Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №5

“Интерпретация исходного кода”

Выполнил:

студент гр. 853504

Кузьма В. В.

Проверил:

Ст. преподаватель КИ Шиманский В. В.

Минск 2021

**Содержание**

[**1. Цель работы 3**](#_Toc64740995)

[**2. Краткие теоритические сведения 5**](#_Toc64740996)

[**3. Интерпретация программы 8**](#_Toc64741001)

[**Приложение. Текст программ 9**](#_Toc64741002)

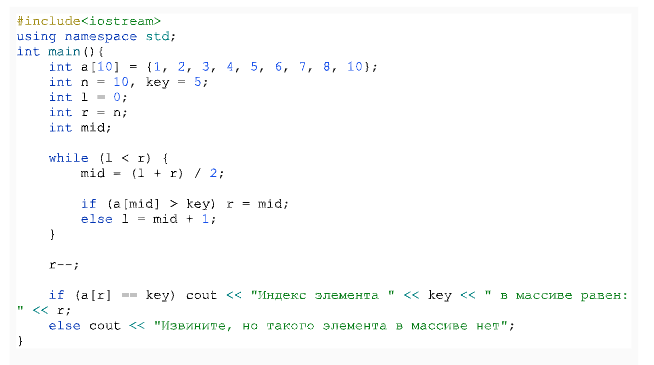
# **Цель работы**

На основе результатов анализа лабораторных работ 1-4 выполнить интерпретацию программы.

Реализация интерпретатора (исходный язык для интерпретации и система получения байт-кода согласовываются).

В качестве тестовых примеров возьмём примеры из первой лабораторной работы:

* Бинарный поиск:

  
  
Рис 1. Реализация бинарного поиска

* Числа Смита:

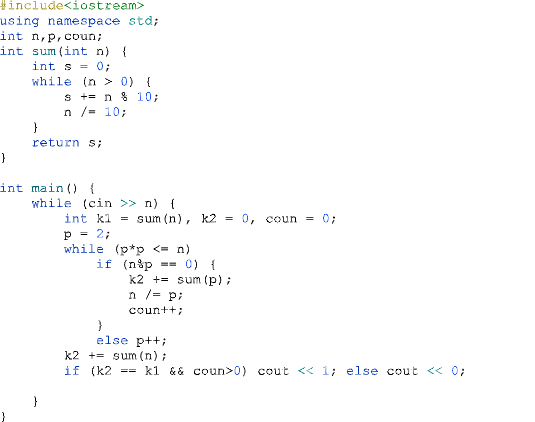


Рис 2. Реализация чисел Смита

# **Краткие теоритические сведения**

**Интерпретатор —** программа (разновидность транслятора), выполняющая интерпретацию.

**Интерпретация** — построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения).

Первым интерпретированным языком программирования высокого уровня был **Lisp**. Его интерпретатор был создан в *1958* году Стивом Расселом на компьютере *IBM* *704*. Рассел вдохновился работой Джона Маккарти и выяснил, что функция *eval* в **Lisp** может быть встроена в машинный код.

**Простой интерпретатор** анализирует и тут же выполняет (собственно интерпретация) программу покомандно (или построчно), по мере поступления её исходного кода на вход интерпретатора. Достоинством такого подхода является мгновенная реакция. **Недостаток** — такой интерпретатор обнаруживает ошибки в тексте программы только при попытке выполнения команды (или строки) с ошибкой.

**Интерпретатор компилирующего типа** — это система из компилятора, переводящего исходный код программы в промежуточное представление, например, в байт-код или p-код, и собственно интерпретатора, который выполняет полученный промежуточный код (так называемая виртуальная машина).

**Достоинством** таких систем является большее быстродействие выполнения программ (за счёт выноса анализа исходного кода в отдельный, разовый проход, и минимизации этого анализа в интерпретаторе).

**Недостатки** — большее требование к ресурсам и требование на корректность исходного кода. Применяется в таких языках, как *Java*, *PHP*, *Tcl*, *Perl*, *REXX* (сохраняется результат парсинга исходного кода), а также в различных *СУБД*.

В случае разделения интерпретатора компилирующего типа на компоненты получаются компилятор языка и простой интерпретатор с минимизированным анализом исходного кода. Причём исходный код для такого интерпретатора не обязательно должен иметь текстовый формат или быть байт-кодом, который понимает только данный интерпретатор, это может быть машинный код какой-то существующей аппаратной платформы. К примеру, виртуальные машины вроде *QEMU*, *Bochs*, *VMware* включают в себя интерпретаторы машинного кода процессоров семейства x86.

Некоторые интерпретаторы (например, для языков *Лисп*, *Scheme*, *Python*, *Бейсик* и других) могут работать в режиме диалога или так называемого цикла **чтения-вычисления-печати** (англ. read-eval-print loop, REPL). В таком режиме интерпретатор считывает законченную конструкцию языка (например, s-expression в языке Лисп), выполняет её, печатает результаты, после чего переходит к ожиданию ввода пользователем следующей конструкции.

