**C#**

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №6**

з дисципліни **«**Системне програмування 2**»**

**Варіант 11**

Виконала:

студентка 2 курсу гр. ІВ-71

Молчанова В.С.

Перевірив:

Павлов В.Г.

Київ 2019 р.

**Тема:** Побудова, на­строювання та використання се­ман­тич­ного аналі­за­то­ра в транслято­рах

**Мета**: Одержання навичок настройки таблиць для семантичних аналізаторів для обробки для визначення описів даних програм і дерев підлеглості з запам’ятовуванням типів даних для результатів кожного графа внутрішнього подання програми, в тому числі таблиць відповідності операндів і операцій. Вивчення процедур розпізнавання типів і формування внутрішнього подання констант у відповідній машинній формі інструментальної машини.

**Варіант:** 

**Лістинг програми:**

public class C\_SemanticAnalysis

{

public C\_Glossary CurrentGlossary { get; }

public Dictionary<string, IVariable> Variables;

public Dictionary<int, string> Errors;

public int FirstErrorIndex = -1;

public C\_SemanticAnalysis(C\_SyntacticAnalysis syntacticAnalysis)

{

Errors = new Dictionary<int, string>();

Variables = new Dictionary<string, IVariable>();

CurrentGlossary = syntacticAnalysis.CurrentGlossary;

Analyze(syntacticAnalysis.Root);

}

public C\_SemanticAnalysis(string expression, C\_Glossary currentGlossary)

{

Errors = new Dictionary<int, string>();

Variables = new Dictionary<string, IVariable>();

CurrentGlossary = currentGlossary;

var lexicalAnalysis = new LexicalAnalysis(expression, currentGlossary);

var syntacticAnalysis = new C\_SyntacticAnalysis(lexicalAnalysis.Result, currentGlossary);

Analyze(syntacticAnalysis.Root);

}

protected void Analyze(Node rootNode)

{

GetType(rootNode);

}

protected VariableType GetType(Node node)

{

if (Errors.Any())

{

PrintErrors();

return VariableType.Void;

}

if (TryConvertToVariable(node, out var variable)) return variable.Type;

if (node.Value == ";")

{

GetType(node.Left);

if (node.Right != null) GetType(node.Right);

return VariableType.Void;

}

if (node.Type == NodeType.Type && node.Parent.Type != NodeType.AssignationOperator)

{

HandleType(node);

return VariableType.Void;

}

if (node.Type == NodeType.AssignationOperator) return HandleAssignationOperator(node);

if (node.Type == NodeType.BinaryOperator) return HandleBinaryOperator(node);

if (node.Type == NodeType.Indexer) return GetArrayMember(node)?.Type ?? VariableType.Error;

if (node.Type == NodeType.Name) return GetVariableByName(node)?.Type ?? VariableType.Error;

if (node.Type == NodeType.Root)

{

GetType(node.Left ?? node.Right);

if (!Errors.Any())

{

Console.WriteLine("No errors were found");

}

return VariableType.Void;

}

return VariableType.Void;

}

protected void AddError(int index, string error)

{

Errors.Add(index, error);

if (FirstErrorIndex == -1) FirstErrorIndex = index;

}

protected void PrintErrors()

{

var error = Errors[FirstErrorIndex];

var zeroRow = new string(' ', SyntacticAnalysis.ExprLabel.Length + FirstErrorIndex) + "^";

var firstRow = "SEMANTIC ERROR: " + error;

Console.WriteLine(zeroRow + "\n" + firstRow);

}

protected object HandleType(Node node)

{

var type = GetType(node, out node);

while (node.Value == ",")

{

CreateVariable(node.Left, type);

node = node.Right;

}

CreateVariable(node, type);

return null;

}

protected IVariable CreateVariable(Node currentNode, VariableType type)

{

IVariable newVariable = new Variable("", VariableType.Error);

if (currentNode.Type == NodeType.Pointer && currentNode.Value == "\*")

{

newVariable = new Pointer(currentNode.Left.Value, type);

Variables.Add(currentNode.Left.Value, newVariable);

