

**Министерство науки и высшего образования Российской Федерации  
ФГБОУ ВО «Владимирский государственный университет  
имени Александра Григорьевича и Николая Григорьевича Столетовых»  
(ВлГУ)**

**Кафедра «Информатики и защиты информации»**

*Курсовая работа на тему:*

# **Разработка компилятора подмножества процедурного языка в ассемблер**

*Специальность: 10.05.04 – Информационно-аналитические системы  
безопасности*



***КАТКОВ Антон Николаевич,  
ст. гр. ИСБ-118***

# Технические требования

Разработка производилась в соответствии со следующими требованиями:

- Требования к входному языку:
  - Должны присутствовать операторные скобки;
  - Должна игнорироваться индентация программы;
  - Должны поддерживаться комментарии любой длины;
  - Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.
- Требования к операторам:
  - Оператор присваивания;
  - Арифметика (`/`, `+`, `*`, `-`, `>`, `<`, `=`);
  - Логические операторы (И, ИЛИ, НЕ);
  - Условный оператор (ЕСЛИ);
  - Оператор цикла (`while`, `break`, `continue`);
  - Базовый вывод (строковой литерал, переменная);
  - Типы (целочисленный, вещественный).

- Компилятор – это программа, которая переводит текст, написанный на языке программирования, в набор машинных кодов.
- Компилятор реализован на языке Python под язык Pascal.
- Использованные библиотеки PLY(Python Lex-Yacc) и LLVMlite.



# Лексический анализатор

```
LexToken(ID, 'f', 20, 263)
LexToken(ASSIGN_OPERATOR, ':=', 20, 265)
LexToken(ID, 'g', 20, 268)
LexToken(ARITHMETIC_OPERATOR1, '-', 20, 270)
LexToken(ID, 'd', 20, 272)
LexToken(ARITHMETIC_OPERATOR2, '/', 20, 274)
LexToken(ID, 'e', 20, 276)
LexToken(colonss, ';', 20, 277)
LexToken(ID, 'sqrt', 21, 279)
LexToken(ASSIGN_OPERATOR, ':=', 21, 284)
LexToken(ID, 'f', 21, 287)
LexToken(colonss, ';', 21, 288)
LexToken(END, 'end', 22, 290)
LexToken(colonss, ';', 22, 293)
LexToken(BEGIN, 'begin', 23, 295)
LexToken(ID, 'h', 24, 301)
LexToken(ASSIGN_OPERATOR, ':=', 24, 303)
LexToken(ID, 'sqrt', 24, 306)
LexToken(op_bracket, '(', 24, 310)
```

- Лексический анализатор преобразует входной поток символов в поток токенов. Реализован посредством библиотеки PLY (Python Lex-Yacc).

# Парсер

- На вход парсеру подаётся набор токенов из лексического анализатора. Результатом является дерево разбора грамматики. Парсер реализован посредством библиотеки PLY (Python Lex-Yacc).

```
Var:
  declarations:
    ID:
      a
      b
    Type:
      integer
    ID:
      c
    Type:
      real
  subprogram_declarations:
    None
  compound_statement:
    statement_list:
      ASSIGN_OPERATOR:
        ID:
          a
        elem:
          element:
            10
      ASSIGN_OPERATOR:
        ID:
          b
        elem:
          element:
            4
      ASSIGN_OPERATOR:
        ID:
          c
        expression:
          elem:
            element:
              a
            /
            elem:
              element:
                b
      WRITE:
        elem:
          element:
            c
```

```
var
  a, b: integer;
  c: real;
begin
  a:=10;
  b:=4;
  c:=a/b;
  write(c);
end.
```

# Таблица символов

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

procedure pr(x: real);
var
  y: integer;
begin
  write(x);
end;

begin
  a:=10;
  b:=4;
  c:=a/b;
  write(c);
end.
```

- Для генерации таблицы символов производится обход дерева по тем ветвям где может быть объявление переменных и параметров. Для хранения таблицы используется словарь.