Следует также отметить, что режимы интерпретации можно найти не только в программном, но и аппаратном обеспечении. Так, многие микропроцессоры интерпретируют машинный код с помощью встроенных микропрограмм, а процессоры семейства x86, начиная с Pentium (например, на архитектуре Intel P6), во время исполнения машинного кода предварительно транслируют его во внутренний формат (в последовательность микроопераций).

**Достоинства и недостатки интепретаторов:**

***+*** *Большая переносимость интерпретируемых программ* — программа будет работать на любой платформе, на которой есть соответствующий интерпретатор.

**+** Как правило, более совершенные и наглядные *средства диагностики ошибок* в исходных кодах.

**+** *Меньшие размеры кода* по сравнению с машинным кодом, полученным после обычных компиляторов.

- Интерпретируемая программа *не может выполняться отдельно без программы-интерпретатора*. Сам интерпретатор при этом может быть очень компактным.

- Интерпретируемая программа *выполняется медленнее*, поскольку промежуточный анализ исходного кода и планирование его выполнения требуют дополнительного времени в сравнении с непосредственным исполнением машинного кода, в который мог бы быть скомпилирован исходный код.

- Практически *отсутствует оптимизация кода*, что приводит к дополнительным потерям в скорости работы интерпретируемых программ.

**Алгоритм работы простого интерпретатора:**

1. Прочитать инструкцию.
2. Проанализировать инструкцию и определить соответствующие действия.
3. Выполнить соответсвующие действия.
4. Если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к **пункту 2**.

Термины **компилятор** и **интерпретатор** описывают способ, с помощью которого выполняется программа. Теоретически любой язык программирования может быть или *компилятором*, или *интерпретатором*, но некоторые языки обычно являются либо *компиляторами*, либо *интерпретаторами*. Например, **Бейсик** — обычно *интерпретатор*, а **С** — обычно *компилятор*. Способ, с помощью которого программа выполняется, не определяется языком программирования, на котором она написана. **Интерпретаторы и компиляторы** — это просто сложные программы, оперирующие с исходным кодом.

**Интерпретатор** построчно читает исходный код программы и выполняет инструкции, содержащиеся в текущей строке, потом переходит к следующей строке. **Компилятор** читает всю программу и преобразует ее в объектный код, который является такой формой кода, которая может напрямую выполняться компьютером. **Объектный код** также называется двоичным кодом или машинным кодом. Если программа скомпилирована, то сам исходный код больше не влияет на дальнейшую работу программы.

Когда используется интерпретатор, он должен присутствовать все время для выполнения программы. Например, в традиционном *Бейсике* необходимо запустить интерпретатор *Бейсика*, затем загрузить программу и набрать *RUN* для запуска. Интерпретатор построчно проверит на правильность и выполнит программу. Этот медленный процесс возникает каждый раз при запуске программы. В противоположность этому, компилятор преобразует программу в объектный код, который может напрямую выполняться компьютером. Поскольку компилятор переводит программу только один раз, то все, что необходимо сделать, — это запустить программу, обычно путем простого набора имени. Таким образом, **компиляция** кода Должна выполняться только один раз, в то время как **интерпретация** — каждый раз при запуске программы.

**3. Интерпретация программы**

Проведем интерпретацию на примере программ из 1-ой лабораторной работы (код в разделе “**Цель работы**”):

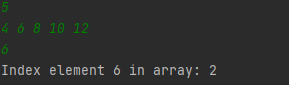


Рис 3. Результат работы бинарного поиска. Пример 1.

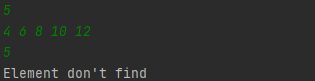


Рис 4. Результаты работы бинарного поиска. Пример 2.

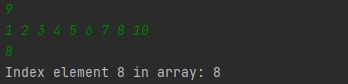


Рис 5. Результаты работы бинарного поиска. Пример 3.  
  
  
  
  
  
Рис 6. Результат работы чисел Смита. Пример 1.



Рис 7. Результат работы чисел Смита. Пример 2.



Рис 8. Результат работы чисел Смита. Пример 3.

**Приложение. Текст программ**

import re  
from ply.lex import LexToken  
  
from Tokenizer import tokens, lexer  
from ply import yacc  
  
  
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  
  
  
start = 'function'  
  
variables = []  
  
  
def token\_in\_stack(name, stack):  
 for item in stack:  
 if isinstance(item, LexToken) and item.value == name:  
 return True  
 return False  
  
  
def p\_cout(p):  
 *'''cout : COUT LESSS args semicolons'''* p[0] = Node('cout', p[3:])  
  
  
def p\_initlist(p):  
 *'''init\_list : LFIGURPAREN args RFIGURPAREN'''* p[0] = Node('init\_list', p[1:])  
  
  
def p\_function(p):  
 *'''function : func\_header func\_body  
  
 '''* func\_list.append(p)  
 p[0] = Node('function', p[1:])  
  
  
def p\_func\_header(p):  
 *'''func\_header :  
 | FUNCTION VARIABLE LPAREN args RPAREN  
 | INTEGER VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_declaration', [p[2], p[4]])  
  
  
def p\_args(p):  
 *'''args :  
 | expr  
 | args COMMA expr  
 | args LESSS expr  
 | args LESS expr semicolons  
 | LESSS args LESS expr'''* if len(p) <= 2:  
 p[0] = Node('args', p[1:] if p[1:] else ['EMPTY'])  
 else:  
 p[0] = p[1].add\_parts([p[3]])  
  
  
def p\_func\_body(p):  
 *'''func\_body : block'''* p[0] = p[1]  
  
  
def p\_if\_s(p):  
 *'''if\_s : IF LPAREN condition RPAREN block  
 | IF LPAREN condition RPAREN line semicolons  
 | IF LPAREN condition RPAREN cout'''* p[0] = Node('if', [p[3], p[5]])  
  
  
def p\_block(p):  
 *'''block : LFIGURPAREN body RFIGURPAREN'''* p[0] = Node('block', [p[2]])  
  
  
def p\_body(p):  
 *'''body :  
 | body line semicolons  
 | body multiline'''* if len(p) > 1:  
 if p[1] is None:  
 p[1] = Node('body', [])  
 p[0] = p[1].add\_parts([p[2]])  
 else:  
 p[0] = Node('body', [])  
  
  
def p\_semicolons(p):  
 *'''semicolons : SEMICOLON  
 | semicolons SEMICOLON'''*def p\_multiline(p):  
 *'''multiline : while\_statement  
 | if\_s  
 | else'''* p[0] = p[1]  
  
  
def p\_line(p):  
 *'''line : assign  
 | func  
 | RETURN expr  
 | RETURN func'''* if len(p) == 3:  
 p[0] = Node('return', [p[2]])  
 else:  
 p[0] = p[1]  
  
  
def p\_else(p):  
 *'''else : ELSE block  
 | ELSE line semicolons  
 | ELSE cout'''* p[0] = Node('else', [p[2]])  
  
  
def p\_while\_statement(p):  
 *'''while\_statement : WHILE LPAREN condition RPAREN block  
 | WHILE LPAREN condition RPAREN line semicolons'''* p[0] = Node('while', [p[3], p[5]])  
  
  
def p\_condition(p):  
 *'''condition : expr cond\_sign expr'''* p[0] = Node('condition', [p[1], p[2], p[3]])  
  
  
def p\_cond\_sign(p):  
 *'''cond\_sign : NOTEQUAL  
 | EQ  
 | LESS  
 | MORE  
 '''* p[0] = p[1]  
  
  
def p\_assign\_array(p):  
 *'''assign : INTEGER variable LSQRPAREN INT RSQRPAREN OPERATOR EQUAL expr  
 '''* if len(p) == 5:  
 value = p[2]  
 if value in variables:  
 raise VariableOverridingError(value, lexer.lineno)  
 else:  
 variables.append(value)  
 p[0] = Node('assign\_array', [p[2], p[4]])  
 else:  
 value = p[1]  
 if value not in variables:  
 raise NotAssignedVariableError(value, lexer.lineno)  
 p[0] = Node('assign\_array', [p[1], p[3]])  
  
  
def p\_assign(p):  
 *'''assign : variable EQUAL expr  
 | VAR variable EQUAL expr  
 | VAR variable EQUAL function  
 | INTEGER variable EQUAL expr  
 | INTEGER variable EQUAL function  
 | INTEGER variable EQUAL init\_list'''* if len(p) == 5:  
 value = p[2]  
 if value in variables:  
 raise VariableOverridingError(value, lexer.lineno)  
 else:  
 variables.append(value)  
 variable\_list.append(){p[2]: p[4]})  
 p[0] = Node('assign', [p[2], p[4]])  
 else:  
 value = p[1]  
 if value not in variables and '[' not in value and '"' not in value:  
 raise NotAssignedVariableError(value, lexer.lineno)  
 variable\_list.append(  
 {p[2]: p[4]})  
 p[0] = Node('assign', [p[1], p[3]])  
  
  
def p\_func(p):  
 *'''func : VARIABLE LPAREN args RPAREN'''* p[0] = Node('func\_call', [p[1], p[3]])  
 eval(p[0])  
  
  
def p\_expr(p):  
 *'''expr : fact  
 | expr OPERATOR fact'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
 eval(p[0])  
  
