Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Владимирский государственный университет
имени Александра Григорьевича и Николая Григорьевича Столетовых»
(ВлГУ)

РАЗРАБОТКА КОМПИЛЯТОРА ПОДМНОЖЕСТВА ПРОЦЕДУРНОГО ЯЗЫКА

Пояснительная записка
На 19 листах

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

Исполнитель _____ студент гр. ИСБ-119 Д. А. Журавлев

Владимир 2022

АННОТАЦИЯ

В данном документе приведено описание компилятора подмножества процедурно-ориентированного языка KIA. Компилятор реализован на языке Python с использованием библиотек ply и llvmlite. Основная функция компилятора – проверка принадлежности исходной цепочки входному языку и генерация выходной цепочки символов виде llvm-кода.

Разработка компилятора подмножества процедурного языка состоит из следующих этапов:

- Реализация лексического анализатора
- Реализация синтаксического анализатора
- Реализация таблицы символов
- Реализация семантический анализатора
- Реализация генератора промежуточного кода
- Трансляция в целевой код
- Оптимизация

Реквизиты к курсовой работе:

Оглавление

АННОТАЦИЯ.....2
1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА.....4
1.1 Основные требования.....4
1.2 Лексический анализатор.....5
1.3 Синтаксический анализатор.....7
1.4 Таблица символов.....9
1.5 Семантический анализ.....10
1.6 Построение генератора промежуточного и целевого кода.....12
2 ПРОВЕРКА ВЫПОЛНЕНИЯ ТРЕБОВАНИЙ К ЯЗЫКУ.....16
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ.....19

1 ПРОЕКТИРОВАНИЕ КОМПИЛЯТОРА

1.1 Основные требования

Разработка будет производиться в соответствии со следующими требованиями:

Требования к входному языку:

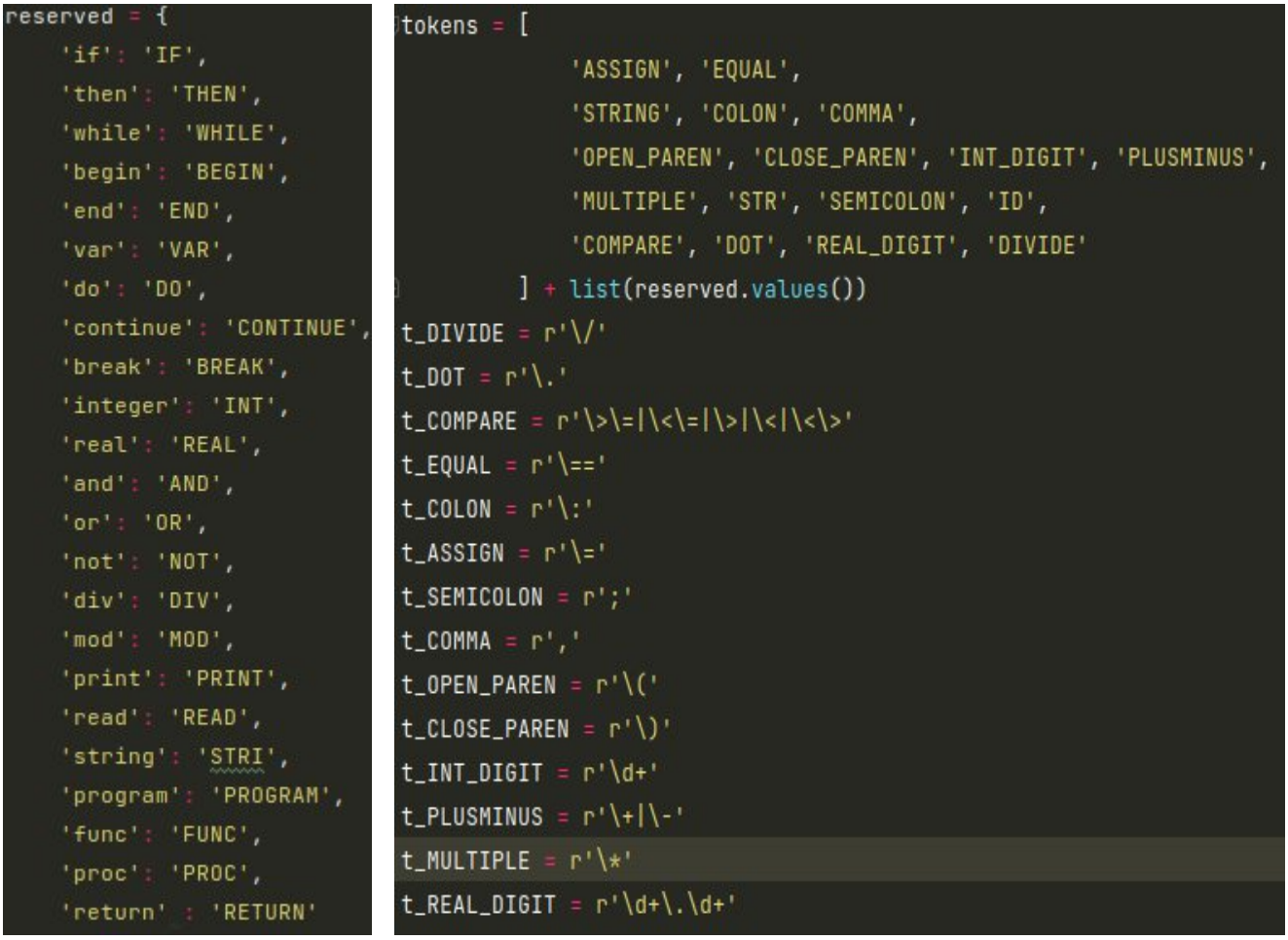
1. Должны присутствовать операторные скобки;
2. Должна игнорироваться индентация программы;
3. Должны поддерживаться комментарии любой длины;
4. Входная программа должна представлять собой единый модуль, но поддерживать вызов функций.

