Санкт-Петербургский политехнический университет Петра Великого

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №2**

**Дисциплина**: Транслирующие системы

**Тема**: «Программирование синтаксического разбора на языке

yacc»

Выполнил студент гр. 3530901/70203 И.Д. Иванов

(подпись)

Преподаватель \_\_\_\_\_\_\_\_\_\_\_ В.Н. Цыган

(подпись)

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_ 2019 г.

Санкт-Петербург

2019

1. **Цель работы**

Изучение и получение навыков применения утилит yacc и lex для генерации синтаксических анализаторов.

1. **Программа работы**

1. Ознакомиться с работой программы YACC.

2. Протестировать примеры.

3. Выполнить индивидуальное задание.

1. **Выполнение работы**

***Теоретическая информация***

Yacc ("yet another compiler compiler'') предназначен для генерации программ синтаксического анализа. Основу исходной yacc-программы составляет грамматика - набор правил, описывающих синтаксическую структуру ввода, и соответствующих действий в виде фрагментов на языке C.

***Структура и синтаксис yacc-программы***

Форма исходного текста yacc-программы полностью совпадает с формой для lex-программы:

*определения*

%%

*правила*

%%

*процедуры пользователя*

***Особенности секции определений***

Объявления, специфические для yacc-программы, следующие:

- объявление объединенного типа (поддерживает различные типы сопутствующего значения), в форме:

*%union*

*{*

*type\_1 name\_1;*

*...*

*type\_n name\_n;*

*}*

- объявление стартового символа, в форме:

*%start start\_sym*

- объявления лексем, в форме:

*%token SYM1 SYM2 ...*

-либо (с уточнением типа сопутствующего значения) в форме:

*%token <type\_k> SYM1 SYM2 ...*

- объявление типа сопутствующего значения для нетерминального символа, в форме:

*%type <type\_k> sym1 ...*

***Формат правил и действий***

Правила записываются в форме:

*sym : SEQ ;*

где *sym* - имя нетерминального символа, *SEQ* - определение этого символа в виде последовательности имен символов.

* 1. **Взаимодействие модулей lex и yacc**

Взаимодействие модулей, написанных на lex и yacc, поясним на примере программы из каталога date/v1. Эта программа только проверяет структуру ввода

Листинг 4.1 (v1.y). Простейший синтаксический анализатор на языке yacc

%token NUMBER MONTH

%start date

%%

date : MONTH NUMBER NUMBER

%%

В этой спецификации определены лексемы NUMBER и MONTH и задан стартовый символ — date. (Стартовый символ — это один из нетерминальных символов, обнаружение которого представляет цель синтаксического разбора.) Затем следует определение date через три терминальных символа. Точка с запятой в конце определения, отделяющая правила друг от друга, в примере отсутствует, т. к. правило здесь единственное.

Лексический анализ сводится к выявлению чисел и строк с названиями месяцев, что задано следующей lex-спецификацией.

Листинг 4.2 (v1.l). Модуль на языке lex для синтаксического анализатора

%{

#include "y.tab.h"

%}

%%

[0-9]+ { return NUMBER; }

jan |

…

dec { return MONTH; }

[ \t\n] ;

. { return 0; }

%%

#ifndef yywrap

int yywrap () { return 1; }

#endif

Если на входе появится литера, не относящаяся к числам и названиям месяцев и не являющаяся разделителем (пробелом, табуляцией или новой строкой), функция yylex вернет ноль — признак конца ввода для yacc-модуля. Имена NUMBER и MONTH — это константы из файла y.tab.h, полученного в результате трансляции v1.y.

Пример 1: Проверка структуры ввода даты.

masdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v1$ ./a.out

asasas

?-syntax error

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v1$ ./a.out

sep 10 10!

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v1$ ./a.out

apr 2020 2020!

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v1$ ./a.out

apr 20 20 20!

?-syntax error

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v1$ ./a.out

september 10 10!

?-syntax error

* 1. **Литеральные лексемы**

Из заголовочного файла y.tab.h видно, что коды терминальных символов, определенных при помощи ключевого слова %token, начинаются с 257. Код 0 зарезервирован для признака конца ввода, а коды от 1 до 256 — для литеральных лексем, или "литералов''. Использование литералов иллюстрируется примером из каталога \_date/v2. В определении date появилась запятая в одиночных кавычках — это и есть литерал, то есть терминальный символ, код которого равен ASCII-коду запятой.

