|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | | МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ | | | | | |  | | | |
|  | | | |  | | | | | | | |
|  | Федеральное государственное автономное образовательное учреждение высшего образования  «Пермский государственный национальный исследовательский университет» | | | | | | | | | |  |
|  | | | | Кафедра математического обеспечения вычислительных систем | | | | | | | |
|  | **Отчет**  **По разработке компилятора языка Pascal** | | | | | | | |  | | |
|  | | |  | | | |  | | | | |
|  | |  | | |  | Работу выполнил студент группы ПМИ-3 4 курса механико-математического факультета  \_\_\_\_\_\_\_\_А.А. Солдатов | | | |  | |
|  | |  | | |  | Научный руководитель:  канд. физ.-мат. наук,  доцент каф. МОВС  \_\_\_\_\_\_\_\_Ф.А. Пономарев  “\_\_\_” \_\_\_\_\_\_\_\_\_\_2022 г. | | | |  | |
|  | | | Пермь 2022 | | | |  | | | | |

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#_Toc102006834)

[2 Структура компилятора 4](#_Toc102006835)

[3 Модуль ввода-вывода 5](#_Toc102006836)

[4 Лексический анализатор 6](#_Toc102006837)

[5 Синтаксический анализатор 7](#_Toc102006838)

[6 Семантический анализатор 8](#_Toc102006839)

# 1 Постановка задачи

Глобальное задание: написать компилятор для подмножества языка Паскаль. Задание разбивается на отдельные этапы:

1. Модуль ввода-вывода (8 баллов, оценивается совместно с лексическим анализатором).

2. Лексический анализатор (12 баллов, оценивается совместно с модулем ввода-вывода).

3. Синтаксический анализатор (12 баллов) с нейтрализацией синтаксических ошибок (8 баллов).

4. Семантический анализатор с нейтрализацией семантических ошибок (20 баллов).

5. Генерация кода (25 баллов).

Для получения минимального проходного балла необходимо реализовать указанные этапы для подмножества языка Паскаль, описанного далее в разделе «Общая минимальная часть».

Для получения 70% баллов за анализаторы и 100% баллов за генерацию кода необходимо дополнительно реализовать конструкции, описанные далее в разделе «Общая дополнительная часть».

Для получения 100% баллов за анализаторы необходимо дополнительно реализовать анализ конструкций, описанных далее в разделе «Индивидуальная часть», в соответствии с заданным вариантом.

В случае не сдачи студентом задания по генерации кода на последнем лабораторном занятии (или ранее), данное задание для данного студента заменяется теоретическим экзаменом, который оценивается исходя из максимума в 25 баллов.

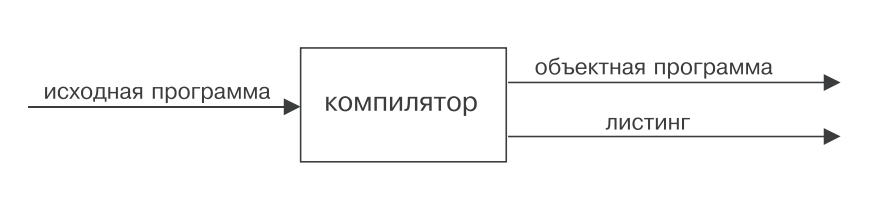
Замена задания по генерации на теоретический экзамен возможна также в случае, если студента не устраивают баллы, полученные за задание по генерации. В этом случае набранные баллы обнуляются, и студент сдает экзамен на тех же условиях, что и студенты, не сдававшие задание по генерации кода вообще.

Общая минимальная часть. Основные разделы программы: раздел описания переменных, раздел операторов. Переменные стандартных типов (Boolean, integer, real, char). Числовые константы. Арифметическое выражение (в выражении допустимы только константы, переменные, операции +, –, \*, / и скобки). Оператор присваивания и составной оператор.

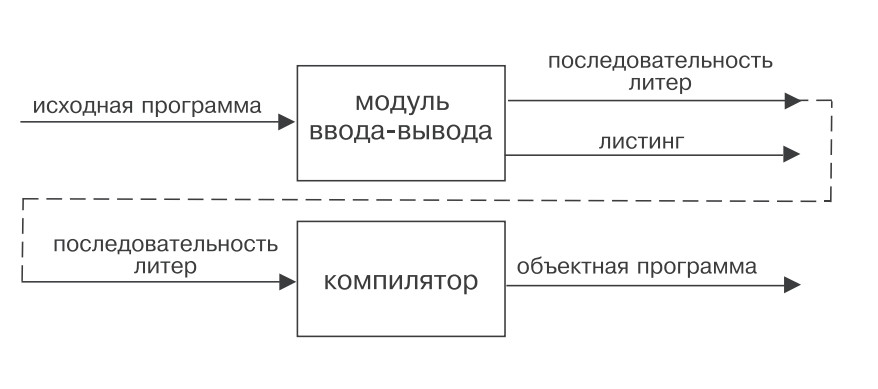
Общая дополнительная часть. Раздел описания типов. Выражение (полностью, включая арифметические, логические операции, сравнения и т.д., но только над константами и простыми переменными (не индексированные, не поля записи, не указатели)). Условный оператор (if). Оператор цикла с предусловием (while).

