



ВЛАДИМИРСКИЙ  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
имени Александра Григорьевича и  
Николая Григорьевича Столетовых



Кафедра информатики и  
защиты информации  
(ИЗИ)

# РАЗРАБОТКА КОМПИЛЯТОРА ПОДМНОЖЕСТВА ПРОЦЕДУРНОГО ЯЗЫКА

**Руководитель:**

к.т.н. доцент кафедры ИЗИ Ю. М. Монахов

**Исполнитель:**

ст. гр. ИСБ-119 В. В. Окунев

г. Владимир 2021

# Аннотация

Разработка компилятора подмножества процедурного языка под LLVM состоит из следующих этапов:

- Реализация лексического анализатора
- Реализация синтаксического анализатора
- Реализация таблицы символов
- Реализация генератора промежуточного кода
- Реализация генератора объектного кода

Реквизиты к курсовой работе: [https://github.com/OBB-606/Compiler\\_final.git](https://github.com/OBB-606/Compiler_final.git)

Master ветка

# Основная часть: Лексический анализ

```
1 reserved = {  
    'if': 'IF',  
    'then': 'THEN',  
    'while': 'WHILE',  
    'begin': 'BEGIN',  
    'end': 'END',  
    'var': 'VAR',  
    'do': 'DO',  
    'continue': 'CONTINUE',  
    'break': 'BREAK',  
    'integer': 'INT',  
    'real': 'REAL',  
    'and': 'AND',  
    'or': 'OR',  
    'not': 'NOT',  
    'div': 'DIV',  
    'mod': 'MOD',  
    'print': 'PRINT',  
    'read': 'READ',  
    'string': 'STRI',  
    'program': 'PROGRAM',  
    'func': 'FUNC',  
    'proc': 'PROC',  
    'return': 'RETURN'
```

```
2 tokens = [  
    'ASSIGN', 'EQUAL',  
    'STRING', 'COLON', 'COMMA',  
    'OPEN_PAREN', 'CLOSE_PAREN', 'INT_DIGIT',  
    'PLUSMINUS', 'MULTIPLE', 'STR', 'SEMICOLON',  
    'ID', 'COMPARE', 'DOT', 'REAL_DIGIT', 'DIVIDE'  
    ] + list(reserved.values())  
t_DIVIDE = r'\/'  
t_DOT = r'\.'  
t_COMPARE = r'\>|<|=|<|>|<|<|>'  
t_EQUAL = r'\=='  
t_COLON = r'\:'  
t_ASSIGN = r'\='  
t_SEMICOLON = r';'  
t_COMMA = r','  
t_OPEN_PAREN = r'\('  
t_CLOSE_PAREN = r'\)'  
t_INT_DIGIT = r'\d+'  
t_PLUSMINUS = r'\+|\\-'  
t_MULTIPLE = r'\\*'  
t_REAL_DIGIT = r'\\d+\\.\\d+'
```

1 – правила зарезервированных слов

2 – регулярные выражения для  
определения мат. Операций,

чисел и т.п

```
program Hello;
var a,b,c : integer
begin
  a = 5;
  b = 10;
  c = a * b;
  print("a * b =");
  print(c)
end.
```

Исходный текст программы



ID	Token	Type token	Value token	Lin	Position
1	PROGRAM		program	1	8
2	ID		Hello	1	8
3	SEMICOLON		;	1	13
4	VAR		var	2	15
5	ID		a	2	19
6	COMMA		,	2	20
7	ID		b	2	21
8	COMMA		,	2	22
9	ID		c	2	23
10	COLON		:	2	25
11	INT		integer	2	27
12	BEGIN		begin	3	35
13	ID		a	4	45
14	ASSIGN		=	4	47
15	INT_DIGIT		5	4	49
16	SEMICOLON		;	4	50
17	ID		b	5	56
18	ASSIGN		=	5	58
19	INT_DIGIT		10	5	60
20	SEMICOLON		;	5	62
21	ID		c	6	68
22	ASSIGN		=	6	70
23	ID		a	6	72
24	MULTIPLE		*	6	74
25	ID		b	6	76
26	SEMICOLON		;	6	77
27	PRINT		print	7	83
28	OPEN_PAREN		(	7	88
29	STRING		"	7	89
30	STR		a * b =	7	90
31	STRING		"	7	97
32	CLOSE_PAREN		)	7	98
33	SEMICOLON		;	7	99
34	PRINT		print	8	105
35	OPEN_PAREN		(	8	110
36	ID		c	8	111
37	CLOSE_PAREN		)	8	112
38	END		end	9	114
39	DOT		.	9	117

Результат  
работы  
лексера

Токены хранятся в таблице,  
которая была создана с помощью  
библиотеки prettytable.

