списком конфликтов свертки-свертки и сдвига-свертки. Данный файл аналогичен тому, что создает рассмотренный ранее генератор синтаксических анализаторов YACC. Рекомендации по разрешению конфликтов разбора те же самые (см. «Обработка ошибок при грамматическом разборе»).

9.8. Пример синтаксического анализатора Parser

Данный пример основан на лексическом анализаторе арифметических выражений, реализует вычисление данных выражений:

```
from sly import Parser
from calclex import CalcLexer

class CalcParser(Parser):
    tokens = CalcLexer.tokens

    @_('expr PLUS term')
    def expr(self, p):
        return p.expr + p.term

    @_('expr MINUS term')
    def expr(self, p):
        return p.expr - p.term

    @_('term')
    def expr(self, p):
        return p.term

    @_('term TIMES factor')
    def term(self, p):
        return p.term * p.factor
```

```
@_('term DIVIDE factor')
def term(self, p):
    return p.term / p.factor
```

```
@_('factor')
def term(self, p):
    return p.factor
```

```
@_('NUMBER')
def factor(self, p):
    return p.NUMBER
```

```
@_('LPAREN expr RPAREN')
def factor(self, p):
    return p.expr
```

```
lexer = CalcLexer()
parser = CalcParser()
with open("input.txt", "r") as f:
    data = f.read()
result = parser.parse(lexer.tokenize(data))
print(result)
```

9.9. Язык JSON

Язык JSON (JavaScript Object Notation) является текстовым машиночитаемым и человекочитаемым форматом обмена данных (языком разметки данных). Это подмножество языка Javascript, которое определяет литерал произвольного объекта, который в рамках Javascript представлен структурой данных типа хэш-таблица (HashMap, Map, dict в иных языках программирования) с набором пар вида ключ-значение, в которых ключ является уникальным и представлен хэшируемым типом данных (обычно это строки) а значение – любое значение из типов данных Javascript: строка, число, массив, объект.

Пример текста в языке JSON:

```
{
  "menu": {
```

```

    "id": "file",
    "value": "File",
    "popup": {
        "menuitem": [
            {"value": "New", "onclick": "CreateNewDoc()"},
            {"value": "Open", "onclick": "OpenDoc()"},
            {"value": "Close", "onclick": "CloseDoc()"}
        ],
        "delay": 1.5
    }
}
}
}

```

Задача разбора текстов и файлов в формате JSON является распространенной на практике, так как этот формат используется для обмена данными в веб-запросах, для хранения данных, в документоориентированных базах данных, для записи файлов конфигурации приложений.

Конфигурация анализатора SLY для разбора JSON приведена далее. Данный пример по своей сути реализует пакет json из стандартной библиотеки Python, то есть производит десериализацию данных из формата JSON в соответствующие типы данных языка Python.

```

from sly import Lexer, Parser

class JSONLexer(Lexer):
    tokens = {"FLOAT", "INTEGER", "STRING"}

    literals = {'{', '}', '[', ']', ',', ':'}
    ignore = " \t\n"

    @_("\""")
    def STRING(self, t):
        t.value = t.value.strip("\"")
        return t

    @_("d+\.\d*")
    def FLOAT(self, t):
        t.value = float(t.value)
        return t

```

```

    @_(r"\d+")
    def INTEGER(self, t):
        t.value = int(t.value)
        return t

class JSONParser(Parser):
    tokens = JSONLexer.tokens
    start = "json"

    @_('object',
        'array')
    def json(self, p):
        return p[0]

    @_('{" members }')
    def object(self, p):
        return {key: value for key, value in p.members}

    @_('pair')
    def members(self, p):
        return [p.pair]

    @_('pair "," members')
    def members(self, p):
        return [p.pair] + p.members

    @_('STRING ":" value')
    def pair(self, p):
        return p.STRING, p.value

    @_('[" elements ]')
    def array(self, p):
        return p.elements

    @_('value')
    def elements(self, p):
        return [p.value]

    @_('value "," elements')
    def elements(self, p):
        return [p.value] + p.elements

