КРАТКОЕ РУКОВОДСТВО ПО LEX И YACC

Оглавление

1 Введение....2
2 Lex....4
2.1 Теория....4
2.2 Практика....5
3 Yacc....9
3.1 Теория....9
3.2 Практика, часть I....10
3.3 Практика, часть 2....13
4 Калькулятор....16

1 Введение

До 1975 года написание компилятора было очень трудоемким процессом. Затем Lesk [1975] и Johnson [1975] опубликовали статьи о lex и yacc. Эти утилиты значительно упрощают написание компилятора.

Детали реализации lex и yacc можно найти в Aho [1986]. Lex и yacc доступны из:

- Mortice Kern Systems (MKS) (www.mks.com);
- GNU flex и bison (www.gnu.org);
- Ming (www.mingw.org);
- Cygwin (www.cygwin.com);
- моя версия Lex и Yacc (epaperpress.com).

Версия от MKS представляет собой высококачественный коммерческий продукт, который продается по цене около 300 долларов США. Программное обеспечение GNU бесплатно. Вывод из flex может быть использован в коммерческом продукте, и, начиная с версии 1.24, то же самое верно и для bison. Ming и Cygwin — это 32-разрядные порты программного обеспечения GNU для Windows. Фактически Cygwin — это порт операционной системы Unix для Windows с компиляторами gcc и g++.

Моя версия основана на версии Ming, но скомпилирована с помощью Visual C++ и включает исправление незначительной ошибки в процедуре обработки файлов. Если вы загружаете мою версию, не забудьте сохранить структуру каталогов при распаковке.

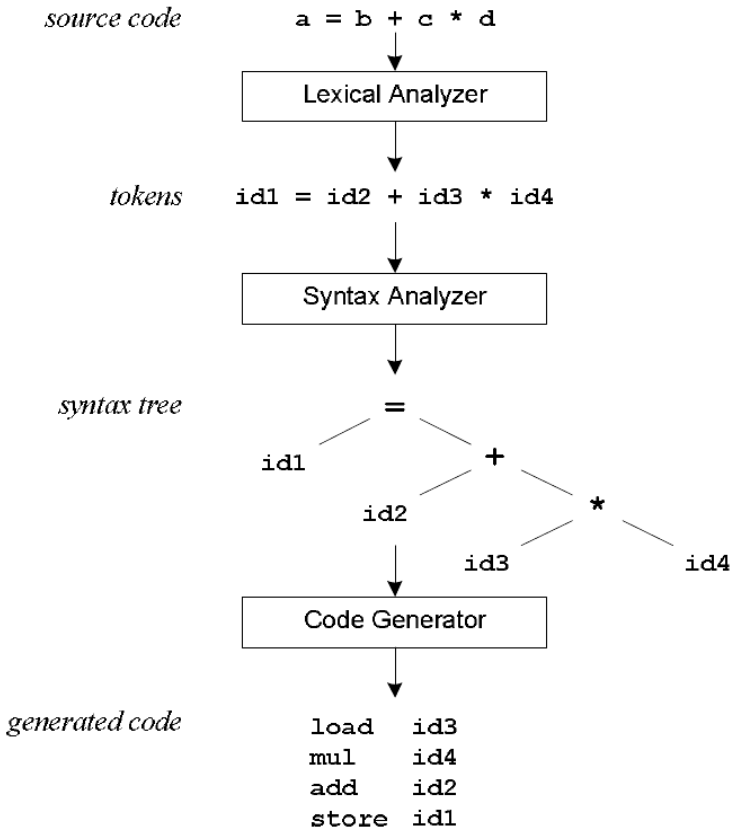

Рисунок 1 – Последовательность компиляции

Lex генерирует код C для лексического анализатора или сканера. Он использует шаблоны, соответствующие строкам на входе, и преобразует строки в токены. Токены представляют собой числовые представления строк и упрощают обработку. Это показано на рисунке 1.

По мере того, как lex находит идентификаторы во входном потоке, он вводит их в таблицу символов. Таблица символов может также содержать другую информацию, такую как тип данных (целочисленный или действительный) и расположение переменной в памяти. Все последующие ссылки на идентификаторы относятся к соответствующему индексу таблицы символов.

