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

Семантический анализ

Выполнил

Студент гр. 053502

Шаргородский И.С.

Проверил

Ассистент кафедры информатики

Гриценко Н.Ю.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc132364306)

[2 Краткие теоретические сведения 4](#_Toc132364307)

[3 Семантические ошибки 5](#_Toc132364308)

[4 Выводы 6](#_Toc132364309)

[Код программ 7](#_Toc132364310)

**1 ЦЕЛЬ РАБОТЫ**

Освоение работы с существующими синтаксическими деревом.

Определить минимум 2 возможных синтаксических ошибки и показать их корректное выявление.

1. **КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

В процессе семантического анализа проверяется наличие семантических ошибок в исходной программе и накапливается информация о типах для следующей стадии – генерации кода. При семантическом анализе используются иерархические структуры, полученные во время синтаксического анализа для идентификации операторов и операндов выражений и инструкций.

Важным аспектом семантического анализа является проверка типов, когда компилятор проверяет, что каждый оператор имеет операнды допустимого спецификациями языка типа. Например, определение многих языков программирования требует, чтобы при использовании действительного числа в качестве индекса массива генерировалось сообщение об ошибке. В то же время спецификация языка может позволить определенное насильственное преобразование типов, например, когда бинарный арифметический оператор применяется к операндам целого и действительного типов. В этом случае компилятору может потребоваться преобразование целого числа в действительное.

В большинстве языков программирования имеет место неявное изменение типов (иногда называемое приведением типов (coercion)). Реже встречаются языки, подобные Ada, в которых большинство изменений типов должно быть явным.

В языках со статическими типами, например С, все типы известны во время компиляции, и это относится к типам выражений, идентификаторам и литералам. При этом неважно, насколько сложным является выражение: его тип может определяться во время компиляции за определенное количество шагов, исходя из типов его составляющих. Фактически, это позволяет производить контроль типов во время компиляции и находить заранее (в процессе компиляции, а не во время выполнения программы!) многие программные ошибки.

1. **СЕМАНТИЧЕСКИЕ ОШИБКИ**

Ошибка операции между разными типами – производится, когда анализатор встречает операцию, производимую между разными типами данных. Результат анализа ошибки представлен на рисунке 1. Входная программа:

a = 3 + "abc"



Рисунок 1 – Пример ошибки операции с разными типами

Ошибка некорректного индекса – производится, когда анализатор встречает индекс для массива, которые не является целым числом. Результат анализа ошибки представлен на рисунке 2. Входная программа:

arr[1.1] = 4



Рисунок 2 – Пример ошибки некорректного индекса

Ошибка явного деления на ноль – производится, когда анализатор встречает выражение, которое содержитделение на ноль. Результат анализа ошибки представлен на рисунке 3. Входная программа:

arr[1] = 4 / 0



Рисунок 3 – Пример ошибки явного деления на ноль

1. **ВЫВОДЫ**

Таким образом, в ходе лабораторной работы было изучено понятие семантического анализа в теории трансляции. Был разработан собственный семантический анализатор выбранного подмножества языка программирования.

**ПРИЛОЖЕНИЕ А**

**(информационное)**

# Код программ

PyAnalyzer::PyAnalyzer(std::vector<std::string>& \_Code) : Code(\_Code)

{

Analyze();

}

void PyAnalyzer::PrintSyntaxTree()

{

SyntaxTree->Print();

}

int PyAnalyzer::Analyze()

{

LexicalAnalisis();

checkSingleTokenDependensies();

checkBrackets();

SyntaxAnalisis();

ReformatSyntaxTree();

checkSyntaxTree();

SemanticAnalisis();

return 0;;

}

int PyAnalyzer::LexicalAnalisis()

{

std::string Token = "";

CodeDeepth.resize(Code.size());

for (int i = 0; i < Code.size(); i++)

{

bool OnlySpaces = true;

bool MakeFalse = false;

for (int j = 0; i < Code.size() && j < Code[i].size(); j++)

{

bool ReturnFlag = false;

Token.clear();

if (MakeFalse) OnlySpaces = false;

char c = Code[i][j];

if (c == ' ' || c == '\t')

{

if (!MakeFalse)

{

CodeDeepth[i] += c == ' ' ? 1 : 4;

}

continue;

}

MakeFalse = true;

if (Delimiters.count(c))

{

Token = c;

Tokens.push\_back(FToken(Token, "delimeter", i, j, ETokenType::Delimeter));

continue;

}

if ((c >= '0' && c <= '9') || c == '.'

|| (j + 1 < Code[i].size() && (Code[i][j] == '+' || Code[i][j] == '-')

&& Code[i][j + 1] >= '0'

&& Code[i][j + 1] <= '9'

&& Tokens.back().ValueName != ")"

&& Tokens.back().ValueName != "]"

&& Tokens.back().TokenType != ETokenType::Number

&& Tokens.back().TokenType != ETokenType::Variable

&& Tokens.back().TokenType != ETokenType::Literal

))

{

Token = ReadNumberConstant(i, j, ReturnFlag);

if (!ReturnFlag) Tokens.push\_back(FToken(Token, "constant", i, j, ETokenType::Number));

continue;

}

if (c == '\"' || c == '\'')

{

Token = ReadLiteral(i, j, ReturnFlag);

if (!ReturnFlag) Tokens.push\_back(FToken(Token, "literal", i, j, ETokenType::Literal));

continue;

}

if (OperatorsCharacters.count(c))

{

Token = ReadOperator(i, j, ReturnFlag);

if (!ReturnFlag) Tokens.push\_back(FToken(Token, "operator", i, j, ETokenType::Operator));

continue;