```
C:\Users\myra0\Desktop\PROGA>python -m Table test.txt
global {'a': 'integer', 'b': 'integer', 'c': 'real'}
```

```
pr {'x': 'real', 'y': 'integer'}
```

```
C:\Users\myra0\Desktop\PROGA>
```

# Генератор промежуточного кода

```
('func', '__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_float', 'c')
('literal_float', 0.0, '__float_0')
('store_float', '__float_0', 'c')
('return_void',)

('func', 'main', 'void')

('literal_int', 10, '__int_2')
('store_int', '__int_2', 'a')
('literal_int', 4, '__int_3')
('store_int', '__int_3', 'b')
('load_int', 'a', '__int_4')
('load_int', 'b', '__int_5')
('int_to_float', '__int_4', '__float_1')
('int_to_float', '__int_5', '__float_2')
('div_float', '__float_1', '__float_2', '__float_3')
('store_float', '__float_3', 'c')
('load_float', 'c', '__float_4')
('print_float', '__float_4')
('return_void',)
```

- Для генерации промежуточного кода производится обход дерева, в результате которого создаются соответствующие команды. Каждая инструкция хранится в кортеже, которые в свою очередь хранятся в массивах. Массивы поделены на блоки.

```
var
  a, b: integer;
  c: real;
begin
  a:=10;
  b:=4;
  c:=a/b;
  write(c);
end.
```



# Генератор объектного кода

```
; ModuleID = "module"
target triple = "x86_64-pc-windows-msvc"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define void @"__init"()
{
entry:
    store i32 0, i32* @"a"
    store i32 0, i32* @"b"
    store double 0x0, double* @"c"
    br label %"exit"
exit:
    ret void
}

@"a" = global i32 0
@"b" = global i32 0
@"c" = global double 0x0
define void @"main"()
{
entry:
    store i32 10, i32* @"a"
    store i32 4, i32* @"b"
    %"__int_4" = load i32, i32* @"a"
    %"__int_5" = load i32, i32* @"b"
    %"__float_1" = sitofp i32 %"__int_4" to double
    %"__float_2" = sitofp i32 %"__int_5" to double
    %"__float_3" = fdiv double %"__float_1", %"__float_2"
    store double %"__float_3", double* @"c"
    %"__float_4" = load double, double* @"c"
    %".5" = bitcast [5 x i8]* @"__fstr_0" to i8*
    %".6" = call i32 (i8*, ...) @"printf"(i8* %".5", double %"__float_4")
    br label %"exit"
exit:
    ret void
}