Yacc генерирует код C для синтаксического анализатора или парсера. Yacc использует правила грамматики, которые позволяют ему анализировать токены (лексемы), выдаваемые lex, и создавать синтаксическое дерево. Синтаксическое дерево накладывает иерархическую структуру на токены. Например, приоритет операций и ассоциативность очевидным образом учитываются в синтаксическом дереве.

На следующем шаге, шаге генерации кода, выполняется обход синтаксического дерева для генерации кода.

Некоторые компиляторы производят машинный код, а другие, как показано выше, выдают ассемблер.

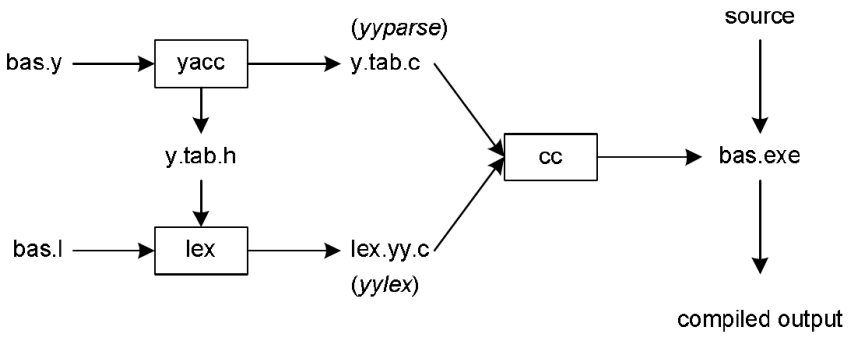

Рисунок 2 – Построение компилятора с помощью Lex/Yacc

На рис. 2 показаны соглашения об именах файлов, используемые lex и yacc. Предположим, что наша цель — написать компилятор BASIC. Во-первых, нам нужно указать все правила сопоставления с образцом для lex (bas.l) и правила грамматики для yacc (bas.y). Команды для создания нашего компилятора bas.exe представлены ниже:

```
yacc –d bas.y                  # create y.tab.h, y.tab.c
lex bas.l                      # create lex.yy.c
cc lex.yy.c y.tab.c –obas.exe  # compile/link
```

Yacc читает описания грамматики в bas.y и генерирует синтаксический анализатор, функцию yyparse, в файле y.tab.c. То, что включено в файл bas.y является объявлениями токенов. Опция –d заставляет yacc генерировать определения токенов и помещать их в файл y.tab.h. Lex читает описания шаблонов в bas.l, включает файл y.tab.h и генерирует лексический анализатор, функцию yylex, в файле lex.yy.c.

Наконец, лексический анализатор и синтаксический анализатор компилируются и компонуются вместе, образуя исполняемый файл bas.exe.

Для запуска компилятора мы из main( ) вызываем функцию yyparse( ). Функция yyparse( ) автоматически вызывает yylex( ) для получения каждого токена.

2 Lex

2.1 Теория

Первая фаза компилятора считывает входной источник и преобразует строки из источника в токены. Используя регулярные выражения, мы можем указать шаблоны для `lex`, которые позволят ему сканировать и сопоставлять строки во входных данных. У каждого шаблона в `lex` есть связанное с ним действие. Обычно действие возвращает токен, представляющий совпадающую строку, для последующего использования синтаксическим анализатором. Однако для начала мы просто напечатаем совпадающую строку, а не вернем значение токена. Мы можем сканировать идентификаторы, используя регулярное выражение

```
letter(letter|digit)*
```

Этот шаблон соответствует строке символов, которая начинается с одной буквы, за которой следует ноль или более букв или цифр. Этот пример прекрасно иллюстрирует операции, разрешенные в регулярных выражениях:

- повторение, выраженное оператором `'*'`;
- чередование, выраженное оператором `'|'`;
- конкатенация.

Любые выражения в форме регулярных выражений могут быть выражены в виде конечного автомата (FSA). Мы можем представить FSA, используя состояния и переходы между состояниями. Существует одно начальное состояние и одно или несколько конечных или принимающих состояний.