Требования к операторам:

1. Оператор присваивания;
2. Арифметические операторы (*, /, -, >, <, =);
3. Логические операторы (И, ИЛИ, НЕ);
4. Условный оператор (ЕСЛИ);
5. Оператор цикла (while, break, continue);
6. Базовый вывод (строковой литерал, переменная);
7. Типы (целочисленный 32 бита, вещественный 32 бита).

1.2 Лексический анализатор

Лексический анализатор в финальном варианте курсовой работы был реализован при помощи библиотеки ply, однако есть реализация лексера и на ANTLR4.
Для его работы необходим набор зарезервированных слов и набор регулярных выражений, при помощи которых лексер будет разбивать исходный код программы на токены. (рисунок 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'
```

```
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+'
```

Рисунок 1 – Регулярные выражения и список зарезервированных слов.

Результат работы лексического анализатора - рисунок 2.

ID Token	Type token	Value token	Lin	Position
1	PROGRAM	program	1	0
2	ID	Factorials	1	8
3	SEMICOLON	;	1	18
4	VAR	var	2	20
5	ID	a	2	24
6	COMMA	,	2	25
7	ID	b	2	26
8	COMMA	,	2	27
9	ID	c	2	28
10	COLON	:	2	30
11	INT	integer	2	32
12	VAR	var	3	40
13	ID	h	3	44
14	COLON	:	3	46
15	REAL	real	3	48
16	FUNC	func	5	54
17	ID	factorial	5	59
18	OPEN_PAREN	(	5	69
19	ID	a	5	70
20	COLON	:	5	71
21	INT	integer	5	73
22	CLOSE_PAREN	)	5	80
23	RETURN	return	5	82
24	INT	integer	5	89
25	SEMICOLON	;	5	96
26	VAR	var	6	101

Рисунок 2 – результат работы лексического анализатора.

1.3 Синтаксический анализатор

Следующим этапом курсовой работы является реализация синтаксического анализатора. Для данного этапа работы была использована библиотека ply, Но опять таки, есть реализация и на ANTLR4.

Ply и Antlr4 генерирует лексер и парсер на основе набора правил для лексера (рисунок 1) и парсера (рисунок 3). Полный листинг КС-грамматики представлен в приложении 1 или в репозитории.

На вход синтаксическому анализатору подается поток токенов, получившийся в результате работы лексера. Результатом работы синтаксического анализатора является синтаксическое дерево разбора. (рисунок 4).