Столбцы этой таблицы:

Id токена, его тип, значение, строка  
и позиция в тексте программы.



1

```
Rule 0    S' -> program
Rule 1    program -> PROGRAM ID SEMICOLON declarations local_declarations body DOT
Rule 2    declarations -> <empty>
Rule 3    declarations -> declarations VAR identList COLON type
Rule 4    identList -> ID
Rule 5    identList -> identList COMMA ID
Rule 6    type -> INT
Rule 7    type -> REAL
Rule 8    type -> STRI
Rule 9    local_declarations -> <empty>
Rule 10   local_declarations -> local_declarations local_declaration SEMICOLON
Rule 11   local_declaration -> subHead declarations body
Rule 12   subHead -> FUNC ID args RETURN type SEMICOLON
Rule 13   subHead -> PROC ID args SEMICOLON
Rule 14   args -> <empty>
Rule 15   args -> OPEN_PAREN paramList CLOSE_PAREN
Rule 16   paramList -> identList COLON type
Rule 17   paramList -> paramList SEMICOLON identList COLON type
Rule 18   body -> BEGIN optionalStatements END
Rule 19   bodyWBC -> BEGIN optionalStatementsWBC END
Rule 20   optionalStatements -> <empty>
Rule 21   optionalStatements -> statementList
```

2

```
grammar kia;

program : PROGRAM ID SEMICOLON declarations subDeclarations comStatement DOT;

declarations : VAR identList COLON type (declarations)*;

identList : ID
          | identList COMMA ID;

type : INT
     | REAL;

subDeclarations : (subHead declarations comStatement)*;

subHead : FUNC ID args RETURN type SEMICOLON
        | PROC ID args SEMICOLON;

args : (OPEN_PAREN paramList CLOSE_PAREN)*;

paramList : identList COLON type
          | paramList SEMICOLON identList COLON type;

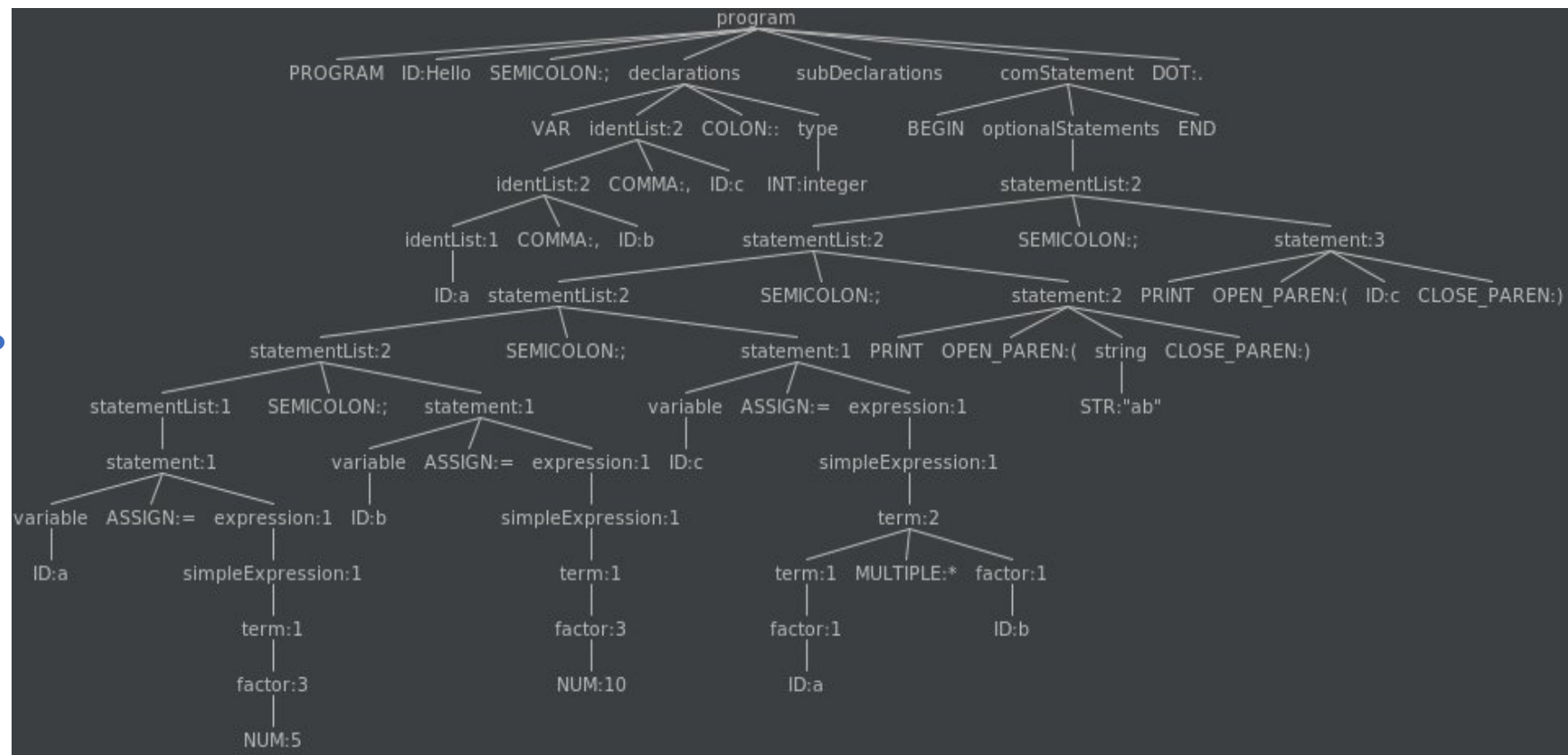
comStatement : BEGIN optionalStatements END SEMICOLON?;

comStatementWBC : BEGIN optionalStatementsWBC END;

optionalStatements : (statementList)*;
```