# 2 Структура компилятора

Компилятор – это программа, которая переводит программу на языке высокого уровня в эквивалентную программу на другом (объектном) языке. Обычно также выдает листинг, содержащий текст исходной программы и сообщения обо всех обнаруженных ошибках. Схема компилятора:



При вводе исходной программы и получении листинга мы имеем дело с конкретными устройствами ввода-вывода (клавиатура, дисплей, магнитные диски). Чтобы легко адаптировать компилятор к различным внешним устройствам конкретной машины, отделим все действия по вводу-выводу данных от собственно процесса компиляции:

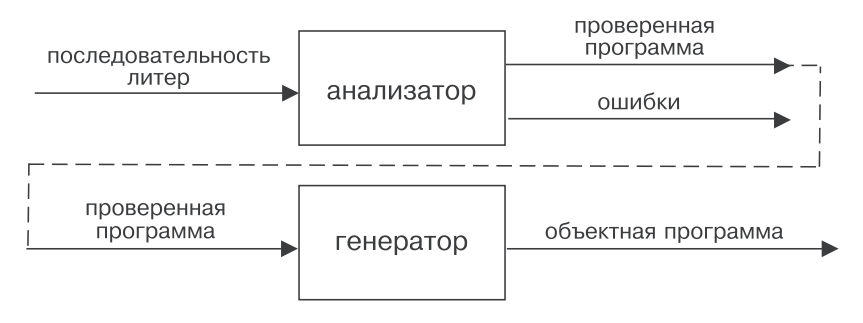


Работа компилятора включает в себя два основных этапа:

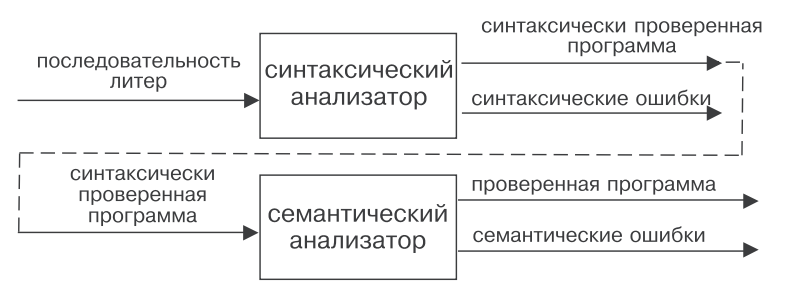
1) анализ — определение правильности исходной программы и формирование (в случае необходимости) сообщений об ошибках;

2) синтез — генерация объектной программы; этот этап выполняется для программ, не содержащих ошибок.

Таким образом, собственно компилятор разбивается на составляющие модули:



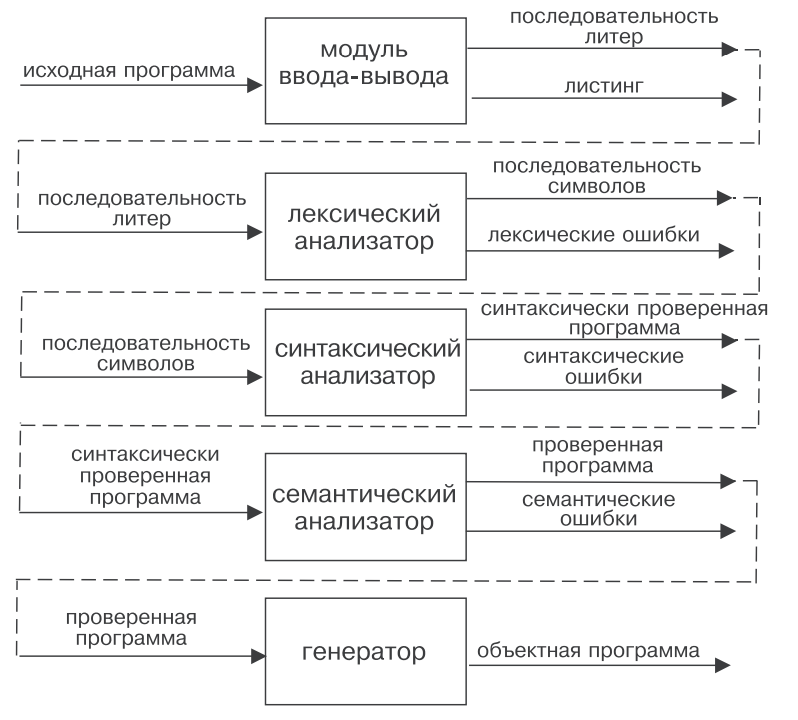
Учитывая особенности описания синтаксиса языков программирования, разделим анализатор на два модуля:



Синтаксический анализатор проверяет, удовлетворяет ли программа формальным правилам. Назначение же семантического анализатора состоит в том, чтобы выяснить, не нарушены ли неформальные правила описания языка.