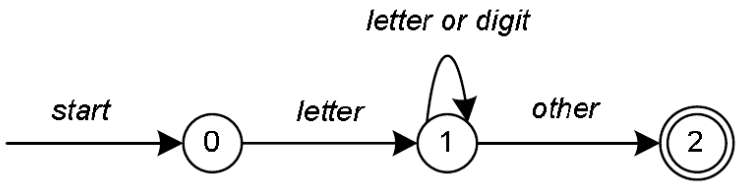

Рисунок 3 – Конечный автомат

На рисунке 3 состояние 0 — это начальное состояние, а состояние 2 — состояние принятия. По мере чтения символов мы совершаем переход из одного состояния в другое. Когда прочитана первая буква, мы переходим в состояние 1. Мы остаемся в состоянии 1, пока считываются другие буквы или цифры. Когда мы читаем символ, отличный от буквы или цифры, мы переходим в состояние 2, принимающее состояние.

Любой FSA может быть выражен в виде компьютерной программы. Например, наш автомат с тремя состояниями легко программируется:

```
start: goto state0

state0: read c
if c = letter goto state1
goto state0

state1: read c
if c = letter goto state1
if c = digit goto state1
goto state2
```

```
state2: accept string
```

Это метод, используемый lex. Регулярные выражения транслируются lex в компьютерную программу, имитирующую FSA. Используя следующий входной символ и текущее состояние, следующее состояние легко определяется путем индексации в созданной компьютером таблице состояний.

Теперь мы можем легко понять некоторые ограничения lex. Например, lex нельзя использовать для распознавания вложенных структур, таких как круглые скобки. Вложенные структуры обрабатываются путем использования стека. Всякий раз, когда мы встречаем '(', мы помещаем его в стек. Когда встречается ')', мы сопоставляем его с вершиной стека и извлекаем его из стека. Lex, однако, имеет только состояния и переходы между состояниями. Поскольку у него нет стека, он плохо подходит для разбора вложенных структур.

Yacc дополняет FSA стеком и может легко обрабатывать такие конструкции, как круглые скобки.

Важно использовать правильный инструмент для работы. Lex хорош в сопоставлении с образцом. Yacc подходит для более сложных задач.

2.2 Практика

Таблица 1. Примитивы сопоставления с образцом (Pattern Matching Primitives)

Metacharacter	**Matches**
.	любой символ кроме новой строки
\n	новая строка
*	ноль или более копий предыдущего выражения
+	одна или более копий предыдущего выражения
?	ноль или одна копия предыдущего выражения
^	начало строки
$	конец строки
a|b	a или b
(ab)+	одна или несколько копий ab (группировка)
"a+b"	литерально "a+b" (экранирование C тем не менее работает)
[]	класс символов

Таблица 2. Примеры сопоставления шаблонов

Expression	**Matches**
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
a(bc)?	a abc
[abc]	один из: a, b, c
[a-z]	любой символ из, a-z
[a\-z]	один из: a, -, z
[-az]	один из: -, a, z
[A-Za-z0-9]+	один или несколько буквенно-цифровых символов
[\t\n]+	пробел
[^ab]	все, кроме: a, b

[a^b]	один из: a, ^, b
[a|b]	один из: a, |, b
a|b	один из: a, b

Регулярные выражения в lex состоят из метасимволов (таблица 1). Примеры сопоставления с образцом показаны в Таблице 2. Внутри символьного класса обычные операторы теряют свое значение. В классе символов разрешены два оператора: дефис (`'-'`) и циркумфлекс (`'^'`). При использовании между двумя символами дефис представляет собой диапазон символов. Циркумфлекс, используемый в качестве первого символа, инвертирует выражение. Если два шаблона соответствуют одной и той же строке, побеждает самое длинное совпадение. Если оба совпадения имеют одинаковую длину, используется первый из перечисленных шаблонов.

Входные данные для Lex разделены на три раздела, при этом разделы отделяются %%. Лучше всего это видно на примере.

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Первый пример — самый короткий lex-файл:

```
%%
```