@"__fstr_0" = internal constant [5 x i8] c"%f \0a\00"
```

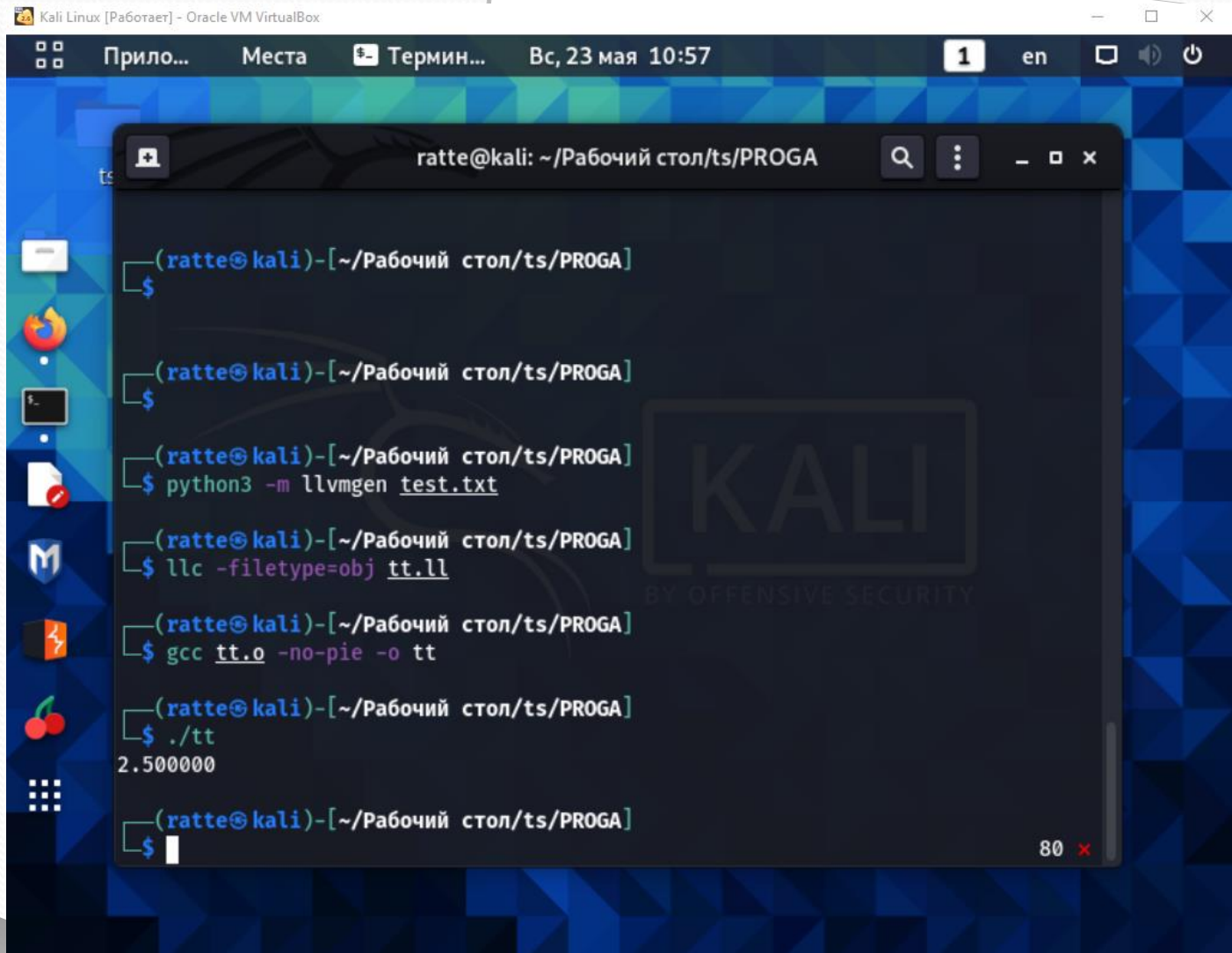
- Для реализации этой стадии была использована библиотека `llvmlite`. Получая инструкции из генератора промежуточного кода, генератор объектного кода обрабатывает их и создает файл `tt.ll`. Далее можно запустить файл из командной строки `линукс`, либо используя `Comppiler.py`, который реализует оптимизацию, трансляцию кода в целевую машину и его выполнение

```
C:\Users\myra0\Desktop\PROGA>python -m Compiller test.txt
2.500000
```

```
C:\Users\myra0\Desktop\PROGA>
```



# Запуск из линукса



The screenshot shows a Kali Linux desktop environment with a terminal window open. The terminal window title is "ratte@kali: ~/Рабочий стол/ts/PROGA". The terminal output shows the following commands and results:

```
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$ python3 -m llvngen test.txt
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$ llc -filetype=obj tt.ll
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$ gcc tt.o -no-pie -o tt
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$ ./tt
2.500000
(ratte@kali)-[~/Рабочий стол/ts/PROGA]
$
```

The terminal window has a search icon, a menu icon, and window control buttons. The desktop background is a blue geometric pattern. The taskbar on the left shows icons for a file manager, Firefox, a terminal, a document, a mail client, a code editor, and a game. The top of the desktop shows a panel with icons for applications, locations, and a terminal, along with the date and time "Вс, 23 мая 10:57".

# Пример 1