}

Token = ReadWord(i, j, ReturnFlag);

if (ReturnFlag) continue;

if (Operators.count(Token))

{

OperatorsTable.Rows.push\_back(FToken(Token, "operator", i, j));

Tokens.push\_back(FToken(Token, "operator", i, j, ETokenType::Operator));

continue;

}

if (Keywords.count(Token))

{

KeywordsTable.Rows.push\_back(FToken(Token, "key word", i, j));

Tokens.push\_back(FToken(Token, "key word", i, j, ETokenType::KeyWord));

if (Token == "def")

{

if (OnlySpaces)

{

j++;

ReadFunctionSignature(i, j, ReturnFlag);

}

else

{

Errors.push\_back(Error(std::string("Incorrect use of |def| keyword : ") + std::to\_string(i) + ":" + std::to\_string(j)));

}

}

else if (Token == "for")

{

j++;

ReadForSignature(i, j, ReturnFlag);

}

continue;

}

if (BuiltinFunctions.count(Token))

{

FunctionsTable.Rows.push\_back(FToken(Token, "build-in function", i, j));

Tokens.push\_back(FToken(Token, "build-in function", i, j, ETokenType::Function));

continue;

}

if (TokenVariables.count(Token))

{

VariablesTable.Rows.push\_back(FToken(Token, "variable", i, j));

Tokens.push\_back(FToken(Token, "variable", i, j, ETokenType::Variable));

continue;

}

if (Functions.count(Token))

{

OperatorsTable.Rows.push\_back(FToken(Token, "function", i, j));

Tokens.push\_back(FToken(Token, "function", i, j, ETokenType::Function));

continue;

}

if (OnlySpaces == false)

{

Errors.push\_back(Error("Unknown token : " + Token + " | " + std::to\_string(i) + ":" + std::to\_string(j)));

continue;

}

if (j + 1 < Code[i].size() && Code[i][j + 1] == '(')

{

Errors.push\_back(Error("Unknown function name : " + Token + " | " + std::to\_string(i) + ":" + std::to\_string(j)));

continue;

}

int jj = j + 1;

while (jj < Code[i].size() && Code[i][jj] == ' ') jj++;

if (jj >= Code[i].size() || Code[i][jj] != '=')

{

Errors.push\_back(Error("Unknown token : " + Token + " | " + std::to\_string(i) + ":" + std::to\_string(j)));

continue;

}

else

{

TokenVariables.insert(Token);

Tokens.push\_back(FToken(Token, "variable", i, j, ETokenType::Variable));

continue;

}

}

}

for (int i = 0; i < CodeDeepth.size(); i++)

{

if (CodeDeepth[i] % 4)

{

Errors.push\_back(Error("Incorrect code deepth on row " + std::to\_string(i + 1)));

}

CodeDeepth[i] /= 4;

}

return 0;

}

int PyAnalyzer::SyntaxAnalisis()

{

SyntaxTree = std::make\_shared<SyntaxNode>(FToken("", "", 0, 0));

SyntaxTree->Parent = SyntaxTree;

SyntaxTree->Token.TokenType = ETokenType::Function;

std::shared\_ptr<SyntaxNode> current = SyntaxTree;;

int deepth = 0;

std::vector<int> brackets;

std::vector<int> BoxBrackets;

std::vector<std::shared\_ptr<SyntaxNode>> FunctionsStack;

for (int i = 0; i < Tokens.size(); i++) {

if (i && Tokens[i - 1].RowIndex != Tokens[i].RowIndex)

{

while (current->Token.ValueName != "")

{

current = current->Parent.lock();

}

int diff = CodeDeepth[Tokens[i - 1].RowIndex] - CodeDeepth[Tokens[i].RowIndex];

if (diff < -1)

{

Errors.push\_back(Error("incorrect code deepth change at line " + std::to\_string(Tokens[i].RowIndex)));

}

if (diff == -1 && Tokens[i-1].ValueName != ":")

{

Errors.push\_back(Error("Unexpected indent at line " + std::to\_string(Tokens[i].RowIndex)));

}

while (diff > 0)

{

current = current->Parent.lock();

diff--;

deepth--;

}

diff = deepth;

while (diff > 0)

{

current = current->Children.back();

diff--;

}

}

if (Tokens[i].ValueName == ":") {

/\*std::shared\_ptr<SyntaxNode> child = std::make\_shared<SyntaxNode>(Tokens[i], current);

current->Children.push\_back(child);\*/

deepth++;

//current = current->Children.back();

}

else if (Tokens[i].ValueName == "(") {

brackets.push\_back(i);

if (i > 0 && Tokens[i - 1].TokenType == ETokenType::Function)

{

current = current->Children.back();

FunctionsStack.push\_back(current);

}

else

{

std::shared\_ptr<SyntaxNode> child = std::make\_shared<SyntaxNode>(Tokens[i], current);

current->Children.push\_back(child);

current = current->Children.back();

}

}

else if (Tokens[i].ValueName == "[") {

if (i > 0 && Tokens[i - 1].TokenType == ETokenType::Variable)

{

FunctionsStack.push\_back(current);

current = current->Children.back();

}

else

{

std::shared\_ptr<SyntaxNode> child = std::make\_shared<SyntaxNode>(Tokens[i], current);

child->Token.ValueName = "[]";

child->Token.TokenType = ETokenType::Function;

child->Token.Description = "create array";

current->Children.push\_back(child);

current = current->Children.back();

FunctionsStack.push\_back(current);

}

}