Ввод копируется в вывод по одному символу за раз. Первое вхождение %% всегда требуется, так как всегда должен быть раздел правил. Однако, если мы не указываем никаких правил, то действие по умолчанию — сопоставить все и скопировать в вывод. По умолчанию для ввода и вывода используются стандартный ввод и стандартный вывод соответственно. Вот тот же пример с явно закодированными значениями по умолчанию:

```
%%
    /* соответствует всему, кроме новой строки */
.   ECHO;
    /* соответствует новой строке */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

В разделе правил указаны два шаблона. Каждый шаблон должен начинаться с первого столбца.

Далее следует пробел (пробел, табуляция или новая строка) и необязательное действие, связанное с шаблоном. Действие может быть одним оператором C или несколькими операторами C, заключенными в фигурные скобки.

Все, что не начинается в первом столбце, дословно копируется в сгенерированный файл на C. Мы можем воспользоваться этим поведением, чтобы указать комментарии в нашем lex-файле. В этом примере есть два шаблона: `'.'` и `'\n'`, с действием `ECHO`, связанным с каждым из шаблонов. Несколько макросов и переменных предопределены lex. `ECHO` — это макрос, который записывает код, соответствующий шаблону. Это действие по умолчанию для любых несовпадающих строк. Обычно `ECHO` определяется как:

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

`yytext` — это указатель на совпадающую строку (оканчивающуюся нулем);
`yyleng` — длина совпавшей строки.
`yyout` — является выходным файлом и по умолчанию имеет значение `stdout`.

Функция `yywrap()` вызывается `lex`, когда вход исчерпан. Если вы закончили, верните 1, или 0, если требуется дополнительная обработка. Каждая программа на C требует наличия функции `main()`. В этом случае мы просто вызываем `yylex()`, основную точку входа для `lex`. Некоторые реализации `lex` включают в себя копии `main()` и `yywrap()` в библиотеке, что устраняет необходимость их явного кодирования. Вот почему наш первый пример, самая короткая lex-программа, работал правильно.

Таблица 3. Предопределенные переменные Lex

Имя	**Функция**
`int yylex(void)`	функция вызова лексисеского анализатора, возвращает токен
`char *yytext`	указатель на совпадающую строку
`yyleng`	длина совпадающей строки
`yylval`	значение, связанное с токеном
`int yywrap(void)`	завершение, вернуть 1, если сделано, 0, если не сделано
`FILE *yyout`	выходной файл
`FILE *yyin`	входной файл
`INITIAL`	начальное условие запуска
`BEGIN`	состояние переключателя условия запуска
`ECHO`	вывести совпадающую строку

Вот программа, которая вообще ничего не делает. Все входные данные совпадают, но ни с одним шаблоном не связано ни одного действия, поэтому выходных данных не будет.

```
%%
.
\n
```

В следующем примере к каждой строке файла добавляются номера строк. Некоторые реализации `lex` предопределяют и вычисляют `yylineno`. Входным файлом для `lex` является `yyin`, а по умолчанию используется стандартный ввод.

```
%{
    int yylineno;
%}
%%
```

```
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

Раздел определений состоит из замен, кода и начальных состояний. Код в разделе определений просто копируется как есть в начало сгенерированного файла на C и должен быть заключен в квадратные скобки с маркерами «%{» и «%}». Подстановки упрощают правила сопоставления с образцом. Например, мы можем определить цифры и буквы:

```
digit     [0-9]
letter    [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*    count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

Определяющий термин и связанное с ним выражение должны разделяться пробелом. Ссылки на подстановки в разделе правил заключены в фигурные скобки (`{letter}`), чтобы отличить их от литералов. Если есть совпадение в разделе правил, выполняется соответствующий код C.

Ниже сканер, который подсчитывает количество символов, слов и строк в файле (аналог Unix `wc`):

```
%{
    int nchar, nword, nline;
%}
%%
\n        { nline++; nchar++; }
[^ \t\n]+ { nword++, nchar += yyleng; }
.         { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

3 Yacc

3.1 Теория

Грамматика для `yacc` описана с использованием варианта формы Бэкуса-Наура (BNF). Эта техника была впервые предложена Джоном Бэкусом и Питером Науром и использовалась для описания ALGOL60. Грамматика BNF может использоваться для выражения контекстно-свободных языков. Большинство конструкций современных языков программирования могут быть представлены в BNF. Например, грамматика для выражения, которое умножает и складывает числа, выглядит так:

```
E -> E + E
E -> E * E
E -> id
```

Здесь определены три правила вывода (продукции). Термины, которые появляются в левой части (`lhs`) продукции, такие как `E` (выражение), являются нетерминальными. Такие термины, как `id` (идентификатор), являются терминалами (токенами, возвращаемыми `lex`) и появляются только в правой части (`rhs`) правила. Эта грамматика указывает, что выражение может быть суммой двух выражений, произведением двух выражений или идентификатором. Мы можем использовать эту грамматику для генерации выражений:

```
E -> E * E      (r2)
  -> E * z      (r3)
  -> E + E * z  (r1)
  -> E + y * z  (r3)
  -> x + y * z  (r3)
```

На каждом шаге мы раскрываем терм, заменяя левые части на соответствующие правые части, согласно правилу вывода.