```
var u:integer;
function power(t, k: integer): integer;
var
    res:integer;
begin
    res := 1;
    while (k > 0) do
        begin
            if (k mod 2 = 1) then
                begin
                    res := res * t;
                end;
            t := t * t;
            k := k div 2;
        end;
    power := res;
end;
begin
    u:=power(5,3);
    write('result');
    write(u);
end.
```

Программа на Pascal

```
; ModuleID = "module"
target triple = "x86_64-pc-windows-msvc"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define void @"__init"()
{
entry:
    store i32 0, i32* @"u"
    br label %"exit"
exit:
    ret void
}

@"u" = global i32 0
define i32 @"power"(i32 %".1", i32 %".2")
{
entry:
    %"return" = alloca i32
    %"t" = alloca i32
    store i32 %".1", i32* %"t"
    %"k" = alloca i32
    store i32 %".2", i32* %"k"
    %"power" = alloca i32
    store i32 0, i32* %"power"
    %"res" = alloca i32
    store i32 0, i32* %"res"
    store i32 1, i32* %"res"
    br label %"while_block"
exit:
    %".22" = load i32, i32* %"return"
    ret i32 %".22"
```

Сгенерированный код 1 часть

```

while_block:
    %"__int_17" = load i32, i32* %"k"
    %"__bool_1" = icmp sgt i32 %"__int_17", 0
    br i1 %"__bool_1", label %"true_while_block", label %"end_while_block"
true_while_block:
    br label %"if_block"
end_while_block:
    %"__int_19" = load i32, i32* %"res"
    store i32 %"__int_19", i32* %"power"
    %"__int_20" = load i32, i32* %"power"
    store i32 %"__int_20", i32* %"return"
    br label %"exit"
if_block:
    %"__int_7" = load i32, i32* %"k"
    %"__int_9" = srem i32 %"__int_7", 2
    %"__bool_0" = icmp eq i32 %"__int_9", 1
    br i1 %"__bool_0", label %"true_if_block", label %"false_if_block"
true_if_block:
    %"__int_4" = load i32, i32* %"res"
    %"__int_5" = load i32, i32* %"t"
    %"__int_6" = mul i32 %"__int_4", %"__int_5"
    store i32 %"__int_6", i32* %"res"
    br label %"end_if_block"
false_if_block:
    br label %"end_if_block"
end_if_block:
    %"__int_11" = load i32, i32* %"t"
    %"__int_12" = load i32, i32* %"t"
    %"__int_13" = mul i32 %"__int_11", %"__int_12"
    store i32 %"__int_13", i32* %"t"
    %"__int_14" = load i32, i32* %"k"
    %"__int_16" = sdiv i32 %"__int_14", 2
    store i32 %"__int_16", i32* %"k"
    br label %"while_block"
}

```

Сгенерированный код 2 часть



```

define void @"main"()
{
entry:
    %".2" = call i32 @"power"(i32 5, i32 3)
    store i32 %".2", i32* @u"
    %"__str_0" = alloca [9 x i8]
    store [9 x i8] c"'result'\00", [9 x i8]* %"__str_0"
    %".5" = bitcast [5 x i8]* @"__fstr_0" to i8*
    %".6" = call i32 (i8*, ...) @"printf"(i8* %".5", [9 x i8]* %"__str_0")
    %"__int_24" = load i32, i32* @u"
    %".7" = bitcast [5 x i8]* @"__fstr_1" to i8*
    %".8" = call i32 (i8*, ...) @"printf"(i8* %".7", i32 %"__int_24")
    br label %"exit"

exit:
    ret void
}

@"__fstr_0" = internal constant [5 x i8] c"%s \0a\00"
@"__fstr_1" = internal constant [5 x i8] c"%i \0a\00"

```

Результат работы

Сгенерированный код 3 часть