}

else if (currentNode.Type == NodeType.Indexer)

{

if (Variables.Keys.Contains(currentNode.Left.Value))

{

AddError(currentNode.Left.StartPosition, CurrentGlossary.VariableWasAlreadyDefined(currentNode.Left.Value));

}

if (TryGetIndex(currentNode.Right, out var length))

{

newVariable = new Array(currentNode.Left.Value, type, length);

Variables.Add(currentNode.Left.Value, newVariable);

}

}

else

{

newVariable = new Variable(currentNode.Value, type);

Variables.Add(currentNode.Value, newVariable);

}

return newVariable;

}

protected VariableType GetType(Node node, out Node nameNode)

{

var stringType = "";

while (node.Type == NodeType.Type)

{

stringType += node.Value + " ";

node = node.Left;

}

stringType = stringType.Trim();

nameNode = node;

return IdentifyType(stringType);

VariableType IdentifyType(string type)

{

if (type == "bool") return VariableType.Bool;

if (type == "char") return VariableType.Char;

if (type == "int" || type == "signed int" || type == "signed") return VariableType.Int;

if (type == "unsigned int" || type == "unsigned") return VariableType.UnsignedInt;

if (type == "short" || type == "short int" || type == "signed short int" || type == "signed short") return VariableType.Short;

if (type == "unsigned short" || type == "unsigned short int") return VariableType.UnsignedShort;

if (type == "long" || type == "long int" || type == "signed long int" || type == "signed long") return VariableType.Long;

if (type == "long short" || type == "unsigned long int") return VariableType.UnsignedLong;

if (type == "double") return VariableType.Double;

if (type == "float") return VariableType.Float;

return VariableType.Error;

}

}

protected VariableType HandleBinaryOperator(Node node)

{

var leftType = GetType(node.Left);

var rightType = GetType(node.Right);

if (leftType == VariableType.Pointer &&

!(rightType == VariableType.Int && (node.Value == "+" || node.Value == "-")))

{

AddError(node.StartPosition, CurrentGlossary.CannotApplyOperator(node.Value, leftType, rightType));

return VariableType.Error;

}

if (rightType == VariableType.Pointer &&

!(leftType == VariableType.Int && (node.Value == "+" || node.Value == "-")))

{

AddError(node.StartPosition, CurrentGlossary.CannotApplyOperator(node.Value, leftType, rightType));

return VariableType.Error;

}

var resultType = TypesToBigger(leftType, rightType);

return resultType;

}

protected VariableType HandleAssignationOperator(Node node)

{

var targetType = VariableType.Error;

var sourceType = GetType(node.Right);

switch (node.Left.Type)

{

case NodeType.Type:

{

targetType = GetType(node, out var nameNode);

var newVariable = CreateVariable(nameNode, targetType);

break;

}

case NodeType.Name:

{

var variable = GetVariableByName(node.Left);

if (variable is Pointer pointer)

{

targetType = VariableType.Pointer;

}

else

{

targetType = variable.Type;

}

break;

}

case NodeType.Indexer:

{

if (!(GetVariableByName(node.Left) is Array array))

{

AddError(node.StartPosition, CurrentGlossary.NotArray(node.Left.Value));

return VariableType.Error;

}

targetType = array.Type;

break;

}

case NodeType.Pointer:

{

if (node.Value == "\*")

{

if (!(GetVariableByName(node.Left) is Pointer pointer))

{

AddError(node.StartPosition, CurrentGlossary.NotPointer(node.Left.Value));

return VariableType.Error;

}

targetType = pointer.Type;

}

break;

}

}

if (TryCast(sourceType, targetType)) return VariableType.Void;

AddError(node.StartPosition, CurrentGlossary.CastError(sourceType, targetType));

return VariableType.Error;

}

protected VariableType TypesToBigger(VariableType left, VariableType right)

{

if (left > right) return left;

return right;

}

protected bool TryConvertToVariable(Node node, out IVariable variable)

{

switch (node.Type)

{

case NodeType.CharLiteral:

{

if (char.TryParse(node.Value, out var charValue))