```

```

    @_('STRING',
      'INTEGER',
      'FLOAT',
      'object',
      'array')
    def value(self, p):
        return p[0]

    def error(self, p):
        raise ValueError("Parsing error at token %s" % str(p))

lexer = JSONLexer()
parser = JSONParser()
with open("input.json", "r") as f:
    data = f.read()
result = parser.parse(lexer.tokenize(data))
print(result)

```

9.10. Язык STL

Язык STL (stereolithography) является форматом хранения данных (языком разметки данных) трёхмерных моделей объектов для использования в технологиях трехмерной печати: стереолитографии, экструзии, в 3D принтерах. Информация об объекте хранится как список треугольных граней, которые описывают его поверхность, и их нормалей. Файлы данного формата создаются в различных CAD-программах для 3D моделирования.

Файл типа STL начинается со строки:

```
solid name
```

где name – необязательная строка (если name опущено, всё равно должен быть пробел после solid). Файл продолжается произвольным числом треугольников, описываемых следующим способом:

```
facet normal ni nj nk
outer loop
vertex v1x v1y v1z
vertex v2x v2y v2z
```

```
vertex v3x v3y v3z
endloop
endfacet
```

где каждое n и v – число с плавающей точкой в формате: знак, мантисса, «e», знак, экспонента, например «-2.648000e-002». Файл завершается строкой:

```
endsolid name
```

Далее приведена конфигурация SLY для разбора файла STL. Анализатор, задаваемый данной конфигурацией, формирует на основе разобранного файла экземпляры классов Vertex (вершина), Facet (треугольник или полигон, набор из трех вершин) и Solid (фигура, произвольное множество треугольников).

```
from dataclasses import dataclass
from typing import List, Tuple
```

```
from sly import Lexer, Parser
```

```
@dataclass
```

```
class Vertex:
```

```
    coords: Tuple[float, float, float]
```

```
    def __repr__(self):
```

```
        return "VERTEX(%f,%f,%f)" % self.coords
```

```
@dataclass
```

```
class Facet:
```

```
    vertices: Tuple[Vertex, Vertex, Vertex]
```

```
    normal: Vertex
```

```
    def __repr__(self):
```

```
        return "\t FACET({normal})\n\t\t{vertices}".format(
            normal=repr(self.normal),
```

```
            vertices="\n\t\t".join(repr(vertex) for vertex in
self.vertices)
```

```
    )
```

```

@dataclass
class Solid:
    name: str
    facets: List[Facet]

    def __repr__(self):
        return "SOLID({name})\n{facets}\n".format(
            name=self.name,
            facets="\n".join(repr(facet) for facet in
self.facets)
        )

class STLlexer(Lexer):
    tokens = {"SOLID_START", "SOLID_END",
             "FACET_START", "FACET_END", "NAME_LITERAL",
             "LOOP_START", "LOOP_END",
             "NAME_LITERAL", "VERTEX_TAG", "FLOAT"}

    @_(r"[ \t\n]+")
    def ignore_whitespace(self, t):
        self.lineno += t.value.count("\n")

    SOLID_START = r"solid"
    SOLID_END = r"endsolid"
    FACET_START = r"facet normal"
    FACET_END = r"endfacet"
    LOOP_START = r"outer loop"
    LOOP_END = r"endloop"
    VERTEX_TAG = r"vertex"

    @_("[+-]?\d+(\.\d*(e[+-]?\d+)?)?")
    def FLOAT(self, t):
        t.value = float(t.value)
        return t

    NAME_LITERAL = r"\w+"

class STLParser(Parser):
    tokens = STLlexer.tokens
    start = "stl"

