**C#**

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №4**

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:** Створення і настроювання висхідних синтаксичних аналізаторів

**Мета**: Одержання навичок настроювання таблиць висхідних синтаксичних аналізаторів та програм побудови графів та дерев розбору на етапі синтаксичного аналізу з запам’ятовуванням покажчиків на вхідні образи та внутрішні коди. Вивчення програм формування повідомлень про помилки та побудови графа підлеглості операцій та операторів в процесі висхідного синтаксичного аналізу.

**Варіант:** 

**Лістинг програми:**

public class SyntacticAnalysis

{

public List<string> Errors;

public Node Root;

//Delegate that returns errors, found in node or null if there was no errors

public delegate List<string> Check(Node node);

public delegate Node Handler(Token token, Node lastNode);

public readonly Dictionary<TokenType, Handler> TokenTypesHandlers;

public readonly Dictionary<NodeType, Check> NodeTypesChecks;

public SyntacticAnalysis(List<Token> tokens)

{

Errors = new List<string>();

TokenTypesHandlers = new Dictionary<TokenType, Handler>

{

{TokenType.ClosingBracket, HandleClosingBracket},

{TokenType.OpeningBracket, HandleOpeningBracket},

{TokenType.NumericalConstant, HandleNumericalConstant },

{TokenType.Name, HandleName },

{TokenType.Operator, HandleOperator },

{TokenType.Punctuation, HandlePunctuation },

{TokenType.ReservedWord,HandleReservedWord },

{TokenType.Quote, HandleQuote },

{TokenType.ErrorLexeme, HandleErrorLexeme }

};

NodeTypesChecks = new Dictionary<NodeType, Check>

{

{NodeType.Brackets, CheckBrackets},

{NodeType.Indexer, CheckIndexer },

{NodeType.BinaryOperator, CheckBinaryOperator },

{NodeType.AssignationOperator, CheckAssignationOperator },

{NodeType.Punctuation, CheckPunctuation },

{NodeType.Quotes, CheckQuotes },

{NodeType.UnaryOperator, CheckUnaryOperator },

{NodeType.Root, CheckRoot },

{NodeType.FunctionCall, CheckFunctionCalls },

{NodeType.Condition, CheckConditions }

};

Root = Analyze(tokens);

}

public Node Analyze(List<Token> tokens)

{

var rootNode = new Node("root", NodeType.Root, null);

var lastNode = new Node("", NodeType.Empty, rootNode);

foreach (var token in tokens)

{

//if (lastNode.Value == "if")

lastNode = TokenTypesHandlers[token.Type](token, lastNode);

}

Errors.AddRange(GetErrors(rootNode));

return rootNode;

}

#region Handlers

private Node HandleOpeningBracket(Token token, Node lastNode)

{

if (token.Value == "[") return HandleBinaryOperator(token, lastNode, NodeType.Indexer);

if (token.Value == "(" &&

lastNode.Type == NodeType.Name) return HandleBinaryOperator(token, lastNode, NodeType.FunctionCall);

return new Node(token.Value, NodeType.Brackets, lastNode);

}

private Node HandleClosingBracket(Token token, Node lastNode)…

private Node HandleOperator(Token token, Node lastNode)

{

if (UnaryOperators.Contains(token.Value)) return HandleUnaryOperator(token, lastNode, NodeType.UnaryOperator);

if (BinaryOperatorsByPriority.Contains(token.Value)) return HandleBinaryOperator(token, lastNode);

return lastNode;

}

private Node HandleUnaryOperator(Token token, Node lastNode, NodeType nodeType)

{

var type = nodeType == NodeType.Empty ? IdentifyType() : nodeType;

if (lastNode.Type == NodeType.Empty)

{

lastNode.Update(token.Value, type);

return lastNode;

}

if (type == NodeType.UnaryOperator)

{

if (lastNode.Parent.Type == NodeType.Empty)

{

lastNode.Parent.Parent.RemoveChild(lastNode.Parent);

new Node(token.Value, type, lastNode);

return lastNode;

}

}

return new Node(token.Value, type, lastNode);

NodeType IdentifyType()

{

if (Types.Contains(token.Value)) return NodeType.Type;

if (TypeModifiers.Contains(token.Value)) return NodeType.TypeModifier;

return NodeType.UnaryOperator;

}

}

private Node HandleBinaryOperator(Token token, Node lastNode, NodeType nodeType = NodeType.Empty)

{

var type = nodeType == NodeType.Empty ? IdentifyType() : nodeType;

if (lastNode.Parent.Type == NodeType.Empty)

{

lastNode.Parent.Update(token.Value, type);

return lastNode.Parent;

}

if (token.Value == ":")

{

var currentNode = lastNode;

while (currentNode != null)

{

if (currentNode.Value == "?")