Цифры справа указывают, какое правило применяется. Чтобы разобрать выражение, нам в действительности нужно сделать обратную операцию. Вместо того, чтобы начинать с одного нетерминала (стартового символа) и генерировать выражение из грамматики, нам нужно сократить выражение до одного нетерминала. Эта процедура известна как *разбор снизу вверх* или *сдвиг-сокращение*, при этом для хранения используется стек.

Вот тот же вывод, но в обратном порядке:

```
1  . x + y * z    shift
2  x . + y * z    reduce(r3)
3  E . + y * z    shift
4  E + . y * z    shift
5  E + y . * z    reduce(r3)
6  E + E . * z    shift
7  E + E * . z    shift
8  E + E * z .    reduce(r3)
9  E + E * E .    reduce(r2)    emit multiply
10 E + E .        reduce(r1)    emit add
11 E .            accept
```

Термины слева от точки находятся в стеке, а остальные входные данные — справа от точки. Мы начинаем с перемещения токенов в стек. Когда вершина стека совпадает с правой частью продукции, мы заменяем совпадающие токены в стеке на левые позиции продукции.

Концептуально совпавшие токены правой части выталкиваются из стека, а левые позиции продукции помещаются в стек.

Совпадающие токены известны как «описание», и мы сокращаем «описание» до левых частей продукции. Этот процесс продолжается до тех пор, пока мы не переместим все входные данные в стек, и в стеке останется только начальный нетерминал.

На шаге 1 мы перемещаем `x` в стек. Шаг 2 применяет к стеку правило `r3`, заменяя `x` на `E`. Мы продолжаем сдвиг и сокращение, пока в стеке не останется единственный нетерминал, начальный символ. На шаге 9, когда мы сокращаем по правилу `r2`, мы выдаем команду умножения. Точно так же на шаге 10 выдается инструкция `add`. Таким образом, умножение имеет более высокий приоритет, чем сложение.

Рассмотрим, однако, сдвиг на шаге 6. Вместо сдвига мы могли бы выполнить сокращение, применив правило `r1`. Это привело бы к тому, что сложение будет иметь более высокий приоритет, чем умножение. Это известно как конфликт сдвига-сокращения. Наша грамматика неоднозначна, поскольку существует более одного возможного вывода, который даст выражение. В этом случае затрагивается приоритет оператора. В качестве другого примера, ассоциативность в правиле

```
E -> E + E
```

неоднозначна, поскольку мы можем рекурсивно выполнять рекурсию слева или справа. Чтобы исправить ситуацию, мы могли бы переписать грамматику или снабдить `yacc` директивами, указывающими, какой оператор имеет приоритет.

Последний метод проще и будет продемонстрирован в практическом разделе.

Следующая грамматика имеет конфликт сокращения-сокращения. Имея `id` в стеке, мы можем сократить его до `T` или до `E`.

```
E -> T
E -> id
T -> id
```

При возникновении конфликта Yacc выполняет действие по умолчанию. Для конфликтов сдвиг-сокращение `yacc` будет выполнять сдвиг. Для конфликтов сокращение-сокращение будет использоваться первое правило в списке. Он также выдает предупреждающее сообщение всякий раз, когда существует конфликт. Предупреждения можно подавить, сделав грамматику однозначной. Несколько методов устранения неоднозначности будут представлены в последующих разделах.

3.2 Практика, часть I

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

Входные данные для `yacc` разделены на три части. Раздел определений состоит из объявлений токенов и кода на C, заключенного в скобки `"%{"` и `"%}"`. Грамматика BNF помещается в раздел правил, а пользовательские подпрограммы добавляются в раздел подпрограмм.