```



```

    @_("solid")
    def stl(self, p):
        return p.solid

    @_("SOLID_START NAME_LITERAL facets_list SOLID_END")
    @_("SOLID_START NAME_LITERAL facets_list SOLID_END
NAME_LITERAL")
    def solid(self, p):
        return Solid(name=p[1], facets=p.facets_list)

    @_("facet")
    def facets_list(self, p):
        return [p.facet]

    @_("facet facets_list")
    def facets_list(self, p):
        return [p.facet] + p.facets_list

    @_("FACET_START triplet loop FACET_END")
    def facet(self, p):
        return Facet(vertices=p.loop, normal=Vertex(p.triplet))

    @_("LOOP_START vertex vertex vertex LOOP_END")
    def loop(self, p):
        return p[1], p[2], p[3]

    @_("VERTEX_TAG triplet")
    def vertex(self, p):
        return Vertex(p.triplet)

    @_("FLOAT FLOAT FLOAT")
    def triplet(self, p):
        return p[0], p[1], p[2]

    def error(self, token):
        pass

lexer = STLLexer()
parser = STLParser()
with open("input.stl", "r") as f:
    data = f.read()

```

```
result = parser.parse(lexer.tokenize(data))
print(result)
```

9.11. Язык f-string

Язык f-string является подмножеством языка Python для задания литералов форматированных строк вида:

```
f"Привет, меня зовут {person.surname} {person.name}. Мне {person.age} лет, я живу в {cities[1]}."
f"Сумма {x} и {y} равна {x + y}"
```

Форматированные строки – это вычисляемые в процессе работы программы строки с подстановками переменных и выражений. Перед литералом такой строки ставится символ `f` для указания факта, что это форматированная строка, подлежащая вычислению. Подстановки переменных и выражений указываются внутри фигурных скобок и могут представлять собой любую операцию на языке программирования. При необходимости, язык f-string можно реализовать в рамках другого языка программирования.

Далее представлена конфигурация SLY для языка f-string. Основной задачей этого анализатора является извлечение подстановок для дальнейшего их вычисления.

```
from sly.lex import Lexer
from sly.yacc import Parser, YaccProduction
```

```
class FStringLexer(Lexer):
    tokens = {
        "STRING_START",
        "STRING_END",
        "EXPR_START",
        "EXPR_END",
        "PIECE",
    }

    ignore = " \t"
```

```

STRING_START = r"f\"
STRING_END = r\"
EXPR_START = r"(?!\\)\{(?!\})"
EXPR_END = r"(?!\\)\}(?!\\)"
PIECE = r"[\{\}\\"'+|\{\{|}\}"

```

```

class FStringParser(Parser):
    tokens = FStringLexer.tokens

    def __init__(self):
        self.names = {}

    @_("STRING_START parts STRING_END")
    def fstring(self, p):
        return p

    @_("parts expr",
       "parts PIECE",
       "PIECE",
       )
    def parts(self, p):
        return p

    @_("EXPR_START parts EXPR_END")
    def expr(self, p):
        return p

lexer = FStringLexer()
parser = FStringParser()
with open("input.txt", "r") as f:
    data = f.read()
result = parser.parse(lexer.tokenize(data))
print(result)

