1. Министерство образования и науки Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —
4. Институт компьютерных наук и технологий
5. **Кафедра «Информационная безопасность компьютерных систем»**

**ЛАБОРАТОРНАЯ РАБОТА № 1**

1. «**Построение грамматик YACC**»
2. по дисциплине «Формальные грамматики и теория компиляторов»
3. Выполнили
4. студенты гр. 33508/3 Князев П.В.
   * + 1. Палёный А.Э.

<*подпись*>

1. Преподаватель Семьянов П.В.
2. <*подпись*>

5. Санкт-Петербург
6. 2018
7. **Постановка задания**

С помощью YACC создать калькулятор полиномов. Затем на его основе создать транслятор языка полиномов. Язык полиномов должен поддерживать тип переменной «полином» и операции над полиномами. Более того, язык должен поддерживать операции присваивания и иметь оператор вывода полинома. Транслятор должен находить лексические, семантические и синтаксические ошибки в коде.

1. **Результаты работы**

В результате работы на языке программирования С++ была написана программа – интерпретатор языка полиномов. Она считывает входной файл с кодом и обрабатывает инструкции. При нахождении ошибки в коде программа останавливает свое выполнение и сообщает об ошибке пользователю.

* 1. **Возможности языка**

Язык поддерживает:

* Произвольный вид полинома: возможна запись полинома с использованием латинских букв нижнего и верхнего регистров (a + b + 1, 12acd, 2(xy), (x+y)(t+i) …)
* Вещественные числа: 12.05x + 3.01y
* Сложение и вычитание полиномов: a + b, 3b + 1, 45xyz - 12
* Умножение полиномов: 2x \* y, (x + y) \* (y + z),
* Возведение полинома в степень: p ^ n, где n – неотрицательное целое число
* Унарное отрицание: -p
* Вложенные выражения: p + ( p + ( … ))
* Присваивание: переменной можно присвоить значение, например, $Var = 2a
* Вывод полинома на консоль: $$out << 2x + 5
* Комментарии: линейные // и многострочные /\* \*/
  1. **Грамматика языка**

Грамматика языка показана на рис. 3. Этот рисунок представляет собой диаграмму состояний процесса обработки. Стрелками обозначены переходы в другое состояние при прямом ходе рекурсии, если это возможно. При обратном ходе рекурсии осуществляются все операции по вычислению выражения. Пример вычисления операций –x + 12 \* y и (х + у) \* (у + z) приведен на рис. 1, 2 в виде отладочных сообщений.

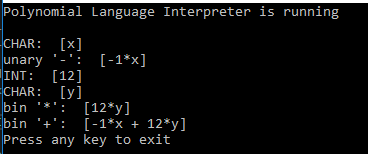
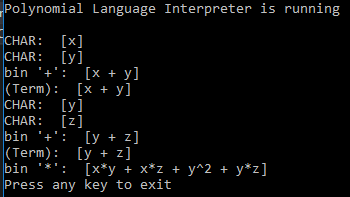
 

Рис. 1 «Обработка выражения –х + 12 \* у» Рис. 2 «Обработка выражения (х + у) \* (у + z)»

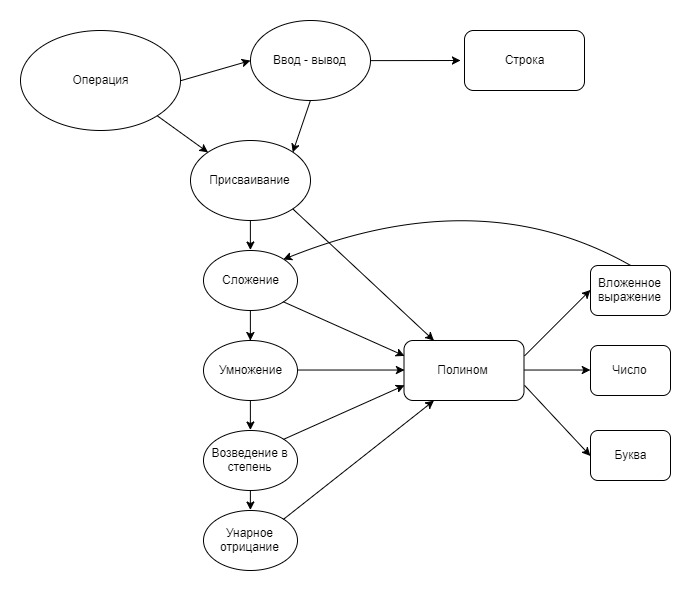
****

Рис. 3 «Диаграмма состояний процесса обработки»

* 1. **Синтаксис**

Для объявления переменной нужно написать строку, с символом «$» перед ней. То есть строка вида $ThisIsVar123 будет распознана как переменная. Переменная может состоять только из латинских букв и цифр.

Переменной можно присваивать различные значения и другие переменные, например,

$a = 1 + 3x; $b = a;

Для вывода переменной нужно использовать конструкцию

$$out << expr << $,

где expr – переменная или выражение, а n$ в конце – количество символов перехода на следующую строку (в данном случае 1). Эта конструкция позволяет выводить несколько значений. Например, инструкция $$out << 3 << 4 << 5; выведет на экран «345». Каждая инструкция в языке должна заканчиваться символом ;. Иначе будет ошибка синтаксиса.

