КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра теоретической кибернетики

А.М. ГУСЕНКОВ, Н.А. ПРОКОПЬЕВ

СПЕЦИАЛИЗИРОВАННЫЕ ЯЗЫКИ ОБРАБОТКИ ИНФОРМАЦИИ

Учебно-методическое 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: учебно-методическое пособие / А.М. Гусенков, Н.А. Прокопьев. – Казань: Казанский федеральный университет, 2024. – 108 с.

В учебно-методическом 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).

Учебно-методическое 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 преподавателями и студентами математических специальностей.

СОДЕРЖАНИЕ

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 (C, C++, 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 C.

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 лексемы 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 C.

Генератор yacc отрабатывает 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 yacc, 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 сокраще-

ниях 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 лексемы.

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.

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.

Формальные языки

Алфавит – это множество атомарных символов. Пример: A_1 = {'a', 'b', 'c', 'd'}, A_2 = {'0', '1', '&'}, A_3 = {'формула', 'число', '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 автомата, 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 $w_1w_2w_3...w_n$, где w_i из L_1, $n \geq 0$, включает в себя пустое слово e,
4. **L_1^R** – обращение, содержит обращенные слова из языка L1,
5. **L_1+L_2** – смешение языков, содержит слова вида $v_1w_1v_2w_2...v_nw_n$, где v_i из L_1, w_i из L_2, $n>0$.

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 $n \geq 0$, $xy^nz \in L$

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.

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,

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

$\boldsymbol{\lambda: S \times X \rightarrow Y}$ – 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 – зацикливается.

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.

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 <левое слово> → <правое слово>, 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.

```
T       =       {'0','1','2','3','4','5','6','7','8','9','+','-
','*','/','(',')'},
N = {'ФОРМУЛА', 'ЗНАК', 'ЧИСЛО', 'ЦИФРА'},
P = {
     'ФОРМУЛА' →'ФОРМУЛА' 'ЗНАК' 'ФОРМУЛА',
     'ФОРМУЛА' →'ЧИСЛО',
     'ФОРМУЛА' → ('ФОРМУЛА'),
     'ЗНАК' →'+' | '-' | '*' | '/' ,
     'ЧИСЛО' →'ЦИФРА',
     'ЧИСЛО' →'ЧИСЛО' 'ЦИФРА',
     'ЦИФРА' → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' |
'9' ,
}
s = 'ФОРМУЛА'
```

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 → uwv, где A – символ из N, u, v – слова из (T∪N)*, w – слово из (T∪N)+ для неукорачивающих КЗ-грамматик, w из (T∪N)* для укорачивающих.

Тип 2: контекстно-свободные. Все правила имеют вид A → w, где A – символ из N, w – слово из (T∪N)+ для неукорачивающих КС-грамматик, w из (T∪N)* для укорачивающих.

Тип 3: регулярные. Все правила имеют вид A → Ba или A → a для леволинейных грамматик, A → aB или A → 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 →aS
     S →bA
     A →e
     A →cA
}
```

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

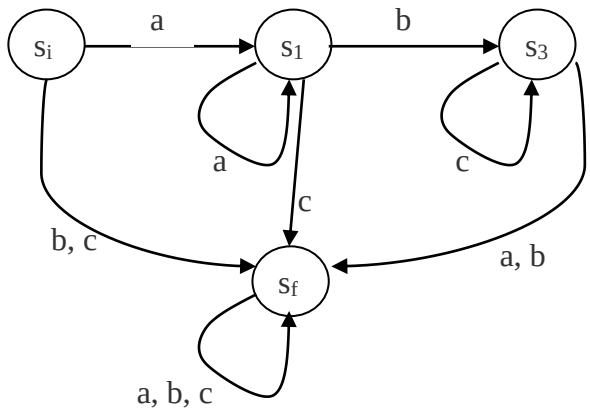

Рис. 1. Автомат распознавания регулярной грамматики

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 = txvyw, где длина цепочки xvy не превосходит k, и либо x, либо y – непустое слово;
2. для всякого натурального числа $m > 0$ цепочка $tx^m vy^m w$ 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 (со-

ответственно 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) * ЦИФРА →
   (4 + 56) * 9

Пример: КС-языки

1. Язык палиндромов L = {w ∈ T | w = wR}

```
T = некоторый алфавит {α ∈ T}
N = {S}
s = S
P = {
     S →αSα
     S →α
     S →ε
}
```

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 →aA
    A →Sb
    S →ε
}
```

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: $L = \{a^{n\wedge 2}\}$.

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. Суще-

ственным 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.	

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 С.

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 С.

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.

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.

Унарные операции

Пусть Rexp – произвольное регулярное выражение.

Rexp? – определяет необязательное вхождение фрагмента, соответствующего регулярному выражению Rexp, т.е. повторение фрагмента 0 или 1 раз.

Rexp* – определяет n-кратное (n>=0) последовательное повторение фрагмента, соответствующего регулярному выражению Rexp.

Rexp+ – определяет n-кратное (n>0) последовательное повторение фрагмента, соответствующего регулярному выражению Rexp.

Rexp{n1,n2} – определяет n-кратное (n1<=n<=n2) последовательное повторение фрагмента, соответствующего регулярному выражению Rexp.

^Rexp – определяет фрагмент, соответствующий регулярному выражению Rexp и стоящий в начале строки.

Rexp$ – определяет фрагмент, соответствующий регулярному выражению Rexp и стоящий в конце строки.

(Rexp) – эквивалентно Rexp.