```

СПИСОК ОСНОВНОЙ ЛИТЕРАТУРЫ

1. *Мотвани Р.* Введение в теорию автоматов, языков и вычислений / Р. Мотвани, Д.Э. Хопкрофт, Д.Д. Ульман. – М.: Вильямс, 2016. – 528 с.
2. *Ахо А.В.* Компиляторы: принципы, технологии и инструментарий / А.В. Ахо, Р. Сети, М.С. Лам, Д.Д. Ульман. – М.: Диалектика-Вильямс, 2017. – 1184 с.
3. *Серебряков В.А.* Теория и реализация языков программирования: Курс лекций / В.А. Серебряков, М.П. Галочкин, Д.Р. Гончар, М.Г. Фуругян – М.: Интуит НОУ, 2016. – 372 с. – URL: <https://book.ru/book/918252> (дата обращения: 28.05.2024).
4. *Niemann T.* Lex & Yacc / T. Niemann. – Portland, Oregon: epaperpress.com. – 41 p. – URL: <https://epaperpress.com/lexandyacc/download/LexAndYacc.pdf> (дата обращения: 28.05.2024).
5. *Levine J.R.* Lex & Yacc, 2nd edition / J.R. Levine, T. Mason, D. Brown. – O'Reilly Media, 2016. – 354 с. – URL: <http://www.nylxs.com/docs/lexandyacc.pdf> (дата обращения: 28.05.2024).
6. *Appel A.W.* Modern Compiler Implementation in C / A.W. Appel. – Cambridge University Press, 2004. – 556 p.
7. Java™ Platform, Standard Edition 8 API Specification. – URL: <https://docs.oracle.com/javase/8/docs/api/> (дата обращения: 28.05.2024).
8. JFlex. – URL: <http://www.jflex.de/> (дата обращения: 28.05.2024).
9. CUP User's Manual. – URL: <http://www2.cs.tum.edu/projects/cup/docs.php> (дата обращения: 28.05.2024).
10. Python Documentation. – URL: <https://docs.python.org/3/> (дата обращения: 28.05.2024).
11. PLY (Python Lex-Yacc). – URL: <https://ply.readthedocs.io/> (дата обращения: 28.05.2024).

12.SLY (Sly Lex Yacc). – URL: <https://sly.readthedocs.io/> (дата обращения: 28.05.2024).

СПИСОК ДОПОЛНИТЕЛЬНОЙ ЛИТЕРАТУРЫ

1. *Ахо А.В.* Теория синтаксического анализа, перевода и компиляции, в 2-х т. / А.В. Ахо, Д.Д. Ульман. – М.: Мир, 1978 – 1104 с.
2. *Серебряков В.А.* Основы конструирования компиляторов / В.А. Серебряков, М.П. Галочкин. – URSS, 2001. – 224 с.
3. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин / Д. Грис. – М.: Мир, 1975. – 544 с.
4. *Кнут Д.* Искусство программирования. Т. 1. Основные алгоритмы / Д. Кнут. – М.: Диалектика (Вильямс), 2020. – 720 с.
5. *Кнут Д.* Семантика контекстно-свободных языков / Д. Кнут // Семантика языков программирования. – М.: Мир, 1980. – С. 137–161.
6. *Серебряков, В. А.* Теория и реализация языков программирования: учебное пособие / В. А. Серебряков. – Москва : ФИЗМАТЛИТ, 2012. – 236 с.
7. *Мартыненко Б.К.* Языки и трансляции / Б.К. Мартыненко. – СПб.: Издательство Санкт-Петербургского университета. 2013. – 268 с.
8. *Aho A.U.* Code generation using tree matching and dynamic programing / A.U. Aho, M. Ganapathi, S.W. Tjiang. – ACM Trans. Progr. Languages and Systems, 1989. – V.11. – N 4. – P. 491–516. – URL: <https://typeset.io/pdf/code-generation-using-tree-matching-and-dynamic-programming-3wz3v3wx5d.pdf> (дата обращения: 28.05.2024).
9. *Fraser C. W.* A Retargetable compiler for ANSI C / C.W. Fraser, D.R. Hanson. – SIGPLAN Notices, 1991. – V. 26. – URL: <https://dl.acm.org/doi/pdf/10.1145/122616.122621> (дата обращения: 28.05.2024).

10. *Graham S.L.* An improved context-free recognizer / S.L. Graham, M.A. Harrison, W.L. Ruzzo. – ACM Trans. Program. Languages and Systems, 1980. – N. 2. – URL:
https://web.archive.org/web/20070824071546id_/http://www.cs.ucdavis.edu/~gusfield/cs224sp07/p415-graham.pdf
11. *Harrison M.A.* Introduction to formal language theory / M.A. Harrison. – Addison-Wesley, 1978. – 608 p.