Дальнейшее разбиение на модули обычно выполняется внутри синтаксического анализатора. Первый модуль – лексический анализатор, просматривает текст (последовательность литер) исходной программы и строит символы (лексемы) — идентификаторы, ключевые слова, разделители, числа. Второй модуль (синтаксический анализатор) выполняет синтаксический анализ последовательности символов. На этом этапе символы рассматриваются как неделимые, и их представление как последовательности литер несущественно.

Итак, в результате мы получили следующую структуру компилятора:



# 3 Модуль ввода-вывода

## 3.1 Ввод

Объявим класс для ввода данных:

class Reader {

public:

Reader(istream& stream);

~Reader();

pair<char, pair<int, int>> getCh();

private:

istream& stream;

int line, index;

};

В конструктор класса передается ссылка на поток ввода, чтобы можно было использовать различные источники ввода.

Также будем хранить номер текущей строки и номер текущего символа в строке в line и index соответственно.

Метод getCh возвращает пару из следующего символа и пары с номером текущей строки и номером текущего символа.

### Тестирование

Протестируем работу модуля на файле с программой на Паскале:

var

r1,r2: integer; // values on cubes

begin

r1 := Random(6)+1;

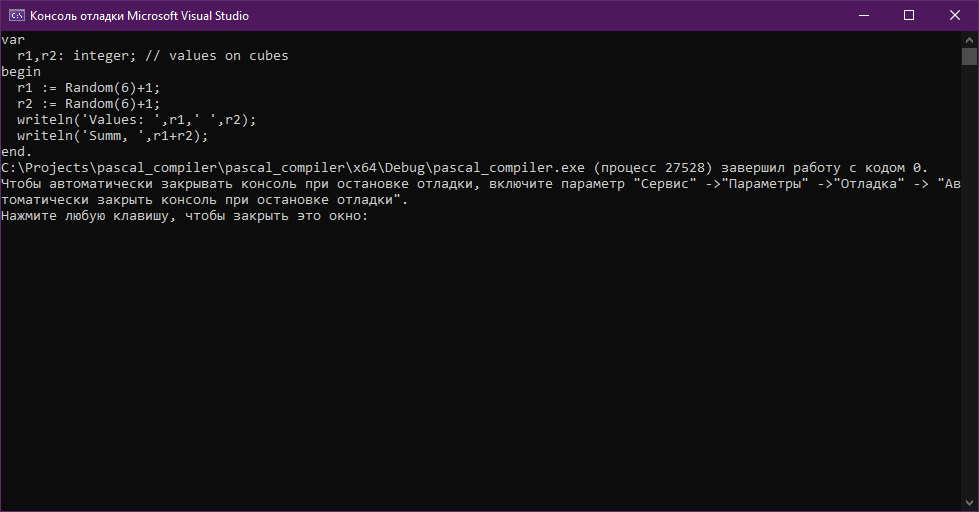
r2 := Random(6)+1;

writeln('Values: ',r1,' ',r2);

writeln('Summ, ',r1+r2);

end.

Вывод:



Теперь выведем символы вместе с их позициями:

v 0 0

a 0 1

r 0 2

0 3

0 4

1 0

1 1

r 1 2

1 1 3

, 1 4

r 1 5

2 1 6

: 1 7

1 8

i 1 9

n 1 10

t 1 11

e 1 12

g 1 13

e 1 14

r 1 15

; 1 16

1 17

/ 1 18

/ 1 19

1 20

v 1 21

a 1 22

l 1 23

u 1 24

e 1 25

s 1 26

1 27

o 1 28

n 1 29

1 30

c 1 31

u 1 32

b 1 33

e 1 34

s 1 35

1 36

b 2 0

e 2 1

g 2 2

i 2 3

n 2 4

2 5

3 0

3 1

r 3 2

1 3 3

3 4

: 3 5

= 3 6

3 7

R 3 8

a 3 9

n 3 10

d 3 11

o 3 12

m 3 13

( 3 14

6 3 15

) 3 16

+ 3 17

1 3 18

; 3 19

3 20

4 0

4 1

r 4 2

2 4 3

4 4

: 4 5

= 4 6

4 7

R 4 8

a 4 9

n 4 10

d 4 11

o 4 12

m 4 13

( 4 14

6 4 15

) 4 16

+ 4 17

1 4 18

; 4 19

4 20

5 0

5 1

w 5 2

r 5 3

i 5 4

t 5 5

e 5 6

l 5 7

n 5 8

( 5 9

' 5 10

V 5 11

a 5 12

l 5 13

u 5 14

e 5 15

s 5 16

: 5 17

5 18

' 5 19

, 5 20

r 5 21

1 5 22

, 5 23

' 5 24

5 25

' 5 26

, 5 27

r 5 28

2 5 29

) 5 30

; 5 31

5 32

6 0

6 1

w 6 2

r 6 3

i 6 4

t 6 5

e 6 6

l 6 7

n 6 8

( 6 9

' 6 10

S 6 11

u 6 12

m 6 13

m 6 14

, 6 15

6 16

' 6 17

, 6 18

r 6 19

1 6 20

+ 6 21

r 6 22

2 6 23

) 6 24

; 6 25

6 26

e 7 0

n 7 1

d 7 2

. 7 3

## 3.2 Вывод

Объявим класс для вывода данных:

class Writer {

public:

Writer(ostream& stream);

~Writer();

/\*template<class T> friend unique\_ptr<Writer>& operator << (const unique\_ptr<Writer>&,const T& t);\*/

ostream& stream;

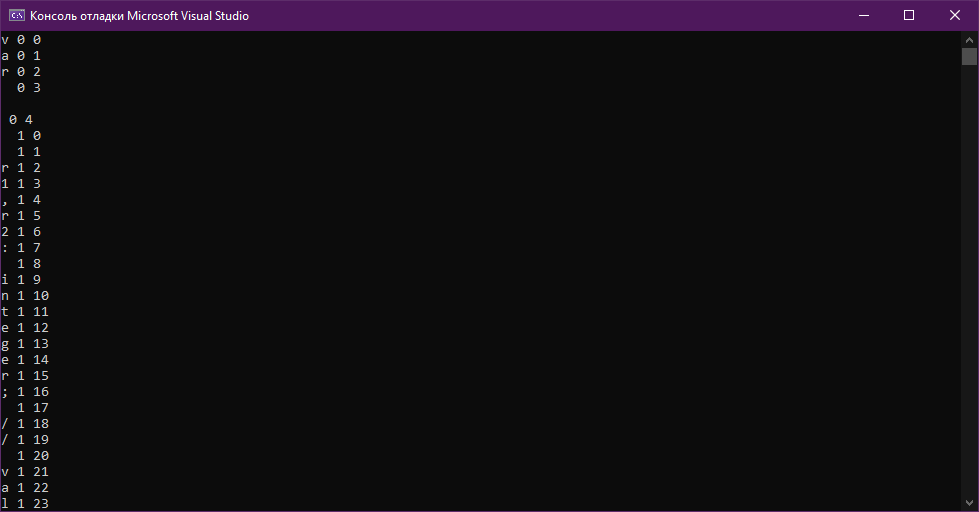
};

В конструктор передается поток вывода.

У меня не получилось адекватно перегрузить оператор «<<» для удобной работы с выводом, поэтому для вывода просто обращаемся к публичному полю stream.

### Тестирование

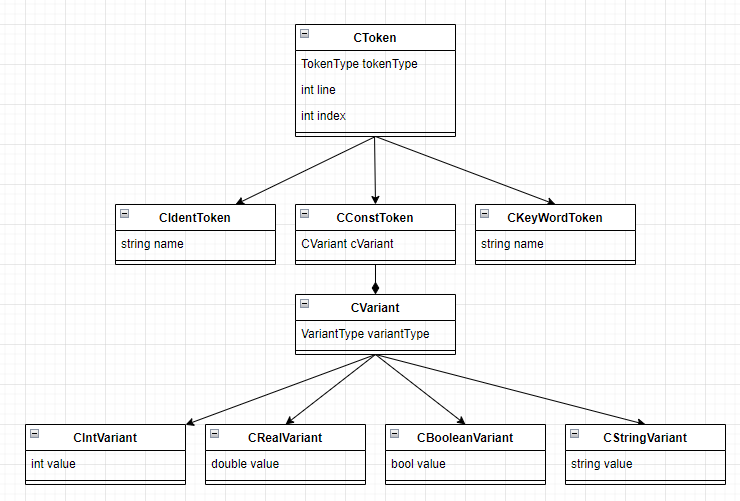
Протестируем модуль с той же программой, что в модуле ввода.



# 4 Лексический анализатор

Лексический анализатор используется для разбиения программы на лексемы/токены. В ходе работы он также проверяет программу на лексические ошибки. Комментарии игнорируются анализатором.