Листинг 4.3.1 (v2.y). Литерал в определении нетерминального символа

date : MONTH NUMBER ',' NUMBER

В предыдущем примере функция yylex лексического анализатора при чтении запятой возвращала результат 0. Теперь в lex-модуль добавлено правило, которое в этом случае возвращает код запятой.

Листинг 4.4.1 (v2.l). Передача литерала из lex-модуля

"," { return yytext[0];}

Пример 2.1: C использованием разделителя ‘,’.

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v2$ ./a.out

apr 13, 22!

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v2$ ./a.out

apr 13 22!

?-syntax error

Листинг 4.3.2 (v2.l). Литерал в определении нетерминального символа (с использованием ‘;’)

date : MONTH NUMBER ',' NUMBER | MONTH NUMBER ';' NUMBER

Листинг 4.4.1 (v2.l). Передача литерала из lex-модуля (с использованием ‘;’)

"," { return yytext[0];}

";" { return yytext[0];}

Пример 2.2: C использованием разделителя ‘,’ и ‘;’.

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v2$ ./a.out

sep 20 ; 20!

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v2$ ./a.out

apr 10, 1010!

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v2$ ./a.out

sep 20 20!

?-syntax error

* 1. **Сопутствующие значения**

Если бы лексический анализатор вычислял величины месяцев и чисел и передавал их вместе с кодом лексемы, то синтаксический анализатор мог бы выводить дату и проверять ее допустимость. В примере \_date/v3 эти возможности использованы.

Листинг 4.5 (v3.l). Задание типа и величины сопутствующего значения

%{

#include <stdlib.h>

#include "y.tab.h"

#define YYSTYPE int

extern YYSTYPE yylval;

%}

%%

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

jan { yylval = 0; return MONTH; }

…

dec { yylval = 11; return MONTH; }

"," { return yytext[0]; }

[ \t\n] ;

. { return 0; }

%%

#ifndef yywrap

int yywrap () { return 1; }

#endif

Здесь добавлено определение типа сопутствующего значения YYSTYPE и ссылка на внешнюю переменную yylval. Лексеме NUMBER сопутствует значение десятичного числа, а лексеме MONTH — номер месяца в диапазоне [0..11].

Синтаксический анализатор использует сопутствующие значения следующим образом. Когда yylex возвращает управление yyparse, величина yylval записывается в стек значений; так продолжается, пока правило не будет применено. Доступ к этим значениям открыт через псевдопеременные $n. В начале кадра стека оставлено место для сопутствующего значения определяемого символа (псевдопеременная $$).

Листинг 4.6 (v3а.у). Доступ к семантическим значениям

%%

date : MONTH NUMBER ',' NUMBER

{ printf("m-d-y: %2u-%2u-%4u\n", $1+1, $2, $4); }

%%

Семантическое значение первого символа доступно через $1 — это номер месяца от 0 до 11, а $2 и $4 — значения дня и года. Литерал ',' в третьей позиции тоже считается символом; у него тоже есть значение, доступное через $3 — но там сейчас случайная величина, так как функция yylex, обнаружив запятую, в yylval ничего не записала.

Пример 3.a: Доступ к семантическим значениям

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/a$ ./a.out

sep 10, 2010!

m-d-y: 9-10-2010

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/a$ ./a.out

apr 12, 2015!

m-d-y: 4-12-2015

Листинг 4.7 (v3b.у). Проверка даты и вывод количества дней от 1970 г.

%{

long abs\_date (int, int, int); /\* month (0-11), day, year \*/

%}

%token NUMBER MONTH

%start date

%%

date : MONTH NUMBER ',' NUMBER

{ printf("%ld\n", abs\_date($1, $2, $4)); }

%%

Проверка даты и вычисление количества дней, прошедших от 01/01/1970 выполняется в функции abs\_date (см. модуль abs\_date.c) при помощи библиотечной функции mktime. Для проверки даты пригодилось умение mktime исправлять неправильную дату. Значение числа дней можно было бы использовать в качестве сопутствующего значения для символа date.

Пример 3.b: Проверка даты и вывод количества дней от 1970 г.

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/b$ ./a.out

oct 10, 2019!

18178

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/b$ ./a.out

jan 1, 1971!

