Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Методы трансляции»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 4

на тему «Семантический анализатор»

Выполнил             Я. Ю. Прескурел

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 9](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 10](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является разработка собственного семантического анализатора для языка программирования Python. Необходимо вывести результат анализа и обработать возможные семантические ошибки.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

К этапам трансляции относятся следующие этапы:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– оптимизация;

– генерация кода.

На этапе генерации компилятор создает код, который представляет собой набор инструкций, понятных для целевой аппаратной платформы, итоговый файл компилируется в исполняемый файл, который может быть запущен на целевой платформе без необходимости наличия кода.

Фаза эмуляции интерпретатора происходит во время выполнения программы. В отличие от компилятора, интерпретатор работает с кодом напрямую, без предварительной генерации машинного кода.

Лексический анализатор – первый этап трансляции. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в лексемы или значащие последовательности. Лексема – это элементарная единица, которая может являться ключевым словом, идентификатором, константным значением. Для каждой лексемы анализатор строит токен, который по сути является кортежем, содержащим имя и значение.[1]

Синтаксический анализатор выясняет, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики языка программирования. Синтаксический анализатор получает на вход результат лексического анализатора и разбирает его в соответствии с грамматикой. Результат синтаксического анализа обычно представляется в виде синтаксического дерева разбора.[2]

Семантический анализ обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка.

В современных языках программирования одним из примеров контекстных условий может служить обязательность описания переменных, то есть для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение.

Число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры.

Абстрактное синтаксическое дерево конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья – с соответствующими операндами. Таким образом, листья являются пустыми операторами и представляют только переменные и константы.

Синтаксические деревья используются в синтаксических анализаторах для промежуточного представления программы между деревом разбора (деревом с конкретным синтаксисом) и структурой данных, которая затем используется в качестве внутреннего представления в компиляторе или интерпретаторе программы для оптимизации и генерации кода. Возможные варианты подобных структур описываются абстрактным синтаксисом.

Абстрактное синтаксическое дерево отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Классическим примером такого отсутствия являются группирующие скобки, так как в абстрактном синтаксическом дереве группировка операндов явно задаётся структурой дерева.

Для языка, который описывается контекстно-свободной грамматикой создание дерева в синтаксическом анализаторе является тривиальной задачей. Большинство правил в грамматике создают новую вершину, а символы в правиле становятся рёбрами. Правила, которые ничего не привносят в дерево, просто заменяются в вершине одним из своих символов. Кроме того, анализатор может создать полное дерево разбора и затем пройти по нему, удаляя узлы и рёбра, которые не используются в абстрактном синтаксисе, для получения абстрактного синтаксического дерева.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ**

В ходе лабораторной работы был реализован конечный вид анализатора кода, который включает в себя лексический, синтаксический и семантический анализы. Были совершены проверки на такие типы ошибок как:

– объявление одноименных переменных или функций в одной области видимости;

– несовпадение параметров и аргументов при вызове функции;

– неверное преобразование типов данных;

– неверный вызов функции;

– неверное применение закрывающихся одинарных и двойных кавычек;

– неверное указание размера массива.

При неверном преобразовании типов данных, когда целочисленной переменной присваивается, например, значение с плавающей точкой, будет выведена ошибка об этом. Пример семантической ошибки при неверном вызове функции представлен на рисунке 3.1.



Рисунок 3.1 – Ошибка при неверном вызове функции

При несовпадении количества параметров и аргументов при вызове функции с учетом того, что параметрам функции не присваивается значение, также будет выведена семантическая ошибка. Пример тестового кода с ошибкой данного типа представлен на рисунке 3.2.

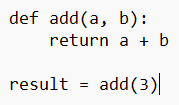


Рисунок 3.2 – Пример тестового кода

Пример семантической ошибки при несовпадении количества параметров и аргументов представлен на рисунке 3.3.



Рисунок 3.3 – Ошибка при различном количестве параметров и аргументов

Ошибка при неверном указании количества элементов в массиве представлена на рисунке 3.4.



Рисунок 3.4 – Ошибка при неверном указании количества элементов в массиве

Таким образом в ходе данной лабораторной работы был организован полноценный анализатор кода, который включает в себя лексический, синтаксический и семантические анализы.