Бинарные операции

Пусть Rexp1 и Rexp2 – произвольное регулярное выражение.

Rexp1|Rexp2 – определяет фрагмент, соответствующий либо выражению Rexp1, либо выражению Rexp2.

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.

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/xy

– 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.

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.

Секция определений

Секция определений может содержать:

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

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 С 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.

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 их указанных состояний.

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

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

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

Встроенные переменные

yytext[]

– 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 yytext.

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”,yytext);
```

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.

Встроенные функции

yymore()

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

yyless(n)

– Оставляет в массиве **yytext** первые n символов, а остальные возвращает во входной поток. Переменная **yyleng** принимает значение n. Лексический анали-

затор 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.

yywrap()

– 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 yywrap возвращает 1. Если имеется необходимость продолжить ввод данных из другого источника, пользователь должен написать свою версию функции yywrap(), которая организует новый входной поток и возвратит 0.

Пример: Входной файл.

```
%%
\"[^"]*  { if( yytext[yyleng - 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 двой-

ная 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 **yytext** мы получим: "абв\"эюя, что и требовалось.

Встроенные макрооперации

ECHO

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

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.

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 yyless 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++;
          yyless(1);
        }
he      h++;
```

Так как **she** включает в себя **he**, анализатор (без использования функции yyless(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 АССВ сначала будет распознана первым правилом, а затем первые три символа этой последовательности АСС будут распознаны вторым правилом.

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 С, 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 С, 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 С 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.

Секция подпрограмм

В секции подпрограмм размещаются функции, написанные на языке С, 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.

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 **yywrap** указывает, что лексический анализатор должен завершить работу.

Пример 3.

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

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 С 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.

ГЕНЕРАТОР СИНТАКСИЧЕСКИХ АНАЛИЗАТОРОВ YACC

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

Генератор уасс отрабатывает 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 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 **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 лексемы.

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

Набор лексем определяется разработчиком лексического анализатора. Для синтаксического анализатора все лексемы 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.

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.

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 лексемы, 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 С 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.

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.

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.

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, заключен-

ные 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 задавать 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.

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 **ITEM**). 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.

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().

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 лексемы автоматически попадает в стек после ее распознава-

ния 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 сверток.

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, …, 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 уасс.

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 yylval;
```

Примером значения лексемы могут служить числовое значение цифры, вычисленное значение константы, адрес идентификатора в таблице имен (построение таблицы имен удобно осуществлять лексическим анализатором). Заметим, 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.

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 разобранная часть строки приведена к виду expr*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 expr*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 отличии 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 yacc 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 yacc уже на этапе построения грамматического анализатора. При этом yacc выдает сообщение о числе выявленных конфликтных ситуаций обоих видов, а в выходной файл y.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 A 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 y.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 лексемы, 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 y.output описания этих конфликтов.

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

ной 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 уасс **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**. Больший приоритет правила вызывает свертку по нему, больший приоритет лексемы вызывает сдвиг.
- Если **приоритеты** заданы и **совпадают**, то принимается во внимание заданная одновременно с приоритетом **ассоциативность**: в случае **левой ассоциативности** 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**.

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> accept
```

– 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 y.output записано:

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

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.

Обработка ошибок при грамматическом разборе

Если входной поток не удовлетворяет заданной грамматике, то грамматический анализатор в момент ввода лексемы, 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 yyerror(). 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 yyerror() 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 лексемы. 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 лексемы 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 **yyerror** и **yyclearin,** которые уасс на макроуровне расширяет в нужные последовательности. Оператор yyerror 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 yyclearin 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();
                   yyclearin;
                   yyerror;
                 }
```

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

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 уасс 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 уасс и lex, а функции **main()** и **yyerror(s)** имеются в стандартной библиотеке уасс. 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);}
```

Пример входного файла

Ниже приведен полный входной файл для уасс, реализующий небольшой настольный калькулятор; калькулятор имеет 26 регистров, помеченных буквами от a до z, и разрешает использовать арифметические выражения, содержащие 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]);
```

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 yylex() 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 yacc.

Если функция yylex() 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 yyerror() вместо библиотечных. Для этого достаточно поместить текст этих функции в секцию подпрограмм входного файла yacc.

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

Использование генераторов flex и bison

Генераторы flex и bison являются совместимыми аналогами lex и yacc, 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 yacc.

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 С 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 С и поместит его в файл со именем **Yfile.tab.c** (зависящим от имени входного файла).

В случае если функция yylex() 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.**

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.

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.

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.

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.

Секция код пользователя

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

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 "Yylex", который записывается в **файл** "Yylex.java".

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

– Создаёт 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.

```
%apiprivate
```

– 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 – yylex'.

```
%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;
Rexp – 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.

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 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 их 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.

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.

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()+">");
                  }
```

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).

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 отрабатывает 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 (последовательность) 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.

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.

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 sym и parser.

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.

Секция список терминальных и нетерминальных символов

Секция используется для именования 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"

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;
```

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 ::= tnlist {: 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 уасс.

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; :}
          ;
```

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
        ;
```

Совместное 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.

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 распознания 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 лексемы считаются терминальными символами грамматики. Для каждого правила есть возможность указать программные действия, осуществляемые при успешном разборе данного правила.

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.

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 лексемы.

```
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 разобранная 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, соответствующе 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_ingore, 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)
```

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 исключающие (exclusive). Исключающие 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 исключающее 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()
```

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 мож-

но 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.

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()
```

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 последова-

тельность 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'),
)
```

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 функци parser.parse(s). Данная 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)
```

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»).

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)
```

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).

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.