```

C:\Users\myra0\Desktop\PROGA>python -m Compiler pow.txt
'result'
125

C:\Users\myra0\Desktop\PROGA>

```

# Пример 2

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

begin
  a:=10;
  b:=3;
  while (b<a) do
  begin
    b:=b+1;
    if (b=5) then
    begin
      write('continue');
      continue;
    end;
    write(b);
    if (b=6) then
    begin
      write('break');
      break;
    end;
  end;
end.
```

Программа на Pascal

```
; ModuleID = "module"
target triple = "x86_64-pc-windows-msvc"
target datalayout = ""

declare i32 @"printf"(i8* %".1", ...)

define void @"__init"()
{
entry:
  store i32 0, i32* @"a"
  store i32 0, i32* @"b"
  store double 0x0, double* @"c"
  br label %"exit"
exit:
  ret void
}

@"a" = global i32 0
@"b" = global i32 0
@"c" = global double 0x0
define void @"main"()
{
entry:
  store i32 10, i32* @"a"
  store i32 3, i32* @"b"
  br label %"while_block"
while_block:
  %"__int_12" = load i32, i32* @"b"
  %"__int_13" = load i32, i32* @"a"
  %"__bool_2" = icmp slt i32 %"__int_12", %"__int_13"
  br i1 %"__bool_2", label %"true_while_block", label %"end_while_block"
true_while_block:
  %"__int_4" = load i32, i32* @"b"
  %"__int_6" = add i32 %"__int_4", 1
  store i32 %"__int_6", i32* @"b"
  br label %"if_block"
end_while_block:
  br label %"exit"
```

Сгенерированный код 1 часть

```

if_block:
    %"__int_7" = load i32, i32* @"b"
    %"__bool_0" = icmp eq i32 %"__int_7", 5
    br i1 %"__bool_0", label %"true_if_block", label %"false_if_block"
true_if_block:
    %"__str_0" = alloca [11 x i8]
    store [11 x i8] c"'continue'\00", [11 x i8]* %"__str_0"
    %".10" = bitcast [5 x i8]* @"__fstr_0" to i8*
    %".11" = call i32 (i8*, ...) @"printf"(i8* %".10", [11 x i8]* %"__str_0")
    br label %"while_block"
false_if_block:
    br label %"end_if_block"
end_if_block:
    %"__int_9" = load i32, i32* @"b"
    %".14" = bitcast [5 x i8]* @"__fstr_1" to i8*
    %".15" = call i32 (i8*, ...) @"printf"(i8* %".14", i32 %"__int_9")
    br label %"if_block.1"
if_block.1:
    %"__int_10" = load i32, i32* @"b"
    %"__bool_1" = icmp eq i32 %"__int_10", 6
    br i1 %"__bool_1", label %"true_if_block.1", label %"false_if_block.1"
true_if_block.1:
    %"__str_1" = alloca [8 x i8]
    store [8 x i8] c"'break'\00", [8 x i8]* %"__str_1"
    %".19" = bitcast [5 x i8]* @"__fstr_2" to i8*
    %".20" = call i32 (i8*, ...) @"printf"(i8* %".19", [8 x i8]* %"__str_1")
    br label %"end_while_block"
false_if_block.1:
    br label %"end_if_block.1"
end_if_block.1:
    br label %"while_block"
}

@"__fstr_0" = internal constant [5 x i8] c"%s \0a\00"
@"__fstr_1" = internal constant [5 x i8] c"%i \0a\00"
@"__fstr_2" = internal constant [5 x i8] c"%s \0a\00"

```

## Сгенерированный код 2 часть

## Результат работы

```

C:\Users\myra0\Desktop\PROGA>python -m Compiler test.txt
4
'continue'
6
'break'

C:\Users\myra0\Desktop\PROGA>

```