# **ВЫВОДЫ**

В ходе лабораторной работы был реализован семантический анализатор, основанный на результатах синтаксического анализатора. В итоге был получен полный анализатор кода программ на языке Python, включающий в себя лексический, синтаксический и семантический анализы.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 18.03.2024.
2. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 18.03.2024.
3. Введение в Python [Электронный ресурс]. – Режим доступа: https://metanit.com/py/tutorial/2.5.php. – Дата доступа: 18.03.2024.
4. Типы данных [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.3.php>. – Дата доступа: 18.03.2024.
5. Операторы в Python [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 18.03.2024.
6. Функции Python [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 18.03.2024.
7. Классы Python [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/>. – Дата доступа: 18.03.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код SemanticAnalyzer.cs

using System;

using System.Collections.Generic;

namespace Lab4

{

internal class FunctionPrototype

{

internal string className;

internal string functionName;

internal int argCount;

}

internal class SemanticAnalyzer

{

readonly Ast ast;

private static readonly List<FunctionPrototype> functions = new List<FunctionPrototype>()

{

new FunctionPrototype()

{

functionName = "randint",

argCount = 2,

},

new FunctionPrototype()

{

className = "array",

functionName = "append",

argCount = 1,

},

new FunctionPrototype()

{

functionName = "print",

argCount = 1,

},

};

internal SemanticAnalyzer(Ast ast)

{

this.ast = ast;

}

internal bool Analyse()

{

int prevIndentation = 0;

bool expectIndent = false;

foreach (var stat in ast.statements)

{

if (expectIndent)

{

if (stat.indentation != prevIndentation + 1)

{

ReportError($"Unexpected indentation in line {stat.line + 1}, expected {prevIndentation + 1}");

return false;

}

}

else

{

if (stat.indentation > prevIndentation)

{

ReportError($"Unexpected indentation in line {stat.line + 1}, expected {prevIndentation} or less");

return false;

}

}

if (stat.statementType == StatementType.STATEMENT\_TYPE\_IF ||

stat.statementType == StatementType.STATEMENT\_TYPE\_ELSE ||

stat.statementType == StatementType.STATEMENT\_TYPE\_FOR ||

stat.statementType == StatementType.STATEMENT\_TYPE\_WHILE)

{

expectIndent = true;

}

else

{

expectIndent = false;

}

if (stat.statementType == StatementType.STATEMENT\_TYPE\_FUNCTION\_CALL)

{

var functionCall = stat as FunctionCall;

var funcStatement = functionCall.left;

string objectName = null;

string functionName = null;

if (funcStatement.expressionType == Expressiontype.EXPRESSION\_TYPE\_NAME)

{

functionName = funcStatement.value;

}

else if (funcStatement.expressionType == Expressiontype.EXPRESSION\_TYPE\_DOT)

{

objectName = funcStatement.left.value;

functionName = funcStatement.right.value;

}

else

{

ReportError($"Unexpected function type in {stat.line + 1}");

return false;

}

var funcEntry = functions.Find(f => f.functionName == functionName);

if (funcEntry != null)

{

int paramCount = functionCall.parameters.Count;

if (funcEntry.argCount != paramCount)

{

ReportError($"Parameter count in function {functionName} does not match ({funcEntry.argCount} expected, {paramCount} provided) in line {stat.line + 1}");

return false;

}

}

else

{

if (objectName != null)

{

ReportError($"Method {objectName}.{functionName} was not found");

}

else

{

ReportError($"Function {functionName} was not found");

}

return false;

}

}

if (stat.statementType == StatementType.STATEMENT\_TYPE\_ASSIGNMENT)

{

var assignment = stat as Assignment;

if (assignment.left.expressionType != Expressiontype.EXPRESSION\_TYPE\_NAME)

{

ReportError($"Left side of assignment must be a variable name in line {stat.line + 1}");

return false;

}

}

if (stat.statementType == StatementType.STATEMENT\_TYPE\_EXPRESSION)

{

var expression = stat as Expression;

if (expression.expressionType == Expressiontype.EXPRESSION\_TYPE\_ADD ||

expression.expressionType == Expressiontype.EXPRESSION\_TYPE\_SUB ||

expression.expressionType == Expressiontype.EXPRESSION\_TYPE\_MUL ||

expression.expressionType == Expressiontype.EXPRESSION\_TYPE\_DIV)

{

if (expression.left.expressionType != Expressiontype.EXPRESSION\_TYPE\_NUMBER ||

expression.right.expressionType != Expressiontype.EXPRESSION\_TYPE\_NUMBER)

{

ReportError($"Operands of arithmetic operations must be numbers in line {stat.line + 1}");

return false;

}

}

}

prevIndentation = stat.indentation;

}

return true;

}

void ReportError(string error)

{

Console.Error.WriteLine($"Semantic error: {error}");

}

}

}