**Отчёт по лабораторной работе №1. Вариант №1.**

**Задание:**

Обеспечить поддержку вещественных и комплексных чисел. Числа вводить в нижеследующем формате

*<вещ.число>::=<число>*

*<вещ.число>::=<число>.<число>*

*<комплксн.число>::=i*

*<комплксн.число>::=i+<вещ.число>*

*<комплксн.число>::=<вещ.число>i*

*<комплксн.число>::=<вещ.число>i+<вещ.число>*

На выходе комплексные числа представлять в следующем виде:

*complex(<комплексн.часть>,<реальн.часть>)*

где комплексн.часть и реальн.часть — вещественные числа. Вещественные числа выводить без изменений.

**Решение:**

1. Составим формальную грамматику программы для анализа комплексных чисел, применяемый стиль синтаксического анализа называется рекурсивным спуском:

**программа:**

**выражение**

**конец ввода**

**выражение:**

**выражение + терм**

**выражение – терм**

**терм**

**терм:**

**терм \* первичное выражение**

**первичное выражение**

**первичное выражение:**

**вещественная часть**

**мнимая часть**

**- первичное выражение**

1. Каждому порождающему правилу грамматики сопоставляется своя функция, вызывающая другие функции. Терминальные символы распознаются лексическим анализатором *get\_token():*

Token\_value get\_token()

{

char ch=0;

do{if(!input->get(ch))return curr\_tok=END;}

while(ch!='\n'&&isspace(ch));

switch(ch)

{

case 0 : return curr\_tok=END;

case '\*': case '+':

case '-':

return curr\_tok=Token\_value(ch);

case '0': case '1': case '2': case '3':

case '4': case '5': case '6': case '7':

case '8': case '9':

case '.' :

input->putback(ch);

\*input>>value;

input->get(ch);

if(ch=='i') return curr\_tok=IM;

input->putback(ch);

return curr\_tok=NUMBER;

case ';' :

case '\n' :

no\_of\_strings++;

return curr\_tok=PRINT;

default :

error("INCORECT WORD");

return curr\_tok=PRINT;

};

};

Самый последний возврат функции хранится в глобальной переменной curr\_tok, имеющей тип перечисления Token\_value:

enum Token\_value

{

NUMBER,END,IM,PLUS='+',MINUS='-',MUL='\*',PRINT=';'

};

1. Нетерминальные символы распознаются функциями синтаксического анализа expr(), term() и prim():

complex<double> expr(bool get)

{

complex<double> left=term(get);

for(;;)

{

switch(curr\_tok)

{

case PLUS : left+=term(true);break;

case MINUS : left-=term(true);break;

default : return left;

};

};

};

complex<double> term(bool get)

{

complex<double> left=prim(get);

for(;;)

{

switch(curr\_tok)

{

case MUL : left\*=prim(true);break;

default : return left;

};

};

};

complex<double> prim(bool get)

{

if(get)get\_token();

switch(curr\_tok)

{

case IM :

{

complex<double> v(0.0,value);

get\_token();

return v;

};

case NUMBER :

{

complex<double> v(value,0.0);

get\_token();

return v;

};

case MINUS : return -prim(true);

default : return error("PRIMARY EXPRESSION EXPECTED");

};

};

1. В главной функции производится анализ способа ввода данных (через аргументы командной строки или стандартный поток ввода std::cin), запускается алгоритм рекурсивного спуска, если ошибок в результате анализа не обнаружено, то принимаем строку:

int main(int argc, char\* argv[])

{

switch(argc)

{

case 1:

input=&cin;

break;

case 2:

input=new istringstream(argv[1]);

break;

default:

error("To much parameters!!!");

return 1;

};

complex<double> answer(0.0,0.0);

while(input)

{

get\_token();

if(curr\_tok==END)break;

if(curr\_tok==PRINT)continue;

answer = expr(false);

if(!no\_of\_errors)

if(answer.imag()) cout<<"OK complex(" << answer.imag() <<", " <<answer.real()<<")" << endl;

else cout<<"OK, "<< answer.real() << endl;

no\_of\_errors = 0;

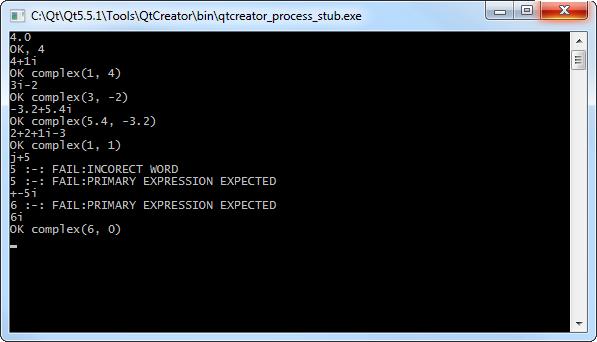
};

if(input!=&cin)delete input;

return no\_of\_errors;

};

Пример работы программы с вводом данных через std::cin:



Пример работы программы с входными параметрами – аргументами командной строки:

