

Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет  
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы трансляции

ОТЧЁТ  
по лабораторной работе  
на тему

Интерпретация исходного кода

Выполнил  
Студент гр. 053502  
Шаргородский И.С.

Проверил  
Ассистент кафедры информатики  
Гриценко Н.Ю.

Минск 2023

## СОДЕРЖАНИЕ

1 Цель работы .....	3
2 Демонстрация работы .....	4
2.1 Результаты работы .....	4
2.2 Синтаксические ошибки.....	5
3 Выводы .....	6
Приложение А (информационное)_Код программы .....	7

## **1 ЦЕЛЬ РАБОТЫ**

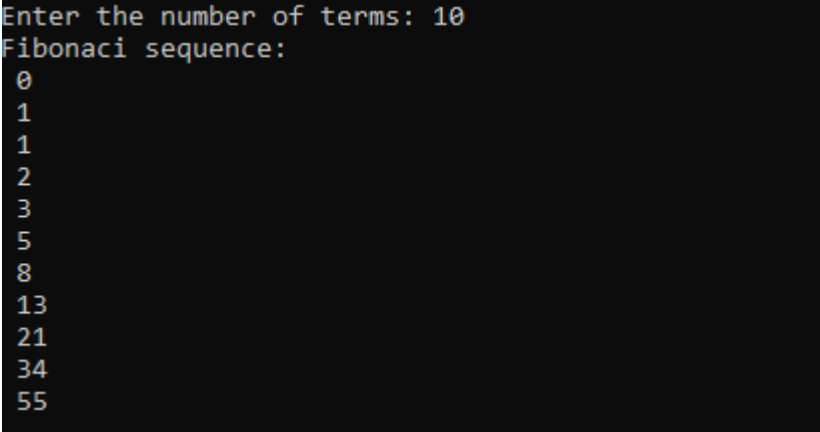
На основе результатов анализа лабораторных работ 1-4 выполнить интерпретацию программы.

## 2 ДЕМОНСТРАЦИЯ РАБОТЫ

### 2.1 Результаты работы

Рассмотрим результат интерпретации программы, вычисляющей последовательность фибоначи (см. рисунок 1). Входная программа:

```
num = int(input("Enter the number of terms: "))
print("Fibonacci sequence:")
a = 0
b = 1
for i in range(1, num + 1):
    print(" ", a)
    c = a + b
    a = b
    b = c
```



```
Enter the number of terms: 10
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
55
```

Рисунок 1 – Результат интерпретации тестовой программы

Рассмотрим результат интерпретации программы, сортирующей массив (см. рисунок 2). Входная программа:

```
a = [1, -4, 5, -5, 23]
print(a)
for i in range(len(a)):
    for j in range(i+1, len(a)-1):
        if (a[i] > a[j]):
            tmp = a[i]
            a[i] = a[j]
            a[j] = tmp

print(a)
```

```
[1, -4, 5, -5, 23]
[-5, -4, 1, 5, 23]
```

Рисунок 2 – Результат интерпритации тестовой программы

## 2.2 Ошибки во время исполнения

Ошибка некорректного типа аргумента функции – производится, когда программа встречает тип данных, не предназначенный для данной функции. Результат анализа ошибки представлен на рисунке 3. Входная программа:

```
num = int(input("Enter the number of terms: "))
```

```
Enter the number of terms: ten
test_ok.py : Incorrect number literal ten | at 0:8
```

Рисунок 3 – Пример ошибки некорректного типа

Ошибка некорректного индекса массива – производится, когда программа встречает обращение к несуществующему индексу массива. Результат анализа ошибки представлен на рисунке 4. Входная программа:

```
a = [1, -4, 5, -5, 23]
for i in range(7):
    print(a[i])
```

```
1
-4
5
-5
23
sort.py : Index out of bounds at | 2:7
```

Рисунок 4 – Пример ошибки некорректного индекса

### **3 ВЫВОДЫ**

Таким образом, в ходе лабораторной работы был реализован интерпритатор, позволяющий исполнять программы на выбранном подмножестве языка.

## ПРИЛОЖЕНИЕ А (информационное) Код программы

```
int PyAnalyzer::Execute()
{
    return ExecScope(SyntaxTree, 0);
}

int PyAnalyzer::ExecScope(std::shared_ptr<SyntaxNode> Scope, int scope_id)
{
    int i = 0;
    if (Scope->Token.TokenType == ETokenType::KeyWord && Scope->Token.ValueName != "else")
        i++;
    for (; i < Scope->Children.size(); i++)
    {
        auto T = Scope->Children[i]->Token;
        if (T.TokenType == ETokenType::Operator)
        {
            if (AssignmentOperators.count(T.ValueName))
            {
                std::string VarName = Scope->Children[i]->Children[0]->Token.ValueName;

                if (Scope->Children[i]->Children[0]->Children.size())
                {
                    std::vector<FVariable>* Varr =
reinterpret_cast<std::vector<FVariable>*>(Vars[VarName].Value);
                    auto index = ExecExpr(Scope->Children[i]->Children[0]->Children[0]);

                    if (index.Type == "")
                    {
                        if (Errors.size() && !(Errors.back().Message.back() >= '0' &&
Errors.back().Message.back() <= '9'))
                            Errors.back().Message = Errors.back().Message + std::string(" | at ") +
std::to_string(T.RowIndex) + ":" + std::to_string(T.ColumnIndex);
                        return 1;
                    }

                    if (index.Type != "int")
                    {
                        Errors.push_back(Error("Incorrect index type, expected 'int', but found : " + index.Type +
" at | " + std::to_string(T.RowIndex) + ":" + std::to_string(T.ColumnIndex)));
                        return 1;
                    }

                    int id = *reinterpret_cast<int*>(index.Value);
                    if (id < 0 || id > Varr->size() - 1)
                    {
                        Errors.push_back(Error("Index out of bounds at | " + std::to_string(T.RowIndex) + ":" +
std::to_string(T.ColumnIndex)));
                        return 1;
                    }

                    Varr->at(id) = ExecExpr(Scope->Children[i]->Children[1]);
                }
            }
        }
    }
}
```

```

        if (Varr->at(id).Type == "") return 1;
        Vars[VarName].Value = Varr;
    }
    else
    {
        if (Vars.find(VarName) == Vars.end())
        {
            Vars[VarName] = ExecExpr(Scope->Children[i]->Children[1]);
            if (Vars[VarName].Type == "") return 1;
            Vars[VarName].Scope = scope_id;
        }
        else
        {
            auto exp = ExecExpr(Scope->Children[i]->Children[1]);
            if (exp.Type == "") return 1;
            Vars[VarName].Value = exp.Value;
            Vars[VarName].Type = exp.Type;
        }
    }
}
else
{
    auto check = ExecExpr(Scope->Children[i]->Children[1]);
    if (check.Type == "") return 1;
}
}
else if (T.TokenType == ETokenType::Function)
{
    FVariable tmp = ExecFunction(Scope->Children[i]);
    if (tmp.Type == "") return 1;
}
else if (T.TokenType == ETokenType::Keyword)
{
    if (T.ValueName == "for")
    {
        if (ExecFor(Scope->Children[i], scope_id+1)) return 1;
    }
    else if (T.ValueName == "while")
    {
        if (ExecWhile(Scope->Children[i], scope_id + 1)) return 1;
    }
    else if (T.ValueName == "if")
    {
        auto exp = ExecExpr(Scope->Children[i]->Children[0]);
        if (exp.Type != "int")
        {
            Errors.push_back(Error("If statement must be of bool type, but found " + exp.Type + " | at "
+ std::to_string(Scope->Children[i]->Token.RowIndex) + ":" +
std::to_string(Scope->Children[i]->Token.ColumnIndex)));
            return 1;
        }
        int st = *reinterpret_cast<int*>(exp.Value);
        if (st)
        {
            if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
            while (i + 1 < Scope->Children.size())

```



```

        && (Scope->Children[i + 1]->Token.ValueName == "else"
            || Scope->Children[i + 1]->Token.ValueName == "elif"))
    {
        i++;
    }
}
}
else if (T.ValueName == "else")
{
    if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
}
else if (T.ValueName == "elif")
{
    auto exp = ExecExpr(Scope->Children[i]->Children[0]);
    if (exp.Type != "int")
    {
        Errors.push_back(Error("If statement must be of bool type, but found " + exp.Type + " | at "
+ std::to_string(Scope->Children[i]->Token.RowIndex) + ":" +
std::to_string(Scope->Children[i]->Token.ColumnIndex)));
        return 1;
    }
    int st = *reinterpret_cast<int*>(exp.Value);
    if (st)
    {
        if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
        while (i + 1 < Scope->Children.size()
            && (Scope->Children[i + 1]->Token.ValueName == "else"
                || Scope->Children[i + 1]->Token.ValueName == "elif"))
        {
            i++;
        }
    }
}
}

////////////////////////////////////
////////////////////////////////////  Clear Scope  //////////////////////////////////////
////////////////////////////////////

auto tmp = Vars;
Vars.clear();

for (auto i : tmp)
{
    if (i.second.Scope != scope_id)
        Vars[i.first] = i.second;
}
return 0;
}

int PyAnalyzer::ExecFor(std::shared_ptr<SyntaxNode> Scope, int scope_id)
{
    int iterc = 0;
    std::string iVarName = Scope->Children[0]->Children[0]->Token.ValueName;

```

```

std::pair<int, int> rng;
std::vector<FVariable> arr;

if (Scope->Children[0]->Children[1]->Token.ValueName == "range")
{
    rng = ExecRange(Scope->Children[0]->Children[1]);
    if (rng.second == -1) return 1;

    if (Vars.find(iVarName) != Vars.end())
    {
        Vars[iVarName].Value = new int(rng.first + iterc);
        Vars[iVarName].Type = "int";
    }
    else
    {
        Vars[iVarName] = FVariable("", "int", new int(rng.first + iterc));
        Vars[iVarName].Scope = scope_id;
    }
}
else
{
    auto exp = ExecExpr(Scope->Children[0]->Children[1]);
    if (Scope->Children[0]->Children[1]->Token.ValueName == "[") exp.Scope = scope_id;
    if (exp.Type == "") return 1;
    if (exp.Type != "array")
    {
        Errors.push_back(Error("Loop statement must be of array type, but found " + exp.Type + " | at " +
std::to_string(Scope->Children[0]->Children[1]->Token.RowIndex) + ":" +
std::to_string(Scope->Children[0]->Children[1]->Token.ColumnIndex)));
        return 1;
    }
    arr = *reinterpret_cast<std::vector<FVariable>*>(exp.Value);

    if (Vars.find(iVarName) == Vars.end())
    {
        Vars[iVarName].Scope = scope_id;
    }
}

while (true)
{
    if (Scope->Children[0]->Children[1]->Token.ValueName == "range")
    {
        if (rng.first + iterc > rng.second) break;
        Vars[iVarName].Value = new int(rng.first + iterc);
        Vars[iVarName].Type = "int";
    }
    else
    {
        if (iterc >= arr.size()) break;
        Vars[iVarName].Value = arr[iterc].Value;
        Vars[iVarName].Type = arr[iterc].Type;
    }
}

```

```

iterc++;

for (int i = 1; i < Scope->Children.size(); i++)
{
    auto T = Scope->Children[i]->Token;
    if (T.TokenType == ETokenType::Operator)
    {
        if (AssignmentOperators.count(T.ValueName))
        {
            std::string VarName = Scope->Children[i]->Children[0]->Token.ValueName;

            if (Scope->Children[i]->Children[0]->Children.size())
            {
                std::vector<FVariable>* Varr =
reinterpret_cast<std::vector<FVariable>*>(Vars[VarName].Value);
                auto index = ExecExpr(Scope->Children[i]->Children[0]->Children[0]);

                if (index.Type == "")
                {
                    if (Errors.size() && !(Errors.back().Message.back() >= '0' &&
Errors.back().Message.back() <= '9'))
                        Errors.back().Message = Errors.back().Message + std::string(" | at ") +
std::to_string(T.RowIndex) + ":" + std::to_string(T.ColumnIndex);
                    return 1;
                }

                if (index.Type != "int")
                {
                    Errors.push_back(Error("Incorrect index type, expected 'int', but found : " + index.Type
+ " at | " + std::to_string(T.RowIndex) + ":" + std::to_string(T.ColumnIndex)));
                    return 1;
                }

                int id = *reinterpret_cast<int*>(index.Value);
                if (id < 0 || id > Varr->size() - 1)
                {
                    Errors.push_back(Error("Index out of bounds at | " + std::to_string(T.RowIndex) + ":" +
std::to_string(T.ColumnIndex)));
                    return 1;
                }

                Varr->at(id) = ExecExpr(Scope->Children[i]->Children[1]);
                if (Varr->at(id).Type == "") return 1;
                Vars[VarName].Value = Varr;
            }
        }
        else
        {
            if (Vars.find(VarName) == Vars.end())
            {
                Vars[VarName] = ExecExpr(Scope->Children[i]->Children[1]);
                if (Vars[VarName].Type == "") return 1;
                Vars[VarName].Scope = scope_id;
            }
            else
            {
                auto exp = ExecExpr(Scope->Children[i]->Children[1]);

```

```

        if (exp.Type == "") return 1;
        Vars[VarName].Value = exp.Value;
        Vars[VarName].Type = exp.Type;
    }
}
else
{
    auto check = ExecExpr(Scope->Children[i]->Children[1]);
    if (check.Type == "") return 1;
}
}
else if (T.TokenType == ETokenType::Function)
{
    FVariable tmp = ExecFunction(Scope->Children[i]);
    if (tmp.Type == "") return 1;
}
else if (T.TokenType == ETokenType::Keyword)
{
    if (T.ValueName == "for")
    {
        if (ExecFor(Scope->Children[i], scope_id + 1)) return 1;
    }
    else if (T.ValueName == "while")
    {
        if (ExecWhile(Scope->Children[i], scope_id + 1)) return 1;
    }
    else if (T.ValueName == "if")
    {
        auto exp = ExecExpr(Scope->Children[i]->Children[0]);
        if (exp.Type != "int")
        {
            Errors.push_back(Error("If statement must be of bool type, but found " + exp.Type + " | at
" + std::to_string(Scope->Children[i]->Token.RowIndex) + ":" +
std::to_string(Scope->Children[i]->Token.ColumnIndex)));
            return 1;
        }
        int st = *reinterpret_cast<int*>(exp.Value);
        if (st)
        {
            if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
            while (i + 1 < Scope->Children.size()
                && (Scope->Children[i + 1]->Token.ValueName == "else"
                    || Scope->Children[i + 1]->Token.ValueName == "elif"))
            {
                i++;
            }
        }
    }
    else if (T.ValueName == "else")
    {
        if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
    }
    else if (T.ValueName == "elif")
    {
        auto exp = ExecExpr(Scope->Children[i]->Children[0]);

```

```

        if (exp.Type != "int")
        {
            Errors.push_back(Error("If statement mast be of bool type, but found " + exp.Type + " | at
" + std::to_string(Scope->Children[i]->Token.RowIndex) + ":" +
std::to_string(Scope->Children[i]->Token.ColumnIndex)));
            return 1;
        }
        int st = *reinterpret_cast<int*>(exp.Value);
        if (st)
        {
            if (ExecScope(Scope->Children[i], scope_id + 1)) return 1;
            while (i + 1 < Scope->Children.size()
                && (Scope->Children[i + 1]->Token.ValueName == "else"
                    || Scope->Children[i + 1]->Token.ValueName == "elif"))
            {
                i++;
            }
        }
    }
}

////////////////////////////////////
////////////////////////////////////  Clear Scope  //////////////////////////////////////
////////////////////////////////////

auto tmp = Vars;
Vars.clear();

for (auto i : tmp)
{
    if (i.second.Scope != scope_id)
        Vars[i.first] = i.second;
}
return 0;
}

```