Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе № 4

По теме «Семантический анализатор»

Выполнил:
студент гр. 053502
Герчик А. В.

Проверил:
Гриценко Н. Ю.

Минск 2023

Содержание

1 Цель работы 3
2 Результаты 4
2.1 Примеры семантических ошибок 4
Примечание. Код программы 6

1 Цель работы

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 C, 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.

В лабораторной работе необходимо показать скриншоты нахождения 2-х **семантических** ошибок.

2 Результаты

2.1 Примеры семантических ошибок

1) Пример с неправильным присвоением, а именно присвоением переменной типа int значения “string” (см. рисунок 2.1, рисунок 2.2).

```
int main()
{
    float a, b, c, Y = 0.0;
    int n = "string";
    cout << "Enter a : ";
```

Рисунок 2.1 - Пример кода с семантической ошибкой

```
"/Users/lnxd/Desktop/BSUIR/SIXTH TERM/MTran/LABS/DONE/LAB4/SemanticAnalyzer/bin/Debug/net6.0/SemanticAnalyzer"
Unhandled exception. System.Exception: not real to do operation = for int and string
```

Рисунок 2.2 - Реакция семантического анализатора на ошибку

2) Пример передачи неверного аргумента в switch (см. рисунок 2.3, рисунок 2.4).

```
int main()
{
    float a, b, c, Y = 0.0;
    int n;
    cout << "Enter a : ";
    cin >> a;
    cout << "Enter b : ";
    cin >> b;
    cout << "Enter c : ";
    cin >> c;
    cout << "Enter N : ";
    cin >> n;
    switch (a)
    {
    case 2:
        Y = (b * c);
        break;
```

Рисунок 2.3 - Пример кода с семантической ошибкой

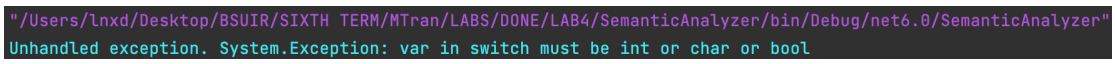

Рисунок 2.4 - Реакция семантического анализатора на ошибку

3) Пример неверного аргумента для cin (см. рисунок 2.5, рисунок 2.6).

```
int main()
{
    float a, b, c, Y = 0.0;
    int n;
    cout << "Enter a : ";
    cin >> a;
    cout << "Enter b : ";
    cin >> b;
    cout << "Enter c : ";
    cin >> c;
    cout << "Enter N : ";
    cin >> n + 228;
```