{

variable = new Variable("", VariableType.Char, charValue);

return true;

}

AddError(node.StartPosition, CurrentGlossary.ParseError(node.Value, VariableType.Char));

variable = null;

return false;

}

case NodeType.NumericalConstant:

{

if (short.TryParse(node.Value, out var shortValue))

{

variable = new Variable("", VariableType.Short, shortValue);

return true;

}

if (ushort.TryParse(node.Value, out var ushortValue))

{

variable = new Variable("", VariableType.UnsignedShort, ushortValue);

return true;

}

if (int.TryParse(node.Value, out var intValue))

{

variable = new Variable("", VariableType.Int, intValue);

return true;

}

if (uint.TryParse(node.Value, out var uintValue))

{

variable = new Variable("", VariableType.UnsignedInt, uintValue);

return true;

}

if (long.TryParse(node.Value, out var longValue))

{

variable = new Variable("", VariableType.Long, longValue);

return true;

}

if (ulong.TryParse(node.Value, out var ulongValue))

{

variable = new Variable("", VariableType.UnsignedLong, ulongValue);

return true;

}

if (float.TryParse(node.Value, out var floatValue))

{

variable = new Variable("", VariableType.Float, floatValue);

return true;

}

if (double.TryParse(node.Value, out var doubleValue))

{

variable = new Variable("", VariableType.Double, doubleValue);

return true;

}

break;

}

case NodeType.Name:

{

variable = GetVariableByName(node);

return true;

}

}

variable = new Variable("", VariableType.Error);

return false;

}

protected IVariable GetVariableByName(Node node)

{

if (Variables.ContainsKey(node.Value))

{

return Variables[node.Value];

}

AddError(node.StartPosition, CurrentGlossary.UnknownVariable(node.Value));

return null;

}

protected Variable GetArrayMember(Node node)

{

var name = node.Left.Value;

if (Variables.ContainsKey(name))

{

if (Variables[name] is Array array)

{

if (TryGetIndex(node.Right, out var index))

{

if (index >= 0 && index < array.Length)

{

return array.Value[index];

}

//AddError(node.Right.StartPosition, CurrentGlossary.IndexOutOfBounds());

}

}

else

{

AddError(node.StartPosition, CurrentGlossary.NotArray(name));

}

return null;

}

AddError(node.Left.StartPosition, CurrentGlossary.UnknownVariable(node.Value));

return null;

}

protected bool TryGetIndex(Node node, out int index)

{

if (TryConvertToVariable(node, out var indexVariable))

{

if (indexVariable is Pointer)

{

AddError(node.StartPosition, CurrentGlossary.IndexIsPointer());

index = -1;

return false;

}

if (TryCast( indexVariable.Type, VariableType.Int))

{

//int.TryParse(indexVariable.Value.ToString(), out index);

index = 0;

return true;

}

AddError(node.StartPosition,

CurrentGlossary.CastError(indexVariable.Type, VariableType.Int));

}

index = -1;

return false;

}

protected bool TryCast(VariableType sourceType, VariableType targetType)

{

if (sourceType == VariableType.Pointer &&

(targetType == VariableType.Float || targetType == VariableType.Double))

{

return false;

}

if (targetType == VariableType.Pointer &&

(sourceType == VariableType.Float || sourceType == VariableType.Double))

{

return false;

}

if (sourceType <= VariableType.Double && targetType <= VariableType.Double)

{

return true;

}

return false;

}

**Контрольні приклади:**

**Приклад 1**

Entered expression: double b, a[4]; short \*n,d; b=2\*a[n]; b=d;

^

SEMANTIC ERROR: Index of array cannot be a pointer

**Приклад 2**

Entered expression: float b, a, c, \*d; b=(2\*a +c/d)\*2\*a;

^

SEMANTIC ERROR: Cannot apply operator '/' to 'Float' and 'Pointer'

**Приклад 3**

Entered expression: double b, a[6]; char \*n; --n; b=n&&b==a[\*n];

^

SEMANTIC ERROR: Cannot apply operator '&&' to 'Pointer' and 'Double'