{

var newNode = new Node(token.Value, NodeType.BinaryOperator, null);

currentNode.Parent.InsertBeforeChild(currentNode, newNode);

return newNode;

}

currentNode = currentNode.Parent;

}

return new Node(token.Value, NodeType.BinaryOperator, lastNode);

}

lastNode = SkipUnaryToParent(lastNode);

var result = new Node(token.Value, type, null);

var lastPriority = BinaryOperatorsByPriority.IndexOf(lastNode.Parent.Value);

var newPriority = BinaryOperatorsByPriority.IndexOf(token.Value);

if (lastPriority != -1 && newPriority > lastPriority)

{

lastNode.Parent.Parent.InsertBeforeChild(lastNode.Parent, result);

}

else

{

lastNode.Parent.InsertBeforeChild(lastNode, result);

}

return result;

NodeType IdentifyType()

{

if (AssignationOperators.Contains(token.Value)) return NodeType.AssignationOperator;

if (Brackets.Contains(token.Value)) return NodeType.Brackets;

if (token.Value == "[") return NodeType.Indexer;

if (Punctuation.Contains(token.Value)) return NodeType.Punctuation;

return NodeType.BinaryOperator;

}

}

private Node HandlePunctuation(Token token, Node lastNode)…

private Node HandleReservedWord(Token token, Node lastNode)…

private Node HandleNumericalConstant(Token token, Node lastNode)…

#endregion

#region Checks

private List<string> CheckBrackets(Node node)

{

var result = CheckClosing(node);

if (node.Value.Length == 2 && (node.Value=="{}" || node.Parent.Value != "if"))

{

node.Parent.RemoveChild(node);

}

return result;

}

private List<string> CheckFunctionCalls(Node node)

{

var result = CheckClosing(node);

return result;

}

private List<string> CheckQuotes(Node node)…

private List<string> CheckIndexer(Node node)

{

var result = CheckClosing(node);

if (!node.Children.Any()) result.Add($"Array name before '{node.Value}' wasn't found");

var beforeIndexer = SkipUnaryToChild(node.Left);

if (beforeIndexer.Type != NodeType.Name && beforeIndexer.Type != NodeType.Type && beforeIndexer.Type != NodeType.NumericalConstant)

result.Add($"'{beforeIndexer.Value}' isn't name of array");

return result;

}

private List<string> CheckBinaryOperator(Node node)

{

var result = new List<string>();

if (node.Children.Count != 2)

{

result.Add($"Operator '{node.Value}' must take two operands");

}

else

{

if (node.Value == "." && node.Children.Any(c => c.Type != NodeType.Name))

{

result.Add($"Operands of {node.Value} must be names of classes, variables or fields");

}

}

if (node.Children.Any(n => n.Type == NodeType.Empty)) result.Add($"Operands of '{node.Value}' can't be empty");

if (node.Value == ":")

{

if (node.Left.Value != "?") result.Add("Unexpected token ':'");

}

return result;

}

private List<string> CheckAssignationOperator(Node node)

{

var result = CheckBinaryOperator(node);

var child = SkipUnaryToChild(node.Children[0]);

if (child.Type != NodeType.Name && child.Type != NodeType.Indexer && child.Value !=".")

result.Add($"The left-hand side of assignation operator {node.Value} must be a variable or indexer");

return result;

}

private List<string> CheckUnaryOperator(Node node)

{

var result = new List<string>();

if (IncDecOperators.Contains(node.Value))

{

var rightPosition = (!node.Children.Any() &&

node.Parent.Type == NodeType.Name || node.Parent.Type == NodeType.Indexer)

|| (node.Children[0].Type == NodeType.Name ||

node.Children[0].Type == NodeType.Indexer);

if (!rightPosition) result.Add($"The operand of {node.Value} must be a variable or indexer");

}

else

{

if (!node.Children.Any()) result.Add($"Expression or variable name expected after {node.Value}");

}

return result;

}

private List<string> CheckPunctuation(Node node)

{

var result = new List<string>();

switch (node.Value)

{

//TODO: when 1 statement and no semicolon

case ";":

{

if (node.Children.Count > 1 && node.Right.Value != ";")

result.Add("Missing semicolon after statement");

// if (!node.Children.All(n => n.IsStatement()))

// result.Add("Only assignment, call, increment or decrement can be used as a statement");

break;

}

}

return result;

}

private List<string> CheckRoot(Node node)

{

var result = new List<string>();

if (!Cycles.Contains(node.Left.Value) && !Conditions.Contains(node.Left.Value) &&

node.Left.Value != ";") result.Add("Missing semicolon after statement");

return result;

}

private List<string> CheckConditions(Node node)

{

var result = new List<string>();

if (node.Value == "if")

{

var bracketNode = node.Left.Value == "()" ? node.Left : null;

if (bracketNode == null)

{

result.Add("Brackets expected after 'if'");

}

}

if (node.Value == "else")

{

if (node.Left.Value != "if") result.Add("Unexpected token 'else'");

}

return result;

}

#endregion

public List<string> GetErrors(Node node)

{

var result = new List<string>();

if (node.Left != null) result.AddRange(GetErrors(node.Left));

if (node.Right != null) result.AddRange(GetErrors(node.Right));

if (NodeTypesChecks.ContainsKey(node.Type))

{

var errorMessages = NodeTypesChecks[node.Type](node);

if (errorMessages != null) result.AddRange(errorMessages);

}

if (node.Children.Count > 2) result.Add($"Unexpected token '{node.Children[2].Value}'");

return result;

}

private List<string> CheckClosing(Node node)

{

var result = new List<string>();

if (node.Value.Length != 2)

{

if (OpeningBrackets.Contains(node.Value)) result.Add($"Bracket '{node.Value}' wasn't closed");

if (ClosingBrackets.Contains(node.Value)) result.Add($"Bracket '{node.Value}' wasn't open");

}

return result;

}

}

**Контрольні приклади:**

**Приклад 1**

Entered expression: b=2\*a[n]; b=d;

No errors were found

**Приклад 2**

Entered expression: b=2\*a[n]] b=d;

2 SYNTAX ERRORS WERE FOUND

**Приклад 3**

Entered expression: b = 3 > 4 ? 2+3 : 1;

No errors were found