Рисунок 2.5 - Пример кода с семантической ошибкой

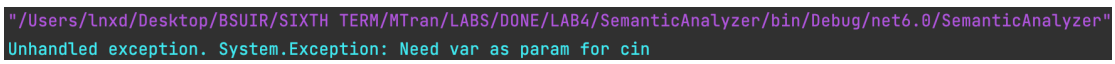

Рисунок 2.6 - Реакция семантического анализатора на ошибку

Примечание. Код программы

```
using LexicalAnalyzer.Functional;
using SyntaxAnalyzer.Nodes;

namespace SemanticAnalyzer.Functional;

internal class Semantic
{
    private AbstractNode Root { get; set; }
       private  Dictionary<string,  List<Token>>  Functions
{ get; set; } = new();

    public Semantic(AbstractNode root)
    {
        Root = root;
    }

    public void CheckCode()
    {
        CheckNodes(Root);
    }

    private void CheckNodes(AbstractNode? node)
    {
        if (node == null) return;

        if (node is StatementsNode statementsNode)
            foreach (var element in statementsNode.Nodes)
                CheckNodes(element);

        if (node is FunctionNode functionNode)
        {
            Functions.Add(functionNode.Function.Identifier,
functionNode.Parameters);
            CheckNodes(functionNode.Body);
        }

        if (node is WhileNode whileNode)
        {
            CheckNodes(whileNode.Condition);
            CheckNodes(whileNode.Body);
```

```
            }

            if (node is CoutNode coutNode)
            {
                var parameters = coutNode.Parameters;

                        foreach (var parameter in parameters)
CheckNodes(parameter);
            }

            if (node is CinNode cinNode)
            {
                var parameters = cinNode.Parameters;
                foreach (var parameter in parameters)
                         if (parameter is BinaryOperationNode
binaryOperationNode &&
                        binaryOperationNode.Operator.Identifier
== "[]")
                        CheckNodes(binaryOperationNode);
                           else if (parameter is VariableNode
variableNode)
                        CheckNodes(variableNode);
                    else
                         throw new Exception("Need var as param
for cin");
            }

            if (node is ForNode forNode)
            {
                CheckNodes(forNode.First);
                CheckNodes(forNode.Second);
                CheckNodes(forNode.Third);
                CheckNodes(forNode.Body);
            }

                         if (node is FunctionExecutionNode
functionExecutionNode)
            {
                                        var parameters =
Functions[functionExecutionNode.Function.Identifier];
                var needParamsCount = parameters.Count;
                                     var executionParams =
functionExecutionNode.Parameters;
```

```
                var gotParamsCount = executionParams.Count;

                if (needParamsCount != gotParamsCount)
                                throw new Exception($"Need
{needParamsCount} params, got {gotParamsCount}");

                  for (var index = 0; index < needParamsCount;
index++)
                {
                                        var returnType =
GetReturnType(executionParams[index]);

                    if (parameters[index].Type != returnType)
                                throw new Exception($"need
{parameters[index].Type}, got {returnType}");
                }
            }

            if (node is SwitchNode switchNode)
            {
                     if (switchNode.Variable.Type != "int" &&
switchNode.Variable.Type != "char" &&
                    switchNode.Variable.Type != "bool")
                     throw new Exception("var in switch must be
int or char or bool");

                CheckNodes(switchNode.Body);
            }

            if (node is CaseNode caseNode)
                  if (caseNode.Literal.Type != "int literal" &&
caseNode.Literal.Type != "char literal" &&
                    caseNode.Literal.Type != "bool literal")
                     throw new Exception("after switch need int
or char or bool literal");

            if (node is KeyWordNode) return;

                        if (node is BinaryOperationNode
binaryOperationNode1)
            {
                                    var returnType1 =
GetReturnType(binaryOperationNode1.LeftNode);
```

```
                                                var   returnType2   =
GetReturnType(binaryOperationNode1.RightNode);

                    if (returnType1 != returnType2)
                    {
                        if ((returnType1 != "int" || returnType2 !=
"float") &&
                            (returnType1 != "int" || returnType2 !=
"char") &&
                                                (returnType1  !=  "float"  ||
returnType2 != "char") &&
                                                (returnType1  !=  "float"  ||
returnType2 != "int") &&
                            (returnType1 != "char" || returnType2 !
= "int") &&
                            (returnType1 != "char" || returnType2 !
= "float"))
                        {
                                                                        if
(binaryOperationNode1.Operator.Identifier != "new" &&

binaryOperationNode1.Operator.Identifier != "[]")
                                    throw new Exception(
                                            $"not real to do operation
{binaryOperationNode1.Operator.Identifier} for {returnType1} and
{returnType2}");

                            if (returnType2 != "int" && returnType2
!= "char")
                                    throw new Exception(
                                            $"not real to do operation
{binaryOperationNode1.Operator.Identifier} for {returnType1} and
{returnType2}");
                        }
                        else
                        {
                                                                        if
(binaryOperationNode1.Operator.Identifier == "new" ||

binaryOperationNode1.Operator.Identifier == "[]")
                                                if (returnType1 != "int" &&
returnType2 != "char")
                                        throw new Exception(
```

```
                                        $"not real to do operation
{binaryOperationNode1.Operator.Identifier} for {returnType1} and
{returnType2}");
                        }
                    }
                }

                if (node is UnaryOperationNode unaryOperationNode)
                {
                                            var   returnType   =
GetReturnType(unaryOperationNode.Operand);