* 1. **Хранение полинома в памяти**

Полином (Polynomial) представлен в виде списка:

[слагаемое 1] , [слагаемое 2] … [слагаемое N].

Слагаемое (PolyElem) представлено в виде структуры:

[double] , [{letter 1, power 1} … {letter M, power M}].

То есть слагаемое имеет коэффициент в начале и список из различных букв со степенями. Сначала был создан класс PolyElem. И в нем были перегружены операторы умножения и возведения в степень. После этого был создан класс Polynomial, представленный в виде списка PolyElem. Для этого класса были перегружены операторы +-\*^, реализующие непосредственно операции над полиномами. Такая структура обеспечивает неограниченное число слагаемых и букв со степенями в одном слагаемом (если хватит памяти и различных букв).

* 1. **Хранение переменных**

Для хранения переменных было создано хранилище переменных (VarStor). Оно представляет собой словарь map<string, Polynomial>. При нахождении сигнатуры $[a-z, 0-9] создается переменная и её имя заносится в хранилище. При этом переменная не содержит никаких данных, т.к. она еще не инициализирована. При инициализации ($a = 2) в хранилище переменных происходит поиск переменной с таким именем, а затем обновление её значения. При повторной инициализации старое значение переменной перезаписывается.

* 1. **Предварительная обработка и обнаружение ошибок**

После запуска программы, сначала происходит препроцессирование, т.е. удаляются пробелы и комментарии. (Рис. 4). Далее, происходит открытие обработанного файла и вызывается функция «yyparse». Данные действия происходят в блоках «try … catch()», что позволяет более точно отследить ошибки при выполнении программы. В блоке catch (std::exception e) будут обрабатываться синтаксические, семантические, лексические ошибки, а также ошибки, вызванные другими факторами.

Процесс обработки ошибок в программе, осуществляется в блоке «catch (std::exception e)», функции main. При этом в функции «yyparse», происходит лишь детектирование возможных ошибок. Когда произошла ошибка, выводится сообщение об исключении: «Exception occurred:». После этого функцией «showErrorLine», происходит вывод ошибочной инструкции в файле без комментариев. Затем с помощью функции showErrorPos выводится номер строчки с ошибкой в оригинальном файле (Рис. 5, 6).

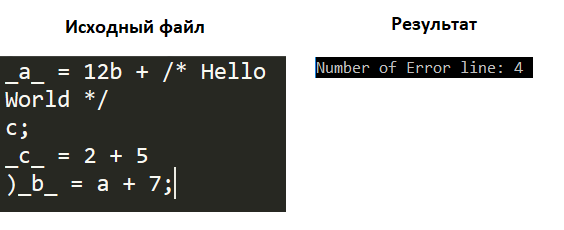


Рис. 5. Результат работы функции «showErrorLine»



Рис. 6. Пример работы функции «showErrorPos»

1. **Вывод**

В результате работы было изучено построение грамматик с помощью yacc. Этот программный продукт облегчает создание и анализ формальных языков со сложной синтаксической структурой.

*Приложение*

Lex.y

%token REAL INT ALPHACHAR VAR CSTR

%start operation

////////////////------------------- inline C code

%{

#include "../VarStor.h"

#define MAXSZ 255

// buf for input string

char gBuf[MAXSZ + 1] = { 0 };

int gIndex = 0;

// append one char

void AppendBuffer(int app);

// clear buf

void ClearBuffer();

// return buf data as string

const char\* GetBuffer();

%}

%union

{

Polynomial poly\_t;

int int\_t;

double real\_t;

}

%{

#define \_DEBUG\_OUT

// output macros

#ifdef \_DEBUG\_OUT

#define debug\_out\_p(val, s) \

printf("%s: %s [%s]\n", s, val##.getName().c\_str(), val##.out().c\_str())

#else

#define debug\_out\_p(val, s)

#endif

#define unused(x) x = 0

#include <malloc.h>

extern void yyerror(const char \*s);

extern int yylex();

%}

////////////////------------------- inline C code

//-- simple operands

%type <real\_t> REAL

%type <real\_t> INT

%type <int\_t> ALPHACHAR

%type <poly\_t> primary

//-- simple operands

//-- one operation between simple operands

%type <int\_t> operation

//-- possible operations between simple operands

%type <poly\_t> expr\_equal

%type <poly\_t> expr\_add

%type <poly\_t> expr\_mul

%type <poly\_t> expr\_pow

// io operator

%type <int\_t> operator

%type <int\_t> out\_operator

%%

/////////////------------ io operators

operator: out\_operator { unused($$); }

// out operator

out\_operator: '$'VAR { unused($$); }

// out expression

out\_operator: out\_operator '<''<' expr\_equal { unused($$); output($4); }

// out const string

out\_operator: out\_operator '<''<' CSTR { unused($$); output(GetBuffer()); }

// out \n

out\_operator: out\_operator '<''<' INT '$' { unused($$); output(int($4), '\n'); }