1, 2 – Часть грамматики языка kia

```
program Hello;
var a,b,c : integer
begin
    a = 5;
    b = 10;
    c = a * b;
    print("a * b =");
    print(c)
end.
```



# AST

```

class Node:
    def parts_str(self):
        st = []
        for part in self.parts:
            st.append(str(part))
        return "\n".join(st)

    def __repr__(self):
        return self.type + ":\n\t" + self.parts_str().replace("\n", "\n\t")

    def add_parts(self, parts):
        self.parts += parts
        return self

    def __init__(self, type, parts):
        self.type = type
        self.parts = parts

```



AST

```

Program:
  Var:
    ID:
      a
      b
      c
    Type:
      integer
  SubDeclare:

  Compound statement:
    Optional statements:
      Statement List:
        Assign:
          Variable:
            a
          Expression:
            5
        Assign:
          Variable:
            b
          Expression:
            10
        Assign:
          Variable:
            c
          Expression:
            *:
              a
              b
      print:
        Strings:
          a * b =
      print:
        c

```

Класс для  
генерации  
дерева в  
Parser.py

# Основная часть: Таблица символов

Таблица символов представляет собой словарь словарей, где первым ключом является `global` или название области действия (имя функции), что указывает на глобальность или локальность переменных.

```
-----SYMBOLS_TABLE-----  
global -- {'a': ['integer'], 'b': ['integer'], 'c': ['integer']}
```

Вторым ключом - имя переменной или функции, а значение — тип данных.

```
-----SYMBOLS_TABLE-----  
global -- {'a': ['integer'], 'b': ['integer'], 'c': ['integer'], 'h': ['real']}  
factorial -- {'factorial': ['integer'], 'a': ['integer'], 'result': ['integer'], 'c': ['integer'], 'd': ['integer'], 'e': ['integer']}
```



## Класс Block

```
class Block:
    def __init__(self):
        self.name = []
        self.instructions = []
        self.return_type = None
        self.params = []

    def append(self, instruction):
        self.instructions.append(instruction)

    def initname(self, name):
        self.name = name
```

Для генерации промежуточного кода был реализован алгоритм обхода дерева, который встречая определенные узлы, генерирует для них трехадресные инструкции.

Кусок исходного текста программы

```
program Factorials;  
var a,b,c : integer
```



Пример генерации промежуточного кода.

```
if part.type == 'Var':  
    if scope == 'global':  
        new = Block()  
        new.initname('__init')  
        new.return_type = 'void'  
        parentblock.append(new)  
    else:  
        new = Block()  
        new.initname(part.type)  
        parentblock.append(new)
```

```
for i in range(0, len(part.parts), 2):  
    for g in range(len(part.parts[i].parts)):  
        if scope != 'global':  
            tradr = 'alloc_' + typeconv[part.parts[i + 1].parts[0]]  
        else:  
            tradr = 'global_' + typeconv[part.parts[i + 1].parts[0]]  
        id = part.parts[i].parts[g]  
  
        new.append((tradr, id))  
        newtmpe = new_temp(typeconv[part.parts[i + 1].parts[0]])  
        new.append(('literal_' + typeconv[part.parts[i + 1].parts[0]],  
                    default[part.parts[i + 1].parts[0]], newtmpe))  
        new.append(('store_' + typeconv[part.parts[i + 1].parts[0]], newtmpe, id))
```



```
-----THREE-ADDRESS CODE-----  
__init | void []  
    ('global_int', 'a')  
    ('literal_int', 0, '__int_0')  
    ('store_int', '__int_0', 'a')  
    ('global_int', 'b')  
    ('literal_int', 0, '__int_1')  
    ('store_int', '__int_1', 'b')  
    ('global_int', 'c')  
    ('literal_int', 0, '__int_2')  
    ('store_int', '__int_2', 'c')
```