364

Листинг 4.8 (v3с.у). Семантическое значение date и вычисление разницы между датами

%{

long abs\_date (int, int, int); /\* month (0-11), day, year \*/

%}

%token NUMBER MONTH

%start between

%%

date : MONTH NUMBER ',' NUMBER

{ $$ = abs\_date($1, $2, $4); }

between : date '-' date

{ printf("%ld\n", $1 - $3); }

%%

Семантическое значение date формируется в конце правила для date и используется в правиле для between. Семантическое значение between не формируется за ненадобностью.

Пример 3c: Семантическое значение date и вычисление разницы между датами

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/c$ ./a.out

aug 10, 2019 - july 13, 2010!

3315

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v3/c$ ./a.out

sep 3, 2018 - sep 5, 2018!

-2

* 1. **Сопутствующие значения разных типов**

Иногда требуется возвращать сопутствующие значения разных типов, например, int и char\*, притом, что канал передачи значений от yylex к yyparse единственный — переменная yylval. В этом случае используется объединение (union). Рассмотрим примеры из \_date/v4.

Листинг 4.9. Определение сопутствующего значения нескольких типов

%union

{

int ival;

char \* text;

};

Выполним пример в каталоге v4/a. Трансляция yacc-модуля не прошла, поскольку в нем не задана информация о типе $1, $2 и $4 — ведь теперь у сопутствующего значения не один тип, а два. Тип можно указать при обращении к $-переменной.

Листинг 4.10 (v4b.y). Явное указание типа при обращении к $-переменной

date : MONTH NUMBER ',' NUMBER

{ print($<text>1, $<ival>2, $<ival>4); }

Тип может быть указан и при объявлении терминального символа, тогда при обращении к $-переменным уточнять его не придется, и этот вариант — предпочтительный.

Листинг 4.11 (v4c.y). Задание типа при объявлении символа

%token <ival> NUMBER

%token <text> MONTH

%start date

%%

date : MONTH NUMBER ',' NUMBER

{ print($1, $2, $4); }

%%

Листинг 4.12 (v4.l). Формирование сопутствующего значения в lex-модуле

[0-9]+ { yylval.ival = atoi(yytext); return NUMBER; }

jan |

…

dec { yylval.text = strdup(yytext); return MONTH; }

Пример 4. Определение сопутствующего значения нескольких типов

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v4/c$ ./a.out

jan 1, 2000!

1-jan-2000

Вернемся к примеру, где подсчитывается количество дней между двумя датами. В нем сопутствующие значения должны быть двух типов:

int — для месяца, дня и года;

long — для нетерминального символа date (количество дней от 01/01/1970)

Листинг 4.13 (v5.y). Вычисление количества дней между двумя датами

%union

{

int ival;

long lval;

};

%token <ival> NUMBER MONTH

%type <lval> date

%start between

%%

date : MONTH NUMBER ',' NUMBER

{ $$ = abs\_date($1, $2, $4); }

between : date '-' date

{ printf("%ld\n", $1 - $3); }

%%

Пример 5. Вычисление количества дней между двумя датами

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v5$ ./a.out

aug 10, 2019 - july 13, 2010!

3315

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/\_date/v5$ ./a.out

sep 3, 2018 - sep 5, 2018!

-2

* 1. **Рекурсивные правила**

Обратимся к программе в каталоге list/v0. Она разбирает список чисел, разделенных запятыми, и выводит число элементов в списке.

Листинг 4.14 (v0/c1.l). Лексический анализатор для разбора списка чисел

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

(.|\n) return yytext[0];

%%

Этот лексический анализатор распознает цепочки десятичных цифр, вычисляет (при помощи библиотечной функции atoi) соответствующие им числовые значения, возвращая их синтаксическому анализатору через переменную yylval вместе с лексемой NUM. Все прочие литеры лексический анализатор возвращает в yacc-модуль в виде литералов.

Листинг 4.15 (v0/c1.у). Синтаксический анализатор для разбора списка чисел

%%

\_\_list: \_list { printf("No. of items: %d\n", $1); }

\_list: /\* empty \*/ { $$ = 0; /\* size is 0 \*/ }

| list /\* not empty, $$ == $1 by default \*/

;

list: NUM { $$ = 1; } /\* size := 1 \*/

| NUM ',' list { $$ = $3 + 1; } /\* size := size of sublist + 1 \*/

;

%%

Пример 6. Анализатор для разбора списка чисел

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/list/v0$ ./c1.out

10,101010,3,345,001

No. of items: 5

?-syntax error

Получено сообщение “?-syntax error”. Чтобы выяснить причину, включим трассировку: найдём в zz.c определение yydebug и исправим 0 на 1, затем повторим трансляцию.

Пример 7. Анализатор для разбора списка чисел с yydebug=1

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/list/v0/1$ ./a.out

Starting parse

Entering state 0

Reading a token: 1,2,3

Next token is token NUM ()

Shifting token NUM ()

Entering state 1

Reading a token: Next token is token ',' ()

Shifting token ',' ()

Entering state 5

Reading a token: Next token is token NUM ()

Shifting token NUM ()

Entering state 1

Reading a token: Next token is token ',' ()

Shifting token ',' ()

Entering state 5

Reading a token: Next token is token NUM ()

Shifting token NUM ()

Entering state 1

Reading a token: Next token is token $undefined ()

Reducing stack by rule 4 (line 11):

$1 = token NUM ()

-> $$ = nterm list ()

Stack now 0 1 5 1 5

Entering state 7

Reducing stack by rule 5 (line 12):

$1 = token NUM ()

$2 = token ',' ()

$3 = nterm list ()

-> $$ = nterm list ()

Stack now 0 1 5

Entering state 7

Reducing stack by rule 5 (line 12):

$1 = token NUM ()

$2 = token ',' ()

$3 = nterm list ()

-> $$ = nterm list ()

Stack now 0

Entering state 4

Reducing stack by rule 3 (line 8):

$1 = nterm list ()

-> $$ = nterm \_list ()

Stack now 0

Entering state 3

Reducing stack by rule 1 (line 5):

$1 = nterm \_list ()

No. of items: 3

-> $$ = nterm \_\_list ()

Stack now 0

Entering state 2

Next token is token $undefined ()

?-syntax error

Error: popping nterm \_\_list ()

Stack now 0

Cleanup: discarding lookahead token $undefined ()

Stack now 0

В строках вывода можно найти номера состояний конечного автомата при синтаксическом разборе. Что значат эти номера и состояния, можно выяснить в файле y.output, полученном при трансляции yacc-модуля. Файл y.output для рассматриваемого примера:

Grammar

0 $accept: \_\_list $end

1 \_\_list: \_list

2 \_list: %empty

3 | list

4 list: NUM

5 | NUM ',' list

Terminals, with rules where they appear

$end (0) 0

',' (44) 5

error (256)

NUM (258) 4 5

Nonterminals, with rules where they appear

$accept (5)

on left: 0

\_\_list (6)

on left: 1

on right: 0

\_list (7)

on left: 2 3

on right: 1

list (8)

on left: 4 5

on right: 3 5

State 0

0 $accept: . \_\_list $end

NUM shift, and go to state 1

$default reduce using rule 2 (\_list)

\_\_list go to state 2

\_list go to state 3

list go to state 4

State 1

4 list: NUM .

5 | NUM . ',' list

',' shift, and go to state 5

$default reduce using rule 4 (list)

State 2

0 $accept: \_\_list . $end

$end shift, and go to state 6

State 3

1 \_\_list: \_list .

$default reduce using rule 1 (\_\_list)

State 4

3 \_list: list .

$default reduce using rule 3 (\_list)

State 5

5 list: NUM ',' . list

NUM shift, and go to state 1

list go to state 7

State 6

0 $accept: \_\_list $end .

$default accept

State 7

5 list: NUM ',' list .

$default reduce using rule 5 (list)

Вернемся к трассе программы при вводе 1,2,3. Программа работает нормально до получения символа 10, недопустимого в состоянии 1. Код 10, по таблице ASCII, означает конец строки — литерал '\n'.

Литерал '\n' пришел из lex-модуля. Исправить ситуацию можно двумя способами

Листинг 4.16 (v0/c2.l). Удаление '\n' при лексическом разборе

%%

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

\n ;

. return yytext[0];

%%

Листинг 4.17 (v0/c2.у). Включение '\n' в синтаксический разбор

\_\_list: \_list '\n' { printf("No. of items: %d\n", $1); }

Проверим эти варианты, собрав программу в сочетаниях: c1.l + с2.y и c2.l + c1.y.

Пример 8. Лексический анализатор для разбора списка чисел+ Включение '\n' в синтаксический разбор (c1.l + с2.y)

|  |  |
| --- | --- |
| asdf@BigAwesomeTurtle:~/Documents/yacc/  works/yacc/list/v0/2$ ./a.out  Starting parse  Entering state 0  Reading a token: 1,2,3  Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Next token is token ',' ()  Shifting token ',' ()  Entering state 5  Reading a token: Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Next token is token ',' ()  Shifting token ',' ()  Entering state 5  Reading a token: Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Next token is token '\n' ()  Reducing stack by rule 4 (line 11):  $1 = token NUM ()  -> $$ = nterm list ()  Stack now 0 1 5 1 5  Entering state 8 | Reducing stack by rule 5 (line 12):  $1 = token NUM ()  $2 = token ',' ()  $3 = nterm list ()  -> $$ = nterm list ()  Stack now 0 1 5  Entering state 8  Reducing stack by rule 5 (line 12):  $1 = token NUM ()  $2 = token ',' ()  $3 = nterm list ()  -> $$ = nterm list ()  Stack now 0  Entering state 4  Reducing stack by rule 3 (line 8):  $1 = nterm list ()  -> $$ = nterm \_list ()  Stack now 0  Entering state 3  Next token is token '\n' ()  Shifting token '\n' ()  Entering state 7  Reducing stack by rule 1 (line 5):  $1 = nterm \_list ()  $2 = token '\n' ()  **No. of items: 3**  -> $$ = nterm \_\_list ()  Stack now 0  Entering state 2 |

При включении “\n” в синтаксический разбор программа работает корректно.

Пример 9. Удаление '\n' при лексическом разборе + Синтаксический анализатор для разбора списка чисел (c2.l + c1.y)

|  |  |
| --- | --- |
| asdf@BigAwesomeTurtle:~/Documents/yacc  /works/yacc/list/v0/3$ ./a.out <test.in  Starting parse  Entering state 0  Reading a token: Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Next token is token ',' ()  Shifting token ',' ()  Entering state 5  Reading a token: Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Next token is token ',' ()  Shifting token ',' ()  Entering state 5  Reading a token: Next token is token NUM ()  Shifting token NUM ()  Entering state 1  Reading a token: Now at end of input.  Reducing stack by rule 4 (line 11):  $1 = token NUM ()  -> $$ = nterm list ()  Stack now 0 1 5 1 5  Entering state 7  Reducing stack by rule 5 (line 12): | $1 = token NUM ()  $2 = token ',' ()  $3 = nterm list ()  -> $$ = nterm list ()  Stack now 0 1 5  Entering state 7  Reducing stack by rule 5 (line 12):  $1 = token NUM ()  $2 = token ',' ()  $3 = nterm list ()  -> $$ = nterm list ()  Stack now 0  Entering state 4  Reducing stack by rule 3 (line 8):  $1 = nterm list ()  -> $$ = nterm \_list ()  Stack now 0  Entering state 3  Reducing stack by rule 1 (line 5):  $1 = nterm \_list ()  **No. of items: 3**  -> $$ = nterm \_\_list ()  Stack now 0  Entering state 2  Now at end of input.  Shifting token $end ()  Entering state 6  Stack now 0 2 6  Cleanup: popping token $end ()  Cleanup: popping nterm \_\_list () |

При удалении “\n” при лексическом разборе, программа также работает корректно.

Листинг 4.18 (v0/c3.l). Удаление разделителей при лексическом разборе

%%

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

[ \t\n]+ ;

. return yytext[0];

%%

Пример 10. Удаление разделителей при лексическом разборе

Содержимое файла test.in:

1, 2, 3

Результат работы программы:

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/list/v0/4$ ./a.out <test.in

No. of items: 3

Листинг 4.19 (v1/c2.y). Использование леворекурсивного правила

%token NUM

%start \_\_list

%%

\_\_list: \_list

\_list: /\* empty \*/ { $$ = 0; }

| list

;

list: NUM { $$ = 1; print($$, $1, 1); }

| list

','

NUM { $$ = $1 + 1; print($$, $3, 2); }

;

%%

print (int len, int val, int rule)

{

printf("%d: %d (rule %d)\n", len, val, rule) ;

}

Листинг 4.20 (v1/c1.y). Использование праворекурсивного правила

%token NUM

%start \_\_list

%%

\_\_list: \_list

\_list: /\* empty \*/ { $$ = 0; }

| list

;

list: NUM { $$ = 1; print($$, $1, 1); }

| NUM

','

list { $$ = $1 + 1; print($$, $3, 2); }

;

%%

print (int len, int val, int rule)