```
Grammar

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
Rule 22    optionalStatementsWBC -> <empty>
Rule 23    optionalStatementsWBC -> statementListWBC
```

Рисунок 3 –Часть КС-грамматики.

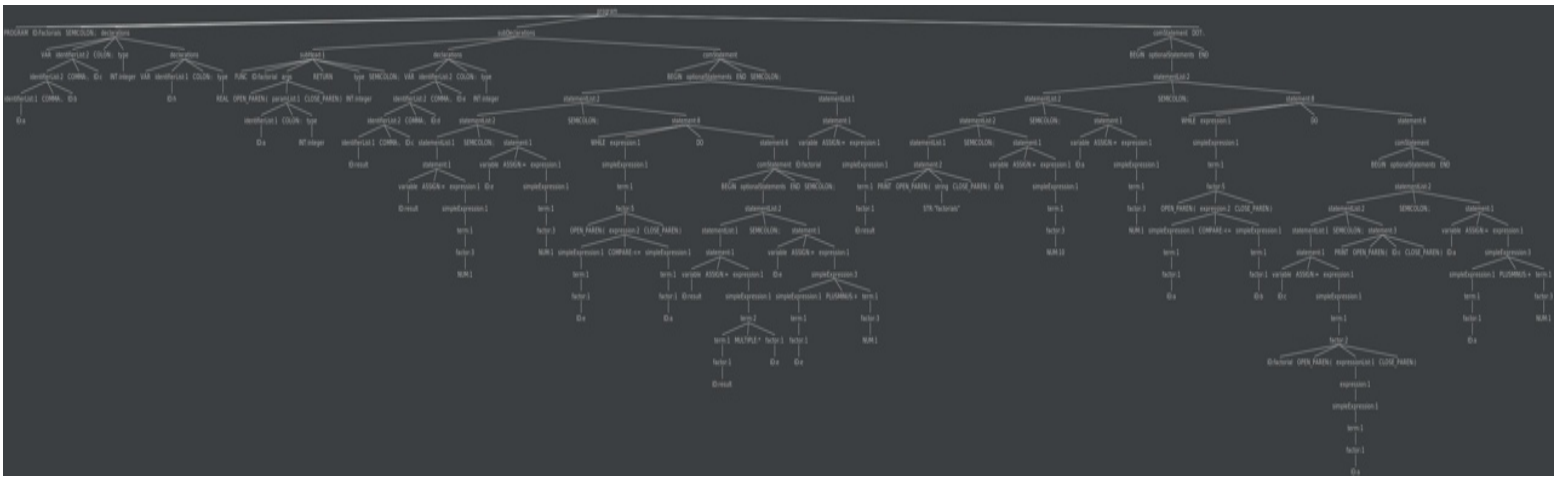


Рисунок 4 – Синтаксическое дерево разбора

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

func factorial (a: integer) return integer;
   var result,c,d,e : integer
   begin
       result = 1;
       e = 1;
       while ( e <= a ) do
       begin
            result = result * e;
            e = e + 1
       end;
       factorial = result
   end;

begin
    print("factorials");
    b = 10;
    a = 1;
    while ( a <= b) do
    begin
        c = factorial(a);
        print(c);
        a = a+1
    end

end.
```

Рисунок 5 – Исходный текст программы

1.4 Таблица символов

В данной курсовой работе таблица символов представляет собой словарь словарей, где хранится тип аргумента и его имя.
Первым ключом является global или название функции, что указывает на глобальность или локальность переменных. Вторым – имя, а значением является тип. (рисунок 6)
Таблица символов генерируется путём обхода дерева.

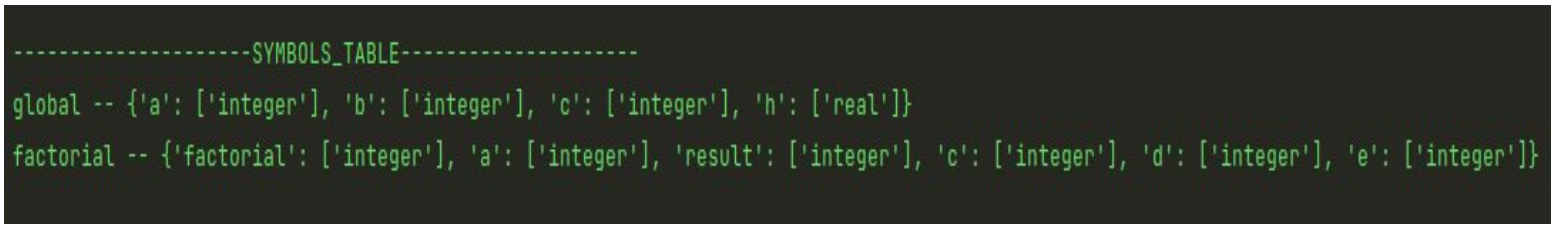


Рисунок 6 – Таблица символов

1.5 Построение генератора промежуточного кода

Для генерации промежуточного кода (трёхадресного кода) был реализован алгоритм обхода синтаксического дерева, который встречая узлы дерева создавал для них трехадресные инструкции. Для хранения трехадресного кода был реализован класс Block (рисунок 7), который хранит в себе списки инструкций, название блока инструкций, его возвращаемый тип и параметры (если блок используется для хранения трехадресного кода функции) и может состоять из других вложенных в него блоков.

```
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
```

Рисунок 7 – Класс Block

Рассмотрим генерацию трехадресного кода для начала программы.
Начало программы : «program Factorials;
var a,b,c : integer»
Кусок кода, отвечающий за генерацию инструкций при встрече подобного текста (рисунок 8).

```
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]],
                    defalt[part.parts[i + 1].parts[0]], newtmpe))
        new.append(('store_' + typeconv[part.parts[i + 1].parts[0]], newtmpe, id))
```

Рисунок 8 – Пример генерации инструкций.

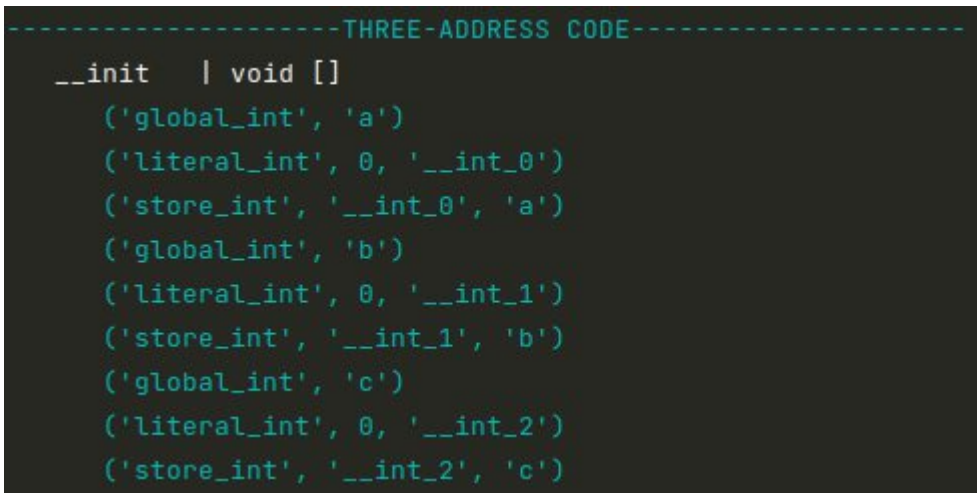


Рисунок 9 – Сгенерированные инструкции.

Полный текст сгенерированных инструкций приведен в приложении 2.

1.6 Построение генератора объектного кода

Для реализации трансляции кода в машинный код мною была использована библиотека llvmlite. Получая трехадресные инструкции из генератора трехадресного кода, транслятор обрабатывает их и создает файл с промежуточным кодом расширения ll.

Рассмотрим генерацию инструкции для выражения в цикле while. Исходный текст программы: «while (a <= b) do»
Кусок кода, отвечающий за генерацию операции сравнения для int’ов (рисунок 10)

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

Рисунок 10 – сравнение двух int’ов

Результат – инструкция llvm (рисунок 11):

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

Рисунок 11 – инструкция llvm для сравнения int’ов

Или пример с вызовом функции.

```
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)
```

Результат:

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

Рисунок 12 – Пример с генерацией объектного кода для вызова функции.

Объектный код целиком представлен в приложении 3

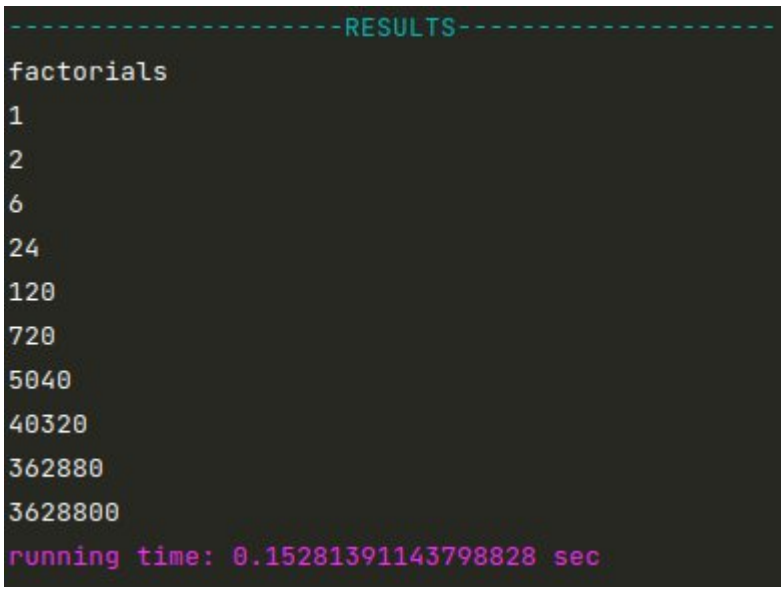
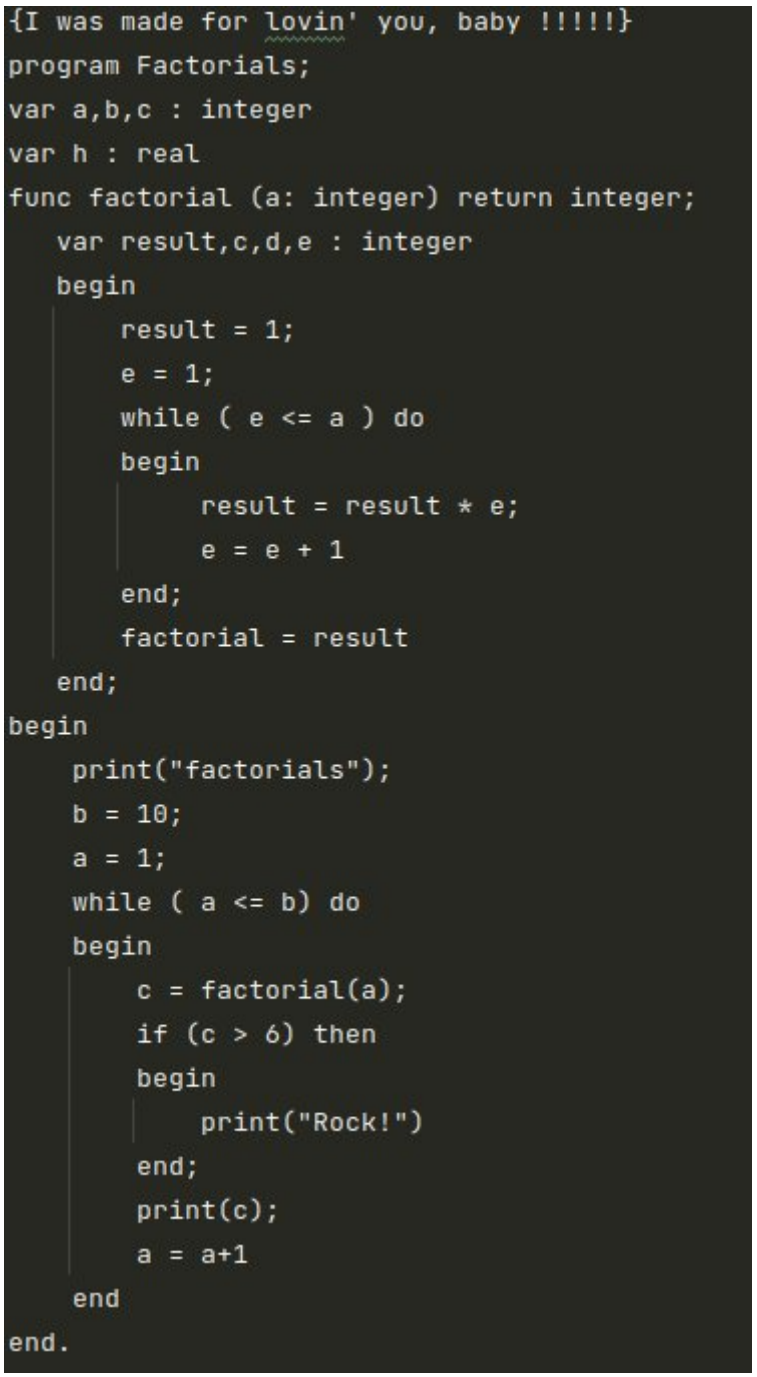


Рисунок 13 – Результат работы программы

ПРОВЕРКА ВЫПОЛНЕНИЯ ТРЕБОВАНИЙ К ЯЗЫКУ

Пример выводит факториалы чисел от 1 до 10 последовательности и показывает выполнение следующих требований:

- Операторы присваивания
- Арифметика
- Операторы цикла (while)
- Условный оператор
- Базовый вывод переменных и строковых литералов
- Поддержка вызова функций
- Игнорирование индентации программы
- Типы (целочисленный 32 бита, с плавающей точкой 32 бита)



```
{I was made for lovin' you, baby !!!!!}
program Factorials;
var a,b,c : integer
var h : real
func factorial (a: integer) return integer;
   var result,c,d,e : integer
   begin
       result = 1;
       e = 1;
       while ( e <= a ) do
       begin
           result = result * e;
           e = e + 1
       end;
       factorial = result
   end;
begin
    print("factorials");
    b = 10;
    a = 1;
    while ( a <= b) do
    begin
        c = factorial(a);
        if (c > 6) then
        begin
            print("Rock!")
        end;
        print(c);
        a = a+1
    end
end.
```

Рисунок 14 – Исходный код программы factorials.kia

```
; 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"
  store double                0x0, double* @"h"
  br label %"exit"
exit:
  ret void
}

@"a" = global i32 0
@"b" = global i32 0
@"c" = global i32 0
@"h" = global double                0x0
define i32 @"factorial"(i32 %".1")
{
entry:
  %"return" = alloca i32
  %"a" = alloca i32
  store i32 %".1", i32* %"a"
  %"factorial" = alloca i32
  store i32 0, i32* %"factorial"
  %"result" = alloca i32
  store i32 0, i32* %"result"
  %"c" = alloca i32
  store i32 0, i32* %"c"
  %"d" = alloca i32
  store i32 0, i32* %"d"
  %"e" = alloca i32
  store i32 0, i32* %"e"
  store i32 1, i32* %"result"
  store i32 1, i32* %"e"
  br label %"whiletest"
exit:
  %".19" = load i32, i32* %"return"
  ret i32 %".19"
whiletest:
  %"__int_10" = load i32, i32* %"e"
  %"__int_11" = load i32, i32* %"a"
  %"__bool_0" = icmp sle i32 %"__int_10", %"__int_11"
  br i1 %"__bool_0", label %"loop", label %"afterloop"
loop:
  %"__int_12" = load i32, i32* %"result"
  %"__int_13" = load i32, i32* %"e"
  %"__int_14" = mul i32 %"__int_12", %"__int_13"
  store i32 %"__int_14", i32* %"result"
  %"__int_15" = load i32, i32* %"e"
  %"__int_17" = add i32 %"__int_15", 1
  store i32 %"__int_17", i32* %"e"
  br label %"whiletest"
afterloop:
  %"__int_18" = load i32, i32* %"result"
  store i32 %"__int_18", i32* %"factorial"
  %"__int_19" = load i32, i32* %"factorial"
  store i32 %"__int_19", i32* %"return"
  br label %"exit"
}

define void @"main"()
{
entry:
  %"__str_0" = alloca [11 x i8]
  store [11 x i8] c"factorials\00", [11 x i8]* %"__str_0"
  %".3" = bitcast [5 x i8]* @"__str_0" to i8*
  %".4" = call i32 (i8*, ...) @"printf"(i8* %".3", [11 x i8]* %"__str_0")
  store i32 10, i32* @"b"
  store i32 1, i32* @"a"
  br label %"whiletest"
exit:
  ret void
whiletest:
  %"__int_22" = load i32, i32* @"a"
  %"__int_23" = load i32, i32* @"b"
  %"__bool_1" = icmp sle i32 %"__int_22", %"__int_23"
  br i1 %"__bool_1", label %"loop", label %"afterloop"
loop:
  %"__int_24" = load i32, i32* @"a"
  %".9" = call i32 @"factorial"(i32 %"__int_24")
  store i32 %".9", i32* @"c"
  %"__int_26" = load i32, i32* @"c"
  %"__bool_2" = icmp sgt i32 %"__int_26", 6
  br i1 %"__bool_2", label %"tblock", label %"fblock"
afterloop:
  br label %"exit"
tblock:
  %"__str_1" = alloca [6 x i8]
  store [6 x i8] c"Rock!\00", [6 x i8]* %"__str_1"
  %".13" = bitcast [5 x i8]* @"__str_1" to i8*
  %".14" = call i32 (i8*, ...) @"printf"(i8* %".13", [6 x i8]* %"__str_1")
  br label %"endblock"
fblock:
  br label %"endblock"
endblock:
  %"__int_28" = load i32, i32* @"c"
  %".17" = bitcast [5 x i8]* @"__int_28" to i8*
  %".18" = call i32 (i8*, ...) @"printf"(i8* %".17", i32 %"__int_28")
  %"__int_29" = load i32, i32* @"a"
  %"__int_31" = add i32 %"__int_29", 1
  store i32 %"__int_31", i32* @"a"
  br label %"whiletest"
}

@"__str_0" = internal constant [5 x i8] c"%s \0a\00"
@"__str_1" = internal constant [5 x i8] c"%s \0a\00"
@"__int_28" = internal constant [5 x i8] c"%i \0a\00"
```

Рисунок 15 – Объектный код

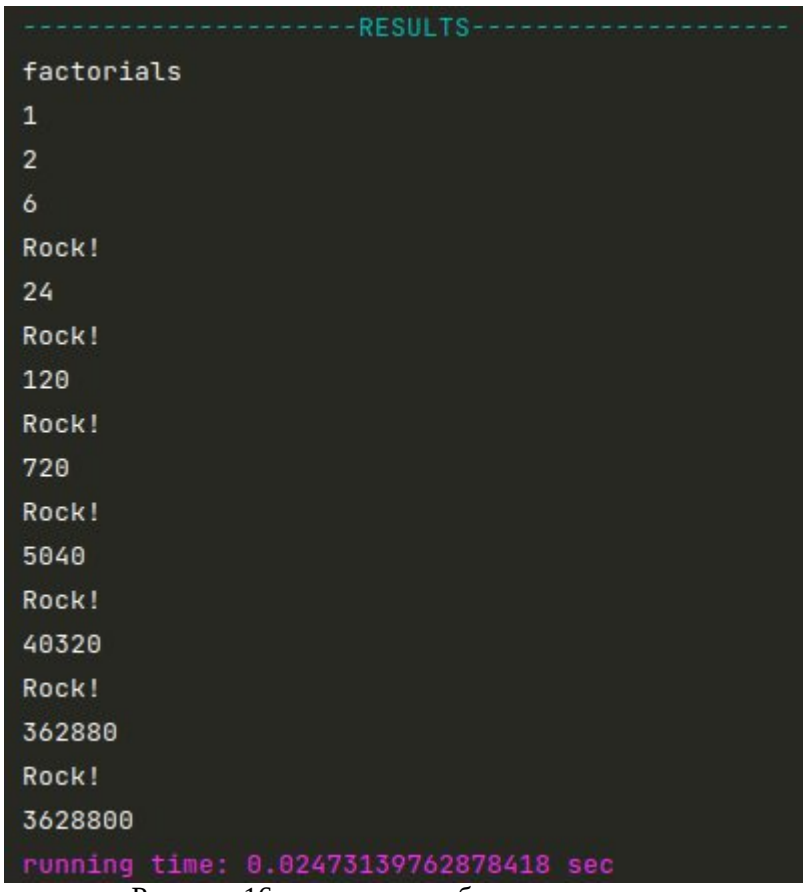


Рисунок 16 – результат работы программы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Ульман Д. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд./ Ульман Д. Д. , Ахо А.В., Лам С.М., Сети Р.-М. :ООО «И.Д. Вильямс», 2008.- 1184 с.
2. PLY (Python Lex-Yacc) «Ply documentation» [Электронный ресурс] Доступ: URL - https://www.dabeaz.com/ply/ply.html#ply_nn5
3. A lightweight LLVM python binding for writing JIT compilers “llvmlite documentation” [Электронный ресурс]Доступ: URL - https://llvmlite.pydata.org/en/v0.14.0/index.html