out\_operator: out\_operator '<''<' '$' { unused($$); output(int(1), '\n'); }

/////////////------------ io operators

///////////////------------///////////---- main block

// empty operator

operation: ';' { unused($$); }

// io operator

operation: operator ';' { unused($$); }

// expression

operation: expr\_equal ';' { unused($$); }

// combination

operation: operation operator ';' { unused($$); }

operation: operation expr\_equal ';' { unused($$); }

operation: operation ';' { unused($$); }

///////////////------------///////////---- main block

////////////////////////////------ expr

// binary and unary operations

//------- priority =

expr\_equal: expr\_add '=' expr\_equal { $$ = assignVar($1, $3); debug\_out\_p($$, "bin '='");}

expr\_equal: expr\_add;

//------- priority =

//------- priority + -

expr\_add: expr\_mul;

expr\_add: expr\_add '+' expr\_mul {$$ = calculate($1, $3, '+'); debug\_out\_p($$, "bin '+'");}

expr\_add: expr\_add '-' expr\_mul {$$ = calculate($1, $3, '-'); debug\_out\_p($$, "bin '-'");}

//------- priority + -

//------- priority \* /

// it's possible to multiply number on letter: 2a, ab

expr\_mul: expr\_mul ALPHACHAR {$$ = calculate($1, Polynomial(1, $2), '\*'); debug\_out\_p($$, "CHAR bin '\*'");}

expr\_mul: expr\_mul '(' expr\_add ')' {$$ = calculate($1, $3, '\*'); debug\_out\_p($$, "expr (term) '\*'");}

expr\_mul: expr\_pow;

expr\_mul: expr\_mul '\*' expr\_pow {$$ = calculate($1, $3, '\*'); debug\_out\_p($$, "bin '\*'");}

//------- priority \* /

//------- priority ^

expr\_pow: primary;

expr\_pow: primary '^' expr\_pow {$$ = calculate($1, $3, '^'); debug\_out\_p($$, "bin '^'");}

expr\_pow: '-' primary {$$ = calculate(Polynomial(0), $2, '-'); debug\_out\_p($$, "unary '-'"); }

expr\_pow: '-' primary '^' expr\_pow {$$ = calculate(Polynomial(0), calculate($2, $4, '^'), '-'); debug\_out\_p($$, "bin '^'; unary '-'"); }

//------- priority ^

//------- highest priority: number, letter

// it's a letter or real/int number

primary: ALPHACHAR { $$ = Polynomial(1, $1); debug\_out\_p($$, "CHAR"); }

primary: REAL { $$ = Polynomial($1); debug\_out\_p($$, "REAL"); }

primary: INT { $$ = Polynomial($1); debug\_out\_p($$, "INT"); }

// it's a variable

primary: VAR { $$ = createVariable(GetBuffer()); debug\_out\_p($$, "Variable"); }

// it's an expression in brackets

primary: '(' expr\_add ')' { $$ = $2; debug\_out\_p($$, "(Term)"); }

////////////////////////////------ expr

%%

void AppendBuffer(int app)

{

assert(gIndex < MAXSZ, "Input string is too long");

gBuf[gIndex++] = app;

}

void ClearBuffer()

{

gIndex = 0;

memset(gBuf, 0, MAXSZ);

}

const char\* GetBuffer()

{

return gBuf;

}

Poly.h

// Structure for polynomial: 1+2x-x^2+5x^7-45 abc^3, etc.

class Polynomial

{

// the list stores the summands 1 2x x^2 ...

std::list<PolyElem> listPoly;

// it's name

std::string name;

Polynomial(const PolyElem &x)

{

this->listPoly.push\_back(x);

}

public:

// ax^n ival - a, cval - x, ipow - n

Polynomial(const double& ival, const char& cval = 0, const double& ipow = 1);

bool None() { return listPoly.empty(); }

// empty list for non - initialized operands

Polynomial() = default;

// assign name to polynomial (need for variables)

void assignName(const std::string& x);

// remove thename of polynomial

void untieName() { this->name.clear(); }

// get name of polynomial (need for variables)

const std::string& getName();

// get the value of polynomial (need for variables)

std::list<PolyElem>& getValue();

Polynomial operator= (const Polynomial &e);

// The output of the polynomial to a string

std::string out(bool in\_column = false);

// unary minus: -(polynomial)

Polynomial operator- ();

// The addition of two polynomials

friend Polynomial operator+ (const Polynomial& leftPoly, const Polynomial& rightPoly);

friend Polynomial operator+= (Polynomial& leftPoly, const Polynomial& rightPoly);

// The substraction of two polynomials

friend Polynomial operator- (const Polynomial& leftPoly, const Polynomial& rightPoly);

friend Polynomial operator-= (Polynomial& leftPoly, const Polynomial& rightPoly);

// The multiplication of two polynomials

friend Polynomial operator\* (const Polynomial& leftPoly, const Polynomial& rightPoly);

//Polyomial Exponentiation: (x1 + ... + xm) ^ n

friend Polynomial operator^ (const Polynomial& leftPoly, const Polynomial& rightPoly);

};