{

printf("%d: %d (rule %d)\n", len, val, rule) ;

}

Пример 11. Левая и правая рекурсии

Содержимое файла test.in:

1,2,3,4,5,6,7,8,9,10,20,30,40,50,60,70,80,90,100

Результат работы при использовании файла c1.y:

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/list/v1/1$ ./a.out < test.in

1: 100 (rule 1)

2: 90 (rule 2)

3: 80 (rule 2)

4: 70 (rule 2)

5: 60 (rule 2)

6: 50 (rule 2)

7: 40 (rule 2)

8: 30 (rule 2)

9: 20 (rule 2)

10: 10 (rule 2)

11: 9 (rule 2)

12: 8 (rule 2)

13: 7 (rule 2)

14: 6 (rule 2)

15: 5 (rule 2)

16: 4 (rule 2)

17: 3 (rule 2)

18: 2 (rule 2)

19: 1 (rule 2)

Результат работы при использовании файла c2.y:

asdf@BigAwesomeTurtle:~/Documents/yacc/works/yacc/list/v1$ ./a.out < test.in

1: 1 (rule 1)

2: 2 (rule 2)

3: 3 (rule 2)

4: 4 (rule 2)

5: 5 (rule 2)

6: 6 (rule 2)

7: 7 (rule 2)

8: 8 (rule 2)

9: 9 (rule 2)

10: 10 (rule 2)

11: 20 (rule 2)

12: 30 (rule 2)

13: 40 (rule 2)

14: 50 (rule 2)

15: 60 (rule 2)

16: 70 (rule 2)

17: 80 (rule 2)

18: 90 (rule 2)

19: 100 (rule 2)

При использовании праворекурсивного правила строка считывается справа налево, а при использовании леворекурсивного правила-наоборот.

**3.6. Индивидуальное задание**

1) Модифицировать исходную программу таким образом, чтобы она была способна принять на вход произвольное количество строк, с сохранением семантических функций для каждой строки.

2) Если в списке стоят несколько запятых подряд, то это подразумевает наличие “нулей” между запятыми. Выводить списки со вставленными нулями и подсчитывать для каждой строки общее количество чисел и количество вставленных нулей.

Текст программы:

Содержимое файла ind.l:

%{

#include <stdlib.h>

#include "y.tab.h"

#define YYSTYPE int

extern YYSTYPE yylval; /\* value of numeric token \*/

%}

%%

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

[ \t]+ ;

,, {unput(',');unput('^');unput(',');}

.|\n return yytext[0];

%%

#ifndef yywrap

int yywrap () { return 1; }

#endif

Содержимое файла ind.y:

%{

int zeroes=0;

%}

%token NUM

%start \_\_list

%%

\_\_list: \_list { printf("\nNo. of items: %d\nNo. of added zerose: %d\n\n",$1,zeroes);zeroes=0; }

| \_\_list '\n' \_list { printf("\nNo. of items: %d\nNo. of added zerose: %d\n\n",$3,zeroes); zeroes=0; }

;

\_list: /\* empty \*/ { $$ = 0; /\* size is 0 \*/ }

| list /\* not empty, $$ == $1 by default \*/

;

list: NUM { $$ = 1;printf("%d ",$1);}

| '^' { $$ = 1;printf("%d ",0);zeroes++;}

| list ',' NUM { $$ = $1 + 1; printf("%d ",$3); } /\* size := size of sublist + 1 \*/

| list ',' '^' { $$ = $1 + 1; printf("%d ",0); zeroes++; }

;

%%

Тестовые данные:

1,2,,3

20,20, 10101010

1,,,2,3,,4

1,2,3,4,5,6,7,8,9,,,,,,0

Результат выполнения программы:

asdf@BigAwesomeTurtle:~/Documents/yacc/Microtask$ ./res.out < test.in

1 2 0 3

No. of items: 4

No. of added zerose: 1

20 20 10101010

No. of items: 3

No. of added zerose: 0

1 0 0 2 3 0 4

No. of items: 7

No. of added zerose: 3

1 2 3 4 5 6 7 8 9 0 0 0 0 0 0

No. of items: 15

No. of added zerose: 5

1. **Выводы**

В ходе выполнения лабораторной работы было произведено ознакомление с программой YACC, а также получены навыки написания синтаксических анализаторов. На приведенных примерах было изучено использование общей формы yacc-программы с использованием определений, правил и процедур пользователя.