Токены могут быть 3 видов: идентификаторы, константы и ключевые слова языка. Константы может иметь один из 4х типов: integer, real, string или boolean (В нашей реализации компилятора будут поддерживаться только эти типы). Диаграмма классов для токенов:



TokenType и VariantType – перечислимые типы:

enum class VariantType {

vtInt,

vtReal,

vtString,

vtBoolean

};

enum class TokenType {

ttIdent,

ttKeyWord,

ttConst

};

Так же определим перечислимый тип для ключевых слов языка Pascal:

enum class CKeyWords {

programSy, // program

beginSy, // begin

endSy, // end

varSy, // var

plusSy, // +

minusSy, // -

multiplySy, // \*

divisionSy, // /

leftBracketSy, // (

rightBracketSy, // )

assignSy, // :=

dotSy, // .

commaSy, // ,

colonSy, // :

semicolonSy, // ;

eqSy, // =

ltSy, // <

gtSy, // >

leSy, // <=

geSy, // >=

neSy, // <>

ifSy, // if

thenSy, // then

elseSy, // else

whileSy, // while

doSy, // do

andSy, // and

orSy, // or

xorSy, // xor

notSy, // not

startCommentSy, // {

endCommentSy, // }

};

Для каждого перечислимого типа определим словарь, позволяющий по типу получить его строковое представление, а для ключевых слов определим и словарь для обратного преобразования из строки в тип.

Определим класс лексического анализатора:

class CLexer {

public:

CLexer(Reader\* reader);

unique\_ptr<CToken> getNextToken();

private:

unique\_ptr<Reader> reader;

char ch;

int line, index;

void skipComment();

pair<VariantType, string> getNumber();

string getFull();

string getString();

bool isLetter(char ch);

bool checkInt(string);

bool isValidNumber(string& number, int& separators);

void getNextChar();

};

В конструктор лексического анализатора передается модуль ввода. Лексический анализатор хранит текущий просматриваемый символ, текущий номер строки и номер символа. Основной метод getNextToken возвращает токен. Методы:

* skipComment() – пропускает комментарий в тексте программы
* getNumber() – возвращает тип числа и его строковую запись
* getFull() – возвращает имя идентификатора или ключевого слова
* getString() – возвращает содержимое строки
* isLetter() – проверяет, является ли символ буквой латинского алфавита или «\_»
* checkInt() – проверяет, поместится ли целое число в integer
* isValidNumber() – проверяет, является ли число целым или дробным
* getNextChar() – обновляет текущий символ, номер строки и символа, считывая их из модуля ввода

Будем обрабатывать следующие ошибки: неизвестный символ(символа нет в языке Pascal и он не является часть строки), непредвиденный символ(неверная последовательность символов), слишком большое число(для integer), неверная запись числа (в числе встретилось больше одной точки).

### Тестирование

Проверим вывод токенов на программе:

var

r1,r2: integer; { values on cubes }

begin

r1 := Random(6)+1;

r2 := Random(6)+1;

writeln('Values: ',r1,' ',r2);

writeln('Summ, ',r1+r2);

end.

Выведем токены с их типами:

0:0 var ttKeyWord

1:2 r1 ttIdent

1:4 , ttKeyWord

1:5 r2 ttIdent

1:7 : ttKeyWord

1:9 integer ttIdent

1:16 ; ttKeyWord

2:0 begin ttKeyWord

3:2 r1 ttIdent

3:5 := ttKeyWord

3:8 Random ttIdent

3:14 ( ttKeyWord

3:15 "6" integer ttConst

3:16 ) ttKeyWord

3:17 + ttKeyWord

3:18 "1" integer ttConst

3:19 ; ttKeyWord

4:2 r2 ttIdent

4:5 := ttKeyWord

4:8 Random ttIdent

4:14 ( ttKeyWord

4:15 "6" integer ttConst

4:16 ) ttKeyWord

4:17 + ttKeyWord

4:18 "1" integer ttConst

4:19 ; ttKeyWord

5:2 writeln ttIdent

5:9 ( ttKeyWord

5:10 "Values: " string ttConst

5:20 , ttKeyWord

5:21 r1 ttIdent

5:23 , ttKeyWord

5:24 " " string ttConst

5:27 , ttKeyWord

5:28 r2 ttIdent

5:30 ) ttKeyWord

5:31 ; ttKeyWord

6:2 writeln ttIdent

6:9 ( ttKeyWord

6:10 "Summ, " string ttConst

6:18 , ttKeyWord

6:19 r1 ttIdent

6:21 + ttKeyWord

6:22 r2 ttIdent

6:24 ) ttKeyWord

6:25 ; ttKeyWord

7:0 end ttKeyWord

7:3 . ttKeyWord

Проверим обработку ошибок на программе:

var

r1,r2: integer; { values on cubes }

r3 : real;

begin

r1 := 11111111111111111111111111111111111111;

r3 := 12341.12.12

r2 := %; {

end.

Вывод:

0:0 var ttKeyWord

1:2 r1 ttIdent

1:4 , ttKeyWord

1:5 r2 ttIdent

1:7 : ttKeyWord

1:9 integer ttIdent

1:16 ; ttKeyWord

2:2 r3 ttIdent

2:5 : ttKeyWord

2:7 real ttIdent

2:11 ; ttKeyWord

3:0 begin ttKeyWord

4:2 r1 ttIdent

4:5 := ttKeyWord

Error: 4:8 1003 IntegerLimit'11111111111111111111111111111111111111'

4:46 ; ttKeyWord

5:2 r3 ttIdent

5:5 := ttKeyWord

Error: 5:8 1002 InvalidNumber'12341.12.12'

6:2 r2 ttIdent

6:5 := ttKeyWord

Error: 6:8 1001 UnknownSymbol'%'

6:9 ; ttKeyWord

Error: 6:11 1000 UnexpectedSymbol'{'

# 5 Синтаксический анализатор

Все правила, описывающие синтаксис языка, имеют вид:

<S> ::= L

где S — обозначение синтаксической конструкции, a L может состоять из символов языка, синтаксических конструкций и метасимволов «|» и «{ }».