  
def p\_fact(p):  
 *'''fact : term  
 | fact OPERATOR term'''* if len(p) == 2:  
 p[0] = p[1]  
 else:  
 p[0] = Node(p[2], [p[1], p[3]])  
 eval(p[0])  
  
  
def p\_term(p):  
 *'''term : arg  
 | LPAREN expr RPAREN'''* if len(p) == 2:  
 p[0] = p[1]  
 variables\_list[p[0]] = p[1]  
 else:  
 p[0] = p[2]  
 variables\_list[p[0]] = p[2]  
  
  
def p\_arg(p):  
 *'''arg : STRING  
 | variable  
 | INT  
 | FLOAT  
 | method  
 | func  
 | STRINGLIT'''* if len(p) == 3:  
 p[1], p[2] = p[2], p[1]  
 value = p[1]  
 if isinstance(value, str):  
 if token\_in\_stack('function', p.stack):  
 variables.append(value)  
 elif value not in variables and '[' not in value and '"' not in value:  
  
 raise NotAssignedVariableError(value, lexer.lineno)  
 p[0] = Node('arg', [p[1]])  
  
  
def p\_method(p):  
 *'''method : variable\_or\_arg DOT func'''* p[0] = Node('method', [p[1], p[3]])  
  
  
def p\_variable\_or\_arg(p):  
 *'''variable\_or\_arg : variable  
 | arg'''* p[0] = p[1]  
  
  
def p\_variable(p):  
 *"""variable : VARIABLE"""* p[0] = p[1]  
  
  
parser\_errors = []  
  
  
def p\_error(p):  
 if p.value == 'include':  
 return  
 print('Unexpected token in line %d: %s' % (p.lineno, p))  
  
  
parser = yacc.yacc()  
  
  
def build\_tree(code):  
 p = parser.parse(code)  
 return p  
  
  
operations = {  
 '+': lambda x, y: x + y,  
 '-': lambda x, y: x - y,  
 '\*': lambda x, y: x \* y,  
 '/': lambda x, y: x / y,  
}  
  
\_\_variables = {}  
  
  
def parse\_tree(tree):  
 try:  
 tree\_type = tree.type  
 parts = tree.parts  
 except AttributeError:  
 return tree  
 if tree\_type == 'assign':  
 \_\_variables[parts[0]] = parse\_tree(parts[1])  
 return  
 if tree\_type == 'arg':  
 arg = parts[0]  
 if isinstance(arg, int):  
 return arg  
 elif arg in \_\_variables:  
 return \_\_variables[arg]  
 else:  
 return 'method'  
 if tree\_type in operations:  
 first = parse\_tree(parts[0])  
 second = parse\_tree(parts[1])  
 if tree\_type == '/' and second == 0:  
 raise CustomZeroDivisionError(11)  
 return operations[tree\_type](first, second)  
 for part in parts:  
 parse\_tree(part)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 with open("input.js") as f:  
 lines = f.readlines()  
 with open("input.js") as f:  
 characters = f.read()  
  
 try:  
 tree = build\_tree(characters)  
 parse\_tree(tree)  
 except CustomException as e:  
 print(e)  
 k = str(e).split()  
 if k[0] == "CustomZeroDivisionError":  
 print(f"Division by zero: line {k[3]} char {lines[int(k[3])].find(k[6][1:-1])}")  
 print(lines[int(k[3])][:-1])  
 print(' ' \* lines[int(k[3])].find('/') + '^')  
 if k[0] == "NotAssignedVariableError":  
 print(f"Unknown variable: line {k[3]} char {lines[int(k[3]) - 1].find(k[6][1:-1])}")  
 print(lines[int(k[3]) - 1][:-1])  
 print(' ' \* lines[int(k[3]) - 1].find(k[6][1:-1]) + '^')  
 if k[0] == "VariableOverridingError":  
 print(f"Variable declared: line {k[3]} char {lines[int(k[3]) - 1].find(k[6][1:-1])}")  
 print(lines[int(k[3]) - 1][:-1])  
 print(' ' \* lines[int(k[3]) - 1].find(k[6][1:-1]) + '^')  
 if k[0] == "ArgumentLessError":  
 print(  
 f"Insuffient number of agruments({k[3]}) arguments( {lines[int(k[3]) - 1].find(k[6][1:-1])} line {k[4]} char {k[5]}")  
 print(lines[int(k[4]))  
 print(' ' \* lines[int(k[4]) - 1].find(k[6][1:-1]) + '^')  
 if k[0] == "ArgumentMoreError":  
 print(  
 f"Excessive number of agruments({k[3]}) arguments( {lines[int(k[3]) - 1].find(k[6][1:-1])} line {k[4]} char {k[5]}")  
 print(lines[int(k[4]))  
 print(' ' \* lines[int(k[4]) - 1].find(k[6][1:-1]) + '^')  
  
 for error in parser\_errors:  
 print(error)