Лучше всего это можно проиллюстрировать, создав небольшой калькулятор, который может складывать и вычитать числа. Мы начнем с изучения связи между `lex` и `yacc`. Вот раздел определений для входного файла `yacc`:

```
%token INTEGER
```

Это определение объявляет токен INTEGER. Когда мы запускаем yacc, он генерирует парсер в файле y.tab.c, а также создает включаемый файл y.tab.h:

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define INTEGER 258
extern YYSTYPE yylval;
```

Lex включает этот файл и использует определения для значений токенов. Для получения токенов yacc вызывает yylex( ). Функция yylex( ) имеет возвращаемый тип int и возвращает значение токена. Значения, связанные с токеном, возвращаются lex в переменной yylval. Например,

```
[0-9]+   {
             yylval = atoi(yytext);
             return INTEGER;
         }
```

сохранит значение целого числа в yylval и вернет токен INTEGER в yacc. Тип yylval определяется YYSTYPE. Поскольку тип по умолчанию целочисленный, в данном случае это работает хорошо.

Значения токена 0–255 зарезервированы для значений символов. Например, если у вас есть такое правило, как

```
[-+]    return *yytext;     /* return operator */
```

возвращается символьное значение для минуса или плюса. Обратите внимание, что мы поставили знак минус первым, чтобы его нельзя было принять за обозначение диапазона. Сгенерированные значения токена обычно начинаются с 258, так как lex резервирует несколько значений для обработки конца файла и ошибок. Вот полная спецификация ввода lex для нашего калькулятора:

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}

%%

[0-9]+       {
                 yylval = atoi(yytext);
                 return INTEGER;
             }
[-+\n]       return *yytext;
[ \t]        ; /* skip whitespace */
.            yyerror("invalid character");
```

```
%%
int yywrap(void) {
    return 1;
}
```

Внутри yacc поддерживает в памяти два стека; стек синтаксического анализа и стек значений. Стек синтаксического анализа содержит терминалы и нетерминалы и представляет текущее состояние синтаксического анализа. Стек значений представляет собой массив элементов YYSTYPE и связывает значение с каждым элементом в стеке синтаксического анализа. Например, когда lex возвращает токен INTEGER, yacc перемещает этот токен в стек синтаксического анализа. В то же время соответствующий yylval смещается в стек значений.

Стеки синтаксического анализа и значений всегда синхронизированы, поэтому легко найти значение, связанное с токеном в стеке. Ниже спецификация ввода yacc для нашего калькулятора:

```
%{
    int yylex(void);
    void yyerror(char *);
%}

%token INTEGER

%%

program: program expr '\n'          { printf("%d\n", $2); }
       |
       ;

expr: INTEGER                       { $$ = $1; }
    | expr '+' expr                 { $$ = $1 + $3; }
    | expr '-' expr                 { $$ = $1 - $3; }
    ;

%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void) {
    yyparse();
    return 0;
}
```

Раздел правил напоминает грамматику БНФ, рассмотренную ранее. Левая часть продукции, или нетерминал, вводится с выравниванием по левому краю, за которым следует двоеточие. Далее следует правая часть производства. Действия, связанные с правилом, вводятся в фигурных скобках.

Используя левую рекурсию, мы указали, что программа состоит из нуля или более выражений.

Каждое выражение заканчивается символом новой строки. При обнаружении новой строки мы печатаем значение выражения. Когда мы применяем правило

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

мы заменяем правую часть продукции в стеке синтаксического анализа левой частью той же продукции. В этом случае мы вынимаем из стека «expr '+' expr» и помещаем обратно в стек «expr». Мы сократили стек, вытащив из стека три терма и вернули один терм обратно. Мы можем ссылаться на позиции в стеке значений в нашем коде C, указав «$1» для первого члена в правой части продукции, «$2» для второго и так далее. «$$» обозначает вершину стека после сокращения. Вышеприведенное действие добавляет значение, связанное с двумя выражениями, извлекает три условия из стека значений и возвращает одну сумму. Таким образом, стеки синтаксического анализа и значения остаются синхронизированными.

Числовые значения изначально вводятся в стек, когда мы преобразуем их из INTEGER в expr. После перемещения INTEGER в стек применяем правило

```
expr: INTEGER     { $$ = $1; }
```

Токен INTEGER извлекается из стека синтаксического анализа, после чего в него помещается expr. Для стека значений мы извлекаем целочисленное значение из стека, а затем снова помещаем его обратно. Другими словами, мы ничего не делаем. На самом деле это действие по умолчанию, и его не нужно указывать. Наконец, когда встречается новая строка, печатается значение, связанное с expr.