Основная идея написания синтаксического анализатора заключается в следующем. Для каждого правила необходимо описать функцию, тело которой является результатом некоторого преобразования правой части этого правила:

procs ( ) { Т( L ) }

При описании функции, соответствующей синтаксической конструкции S, мы будем придерживаться следующих соглашений:

а) перед обращением к функции переменная хранящая текущий символ содержит код первого символа, который должен быть проанализирован этой функцией;

б) функция проверяет, выводима ли входная последовательность символов из S, и в случае ошибки формирует соответствующее сообщение;

в) перед тем, как выйти из функции, в переменной запоминается код символа, который встретился сразу после анализа конструкции S.

## BNF

<program> ::= program <identifier>;<block>.|<block>.

<block> ::= <typePart> <varPart> <statementPart>

<varPart>::= <empty> | var <varDeclaration>; {<varDeclaration>;}

<varDeclaration> ::= <identifier> {,<identifier>} : <type>

<typePart> ::= <empty> | type <typeDeclaration>; {<typeDeclaration>;}

<typeDeclaration> ::= <identifier> = <type>

<type> ::= <identifier>

<empty> ::=

<statementPart> ::= <compoundStatement>

<compoundStatement> ::= begin <empty> | <statement>{;<statement>} end

<statement> ::= <simpleStatement>|<structuredStatement>

<simpleStatement> ::= <variable> := <expression>

<variable> ::= <identifier>

<expression> ::= <simpleExpression>|<simpleExpression> <relationalOperator> <simpleExpression>

<relationalOperator> ::= <|>|>=|<=|=|<>

<simpleExpression> ::= <term>{<addingOperator><term>}

<addingOperator> ::= + | - | or

<term> ::= <factor>{<multiplyingOperator><factor>}

<multiplyingOperator> ::= \* | / | and

<factor> ::= <variable> | <unsignedConst> | (<expression>) | <unaryOperator><factor>

<unaryOperator> ::= + | - | not

<unsignedConst> ::= <unsignedNumber> | <stringConst>

<unsignedNumber> ::= <unsignedInteger> | <unsignedReal>

<structuredStatement> ::= <compoundStatement>|<ifStatement>|<whileStatement>

<ifStatement> ::= if <expression> then <statement>| if<expression> then <statement> else <statement>

<whileStatement> ::= while <expression> do <statement>

В классе синтаксического анализатора придется реализовать методы для каждой(почти) конструкции.

## Проектирование

Класс синтаксического анализатора:

class CParser {

public:

CParser(CLexer\* lexer, Writer\* writer);

void parse();

private:

shared\_ptr<CToken> token;

unique\_ptr<CLexer> lexer;

unique\_ptr<Writer> writer;

void getNextToken();

void skipTo(bool, bool, vector<CKeyWords>);

bool fits(bool, bool, vector<CKeyWords>);

CKeyWords getTokenKeyWord();

VariantType getTokenVariantType();

bool isKeyWord();

bool isIdent();

bool isConst();

void passKeyword(CKeyWords);

void passConst(VariantType);

void passIdent();

void program();

void block();

void varPart();

void varDeclaration();

void type();

void typePart();

void typeDeclaration();

void statementPart();

void compoundStatement();

void statement();

void simpleStatement();

void expression();

void simpleExpression();

void term();

void factor();

void structuredStatement();

void ifStatement();

void whileStatement();

};

Основной метод parse() запускает проверку. Метод getNextToken() перезаписывает текущий токен, получая его из лексического анализатора. isKeyWord(), isIdent(), isConst() нужны для определения типа токена, passKeyword(CKeyWords), passConst(VariantType), void passIdent() проверяют токены соответствующего типа и вызывают getNextToken(). Методы skipTo и fits будут нужны для нейтрализации ошибок. Остальные методы реализуют БНФ.