Для генерации объектного кода была использована библиотека `llvmlite`. Получая трехадресные инструкции из генератора трехадресного кода, транслятор обрабатывает их и создает файл с промежуточным кодом расширения `LL`.

```
( a <= b )
```



```
def emit_le_int(self, left, right, target):  
    self.temps[target] = self.builder.icmp_signed(  
        '<=', self.temps[left], self.temps[right], target)
```

```
%"__bool_1" = icmp sle i32 %"__int_22", %"__int_23"
```

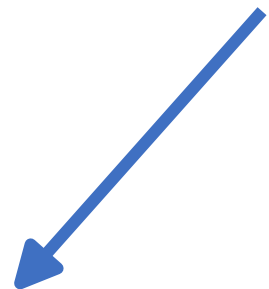


Еще один пример с вызовом функции factorial.

```
c = factorial(a);
```



```
def emit_call_func(self, funcname, *args):  
    target = args[-1]  
    func = self.globals[funcname]  
    argvals = [self.temps[name] for name in args[:-1]]  
    self.temps[target] = self.builder.call(func, argvals)
```



```
%.9" = call i32 @"factorial"(i32 %"__int_24")
```

# Пример работы : Hello

```
program Hello;  
var a,b,c : integer  
begin  
    a = 5;  
    b = 10;  
    c = a * b;  
    print("a * b =");  
    print(c)  
end.
```



```
-----LLVM CODE-----  
; ModuleID = "I want zachot"  
target triple = "x86_64-unknown-linux-gnu"  
target datalayout = ""  
  
declare i32 @"printf"(i8* %".1", ...)  
  
define void @"__init"()  
{  
entry:  
    store i32 0, i32* @"a"  
    store i32 0, i32* @"b"  
    store i32 0, i32* @"c"  
    br label %"exit"  
exit:  
    ret void  
}  
  
@"a" = global i32 0  
@"b" = global i32 0  
@"c" = global i32 0
```

```
define void @"main"()  
{  
entry:  
    store i32 5, i32* @"a"  
    store i32 10, i32* @"b"  
    %"__int_5" = load i32, i32* @"a"  
    %"__int_6" = load i32, i32* @"b"  
    %"__int_7" = mul i32 %"__int_5", %"__int_6"  
    store i32 %"__int_7", i32* @"c"  
    %"__str_0" = alloca [8 x i8]  
    store [8 x i8] c"a * b =\00", [8 x i8]* %"__str_0"  
    %".6" = bitcast [5 x i8]* @"__str_0" to i8*  
    %".7" = call i32 (i8*, ...) @"printf"(i8* %".6", [8 x i8]* %"__str_0")  
    %"__int_8" = load i32, i32* @"c"  
    %".8" = bitcast [5 x i8]* @"__int_8" to i8*  
    %".9" = call i32 (i8*, ...) @"printf"(i8* %".8", i32 %"__int_8")  
    br label %"exit"  
exit:  
    ret void  
}  
  
@"__str_0" = internal constant [5 x i8] c"%s \0a\00"  
@"__int_8" = internal constant [5 x i8] c"%i \0a\00"
```

```
-----RESULTS-----  
a * b =  
50  
running time: 0.013228416442871094 sec
```



Результат



# Пример работы с учетом требований к языку: factorials.kia

```
{I was made for lovin' you, baby !!!!!}
```

Комментарий

```
program Factorials;
```

```
var a,b,c : integer
```

Типы int и real

```
var h : real
```

```
func factorial (a: integer) return integer;
```

```
var result,c,d,e : integer
```

```
begin
```

```
    result = 1;
```

Оператор присваивания

```
    e = 1;
```

```
    while ( e <= a ) do
```

```
    begin
```

```
        result = result * e;
```

Арифметика

```
        e = e + 1
```

```
    end;
```

```
    factorial = result
```

```
end;
```

```
begin
```

```
    print("factorials");
```

Вывод строки

```
    b = 10;
```

```
    a = 1;
```

```
    while ( a <= b) do
```

Цикл while

```
    begin
```

```
        c = factorial(a);
```

Поддержка вызова функций

```
        if (c > 6) then
```

Условный оператор

```
        begin
```

```
            print("Rock!")
```

```
        end;
```

```
        print(c);
```

Вывод переменной

```
        a = a+1
```

```
    end
```

```
end.
```

```
-----RESULTS-----  
factorials  
1  
2  
6  
Rock!  
24  
Rock!  
120  
Rock!  
720  
Rock!  
5040  
Rock!  
40320  
Rock!  
362880  
Rock!  
3628800  
running time: 0.02473139762878418 sec
```

## Пример работы : дерево разбора factorials.kia