В случае синтаксических ошибок yacc вызывает предоставленную пользователем функцию yyerror(). Если вам нужно изменить интерфейс на yyerror(), вы можете изменить соответствующий файл, который включается в состав yacc, в соответствии с вашими потребностями. Последняя функция в нашей спецификации yacc — это main()... если вам интересно, где она находится. Этот пример по-прежнему имеет неоднозначную грамматику. Yacc будет выдавать предупреждения о сдвиге-сокращении, но по-прежнему будет обрабатывать грамматику, используя сдвиг в качестве операции по умолчанию.

3.3 Практика, часть 2

В этом разделе мы расширим калькулятор из предыдущего раздела, добавив некоторые новые функции. Новые функции включают арифметические операторы умножения и деления. Круглые скобки могут использоваться для переопределения приоритета оператора, а в операторах присваивания могут быть указаны односимвольные переменные. Ниже показаны пример ввода и вывода калькулятора:

```
user: 3 * (4 + 5)
calc: 27
user: x = 3 * (5 + 4)
user: y = 5
user: x
calc: 27
user: y
calc: 5
user: x + 2*y
calc: 37
```

Лексический анализатор возвращает токены VARIABLE и INTEGER. Для переменных yylval указывает индекс в sym, нашей таблицы символов. Для этой программы sym просто

хранит значение связанной переменной. Когда возвращаются токены INTEGER, yylval содержит отсканированное число. Вот входная спецификация для lex:

```
%{
    #include <stdlib.h>
    void yyerror(char *);
    #include "y.tab.h"
%}

%%
    /* variables */
[a-z]     {
              yylval = *yytext – 'a'; // ASCII???
              return VARIABLE;
          }

    /* integers */
[0-9]+    {
              yylval = atoi(yytext);
              return INTEGER;
          }

    /* operators */
[-+()=/*\n]   { return *yytext; }

    /* skip whitespace */
[ \t]     ;
.
    /* anything else is an error */
    yyerror("invalid character");

%%
int yywrap(void) {
    return 1;
}
```

Спецификация ввода для yacc приведена ниже. Токены для INTEGER и VARIABLE используются yacc для создания #define в y.tab.h , который будет использоваться в lex. Далее следуют определения арифметических операторов. Мы можем указать %left для левой ассоциативности или %right для правой ассоциативности[1]. Последнее указанное определение имеет наивысший приоритет. Таким образом, умножение и деление имеют более высокий

1 Ассоциативность определяет, какая часть выражения должна быть вычислена первой. Например, результатом выражения 42 - 15 - 6 может быть 21 или 33 в зависимости от того, какая ассоциативность будет применяться для оператора «-»: левая или правая. Оператор «-» имеет левую ассоциативность, т.е. сначала вычисляется 42 - 15, а затем из результата вычитается 6. Если бы он имел правую ассоциативность, сначала вычислялась бы правая часть выражения (15 - 6), а затем результат вычитался бы из 42. Все бинарные операторы (операторы с двумя операндами), кроме операторов присваивания, — лево-ассоциативные, т.е. они обрабатывают выражения слева направо. Таким образом, a + Ь + c — то же, что и (a + b) + c. Операторы присваивания и условные операторы — право-ассоциативные, т.е. обрабатывают выражения справа налево. Иначе говоря, a=b=c эквивалентно a = (b = c).

приоритет, чем сложение и вычитание. Все четыре оператора левоассоциативны. Используя эту простую технику, мы можем устранить неоднозначность нашей грамматики.

```
%token INTEGER VARIABLE
%left '+' '-'
%left '*' '/'
%{
    void yyerror(char *);
    int yylex(void);
    int sym[26];
%}

%%

program: program statement '\n'
       |
       ;

statement: expr                  { printf("%d\n", $1); }
         | VARIABLE '=' expr     { sym[$1] = $3; }
         ;

expr: INTEGER
    | VARIABLE              { $$ = sym[$1]; }
    | expr '+' expr         { $$ = $1 + $3; }
    | expr '-' expr         { $$ = $1 - $3; }
    | expr '*' expr         { $$ = $1 * $3; }
    | expr '/' expr         { $$ = $1 / $3; }
    | '(' expr ')'          { $$ = $2; }
    ;

%%

void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int main(void) {
    yyparse();
    return 0;
}
```

4 Калькулятор

(продолжение следует)