## Нейтрализация ошибок

Без нейтрализации ошибок синтаксический анализатор будет работать до первой ошибки, однако на практике необходимо, чтобы анализ продолжался и после. Основная идея состоит в том, что после выявления ошибки, пропускается один или несколько токенов, чтобы найти токен с которого можно продолжить анализ. Реализуем метод skipTo, в нее передается нужно ли найти ближайший идентификатор(bool), нужно ли найти ближайшую константу(bool) и список ключевых слов. Пока не найдется что-либо из перечисленного, метод будет пропускать токены. Метод fits будет проверять, найден ли нужный токен. Конструкции в которых возможны ошибки обернем в try catch, если произошла ошибка, то выведем ее с помощью модуля вывода и вызовем метод skipTo, с параметрами, которые подбираются для каждой конкретной конструкции

## Тестирование

Проверим программу, написанную без ошибок:

var

r1,r2: integer; { values on cubes }

r1:real;

r12:int;

begin

r1 := 2+1;

r2 := 3+1;

r1:= r1+r2+r2+r1+1234;

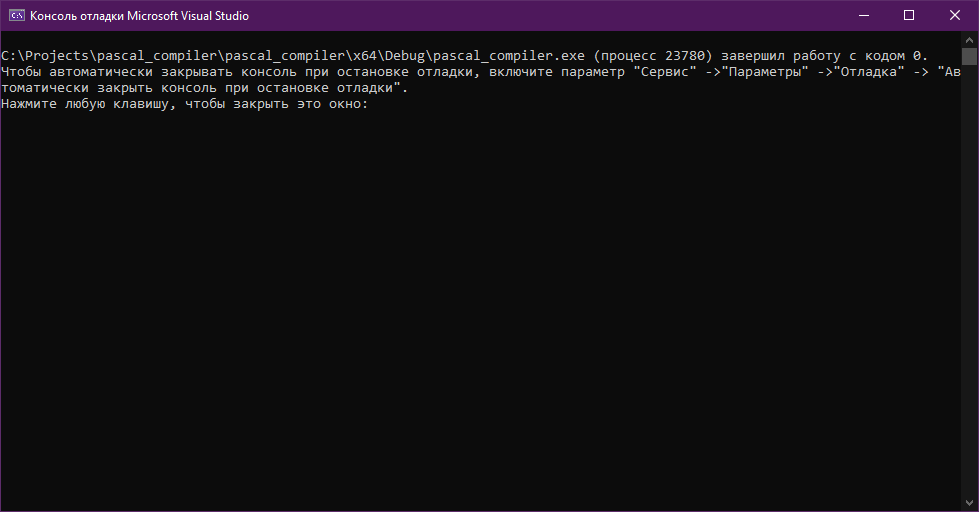
while r1<33 do

if r1<12 then r1:=r1+1 else r1:=r1+2;

r12:= 12+(r1\*3/(2+1))

end.

Результат:



Добавим ошибки в программу:

var

r1,r2,: integer; { values on cubes }

r1,:real;

r12.int;

begin

r1 := 2+1;

r2 := 3+1;

r1:= r1+r2+r2+r1+1234;

while r1<33 do

if r1<12 then r1:=r1+1 else r1:=r1+2;

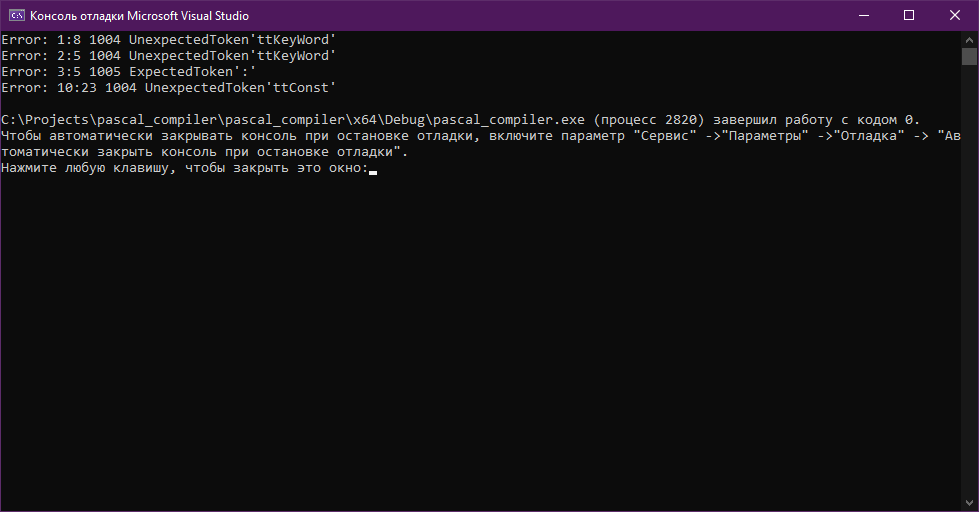
r12:= 12+(r1\*3/(2+1))1

r2:=0

r1:214

end.

Результат:



# 6 Семантический анализатор

Семантический анализатор должен проверять программу на соответствие неформальным правилам языка, т.к. в языке есть зависимость от контекста. Контекстные условия, которые нужно проверять при анализе программы:

1. В любой области действия без внутренних по отношению к ней областей действия никакой идентификатор не может быть описан более одного раза.
2. Каждому прикладному вхождению нестандартного идентификатора (стандартные идентификаторы — integer, boolean, real, char, true, false и др.) должно найтись соответствующее ему определяющее вхождение.
3. Контекстные условия предполагают также проверку соответствия типов величин, входящих в синтаксические конструкции программ: соответствия количества индексов у переменных с индексами и размерности соответствующих массивов и др.