                    if (returnType == "string" || returnType ==
"bool")
                     throw new Exception(
                                    $"not real to do operation
{unaryOperationNode.Operator.Identifier} for {returnType}");
                }

                return;
            }

            private string GetReturnType(AbstractNode abstractNode)
            {
                     if  (abstractNode  is  BinaryOperationNode
binaryOperationNode)
                {
                                            var   returnType1   =
GetReturnType(binaryOperationNode.LeftNode);
                                            var   returnType2   =
GetReturnType(binaryOperationNode.RightNode);

                    if (returnType1 != returnType2)
                    {
                                                                if
((binaryOperationNode.Operator.Identifier == "new" ||

binaryOperationNode.Operator.Identifier  ==  "[]")  &&  returnType2
== "int")
                        {
                                                                if
(binaryOperationNode.Operator.Identifier == "new")
```

```
                                                    return
GetReturnType(binaryOperationNode.LeftNode) + "#";

                                                var  returnType3  =
GetReturnType(binaryOperationNode.LeftNode);

                              if (returnType3.EndsWith('#')) return
returnType3.Remove(returnType3.Length - 1);

                               if (returnType3 == "string") return
"char";

                              throw new Exception($"not real to do
operation [] for {returnType3}");
                        }

                        if ((returnType1 != "int" || returnType2 !=
"float") &&
                            (returnType1 != "int" || returnType2 !=
"char") &&
                                        (returnType1  !=  "float"  ||
returnType2 != "char") &&
                                        (returnType1  !=  "float"  ||
returnType2 != "int") &&
                            (returnType1 != "char" || returnType2 !
= "int") &&
                            (returnType1 != "char" || returnType2 !
= "float"))
                            throw new Exception(
                                         $"not  real  to  do  operation
{binaryOperationNode.Operator.Identifier}  for  {returnType1}  and
{returnType2}");

                        if (binaryOperationNode.Operator.Identifier
== "+" || binaryOperationNode.Operator.Identifier == "-" ||
                            binaryOperationNode.Operator.Identifier
== "*" || binaryOperationNode.Operator.Identifier == "/")
                        {
                                      if  (returnType1  ==  "float"  ||
returnType2 == "float") return "float";

                            return "int";
                        }
```

```
                    if (binaryOperationNode.Operator.Identifier
== "==" ||
                        binaryOperationNode.Operator.Identifier
== "!=" ||
                        binaryOperationNode.Operator.Identifier
== "<" || binaryOperationNode.Operator.Identifier == ">")
                        return "int";

                                                        return
GetReturnType(binaryOperationNode.LeftNode);
                }

                 if (binaryOperationNode.Operator.Identifier ==
"==" || binaryOperationNode.Operator.Identifier == "!=" ||
                     binaryOperationNode.Operator.Identifier ==
"<" || binaryOperationNode.Operator.Identifier == ">")
                    return "int";

                //check later
                if ((binaryOperationNode.Operator.Identifier ==
"new" ||
                     binaryOperationNode.Operator.Identifier ==
"[]") && returnType2 == "int")
                {
                    if (binaryOperationNode.Operator.Identifier
== "new")
                                                        return
GetReturnType(binaryOperationNode.LeftNode) + "#";

                                                var  returnType3  =
GetReturnType(binaryOperationNode.LeftNode);

                        if (returnType3.EndsWith('#')) return
returnType3.Remove(returnType3.Length - 1);

                    if (returnType3 == "string") return "char";

                        throw new Exception($"not real to do
operation [] for {returnType3}");
                }

                return returnType1;
```

```
            }

                        if (abstractNode is UnaryOperationNode
unaryOperationNode)
            {
                                                var returnType =
GetReturnType(unaryOperationNode.Operand);

                    if (returnType == "string" || returnType ==
"bool")
                            throw new Exception($"not real to do
{unaryOperationNode.Operator.Identifier} for {returnType}");

                return returnType;
            }

            if (abstractNode is IfNode ifNode)
            {
                CheckNodes(ifNode.Condition);
                CheckNodes(ifNode.Body);
                CheckNodes(ifNode.ElseBody);

                var returnType1 = GetReturnType(ifNode.Body);
                                            var returnType2 =
GetReturnType(ifNode.ElseBody!);


                if (returnType1 != returnType2)
                    if ((returnType1 != "int" || returnType2 !=
"float") &&
                        (returnType1 != "int" || returnType2 !=
"char") &&
                                        (returnType1 != "float" ||
returnType2 != "char") &&
                                        (returnType1 != "float" ||
returnType2 != "int") &&
                        (returnType1 != "char" || returnType2 !
= "int") &&
                        (returnType1 != "char" || returnType2 !
= "float"))
                        throw new Exception(
                                    $"different return types :
{returnType1} and {returnType2}");
```

```
            }


                if (abstractNode is VariableNode variableNode)
return variableNode.Variable.Type;

            if (abstractNode is LiteralNode literalNode) return
literalNode.Literal.Type.Split()[0];

                        if (abstractNode is VariableTypeNode
variableTypeNode)                                          return
variableTypeNode.VariableType.Identifier;

            return "nothing";
        }
    }
```