Нам необходимо хранить область видимости, в которой будут записаны доступные типы и доступные идентификаторы (переменные с их типом).

Для хранения базовых типов реализуем перечислимый тип SemType, в нем также будет присутствовать тип, который обозначает ошибку, чтобы мы могли его передавать как результат выражений.

Реализуем класс CScope. В нем должны быть словари(map), хранящие пары (имя типа, SemType), (имя идентификатора(переменной), имя типа). То есть:

enum class SemType {

stInt,

stString,

stReal,

stBoolean,

stError

};

class CScope {

public:

CScope();

SemType getIdentType(string ident);

SemType getTypeforType(string ident);

bool identDefined(string ident);

bool typeDefined(string ident);

void addIdent(string ident, string identType);

void addType(string ident, SemType exprType);

private:

map<string, string>idents;

map<string, SemType> types;

};

В CScope так же будут методы позволяющие добавлять типы и идентификаторы, а также получать тип по идентификатору, определять существует ли тип или идентификатор.

Так как выполняемый вариант задания предполагает одну область видимости, у нас будет только один экземпляр CScope. В его конструкторе будем добавлять базовые типы языка Pascal и стандартные идентификаторы true и false:

CScope::CScope() {

addIdent("true", "boolean");

addIdent("false", "boolean");

addType("integer", SemType::stInt);

addType("string", SemType::stString);

addType("real", SemType::stReal);

addType("boolean", SemType::stBoolean);

}

В нашей реализации синтаксический анализ и семантический анализ будут проводиться одновременно. Для этого нужно изменить ранее реализованный класс CParser. В нем придется изменить методы, которые работают с идентификаторами и выражениями, так чтобы во время их выполнения можно было узнать тип текущего идентификатора или константы. В разделе описания типов мы проверяем, существует ли уже идентификатор создаваемого типа, существует ли базовый тип. В разделе описания переменных нам необходимо проверять существование типов, а также то, что идентификаторы переменных встречаются первый раз. Методы всех конструкций, которые связаны с выражениями возвращали свой тип. Если что-то пойдет не так внутри выражения, то вернется тип, обозначающий ошибку.

Были выделены следующие возможные ошибки:

* IdentAlreadyDefined – идентификатор уже был объявлен
* IdentNotDefined – обращение к необъявленному идентификатору
* TypeMismatch – несовпадение типов

## Тестирование

Проверим работу анализатора на программе с семантическими ошибками:

program dead;

type

yaUstal = integer;

var

r6:yaUstal;

r5:string;

r1,r2: integer; { values on cubes }

r1:real;

r12:int;

begin

r5:=r6;

r1 := 2+1;

r2 := 3+1;

r1:= r2+r1\*(r1+r2/(r1+r5));

r1:= r1+r2+r2+r1+1234;

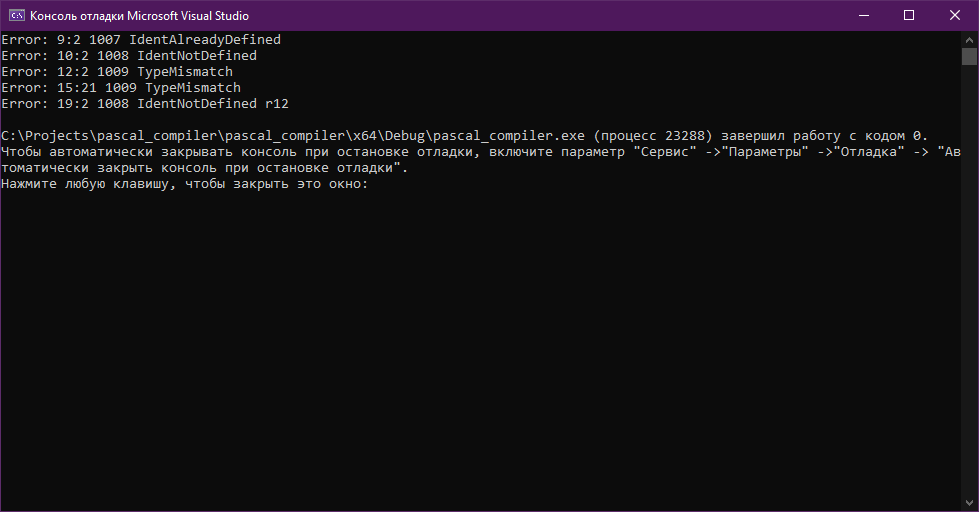
while r1<33 do

if r1<12 then r1:=r1+1 else r1:=r1+2;

r12:= 12+(r1\*3/(2+1))

end.

Результат:



Все ошибки найдены.