Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ национальный исследовательский  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ**

**Н. Г. ЧЕРНЫШЕВСКОГО»**

Факультет компьютерных наук и информационных технологий

**Теория формальных языков и трансляций.**

Лабораторная работа №3. Включение семантики в анализатор. Создание

внутренней формы представления программы.

Студента 4 курса 441 группы

направления 02.03.03 — Математическое обеспечение и администрирование

информационных систем

факультета КНиИТ

Алексеева Олега Сергеевича

Саратов 2024

**Задание**

Дополнить анализатор, разработанный в рамках лабораторных работ №1 и 2, этапом формирования внутренней формы представления программы.

В качестве внутренней формы представления программы используется

ПОЛИЗ – Польская инверсная модель.

# **Описание основных алгоритмов и структур данных, используемых в программе**

**GetValue(StackElement se):**  
Возвращает числовое значение элемента стека. Если элемент представляет переменную (IsVar == true), то берёт значение из массива variables по индексу se.VarIndex; иначе – возвращает сохранённое значение se.Value.

**Push(StackElement se):**  
Добавляет (помещает) переданный элемент в конец стека.

**Pop():**  
Извлекает и удаляет последний (верхний) элемент из стека. Если стек пуст, генерирует исключение.

**PushElm(PostfixEntry entry):**  
Преобразует элемент внутренней формы (ПОЛИЗа) в элемент стека (StackElement).

Если entry – константа, то парсит строковое значение константы из таблицы Constants и сохраняет его;

Если entry – переменная, то сохраняет индекс переменной и её текущее значение из массива variables;

Если entry – указатель (адрес команды), то сохраняет его как значение.  
После чего помещает полученный элемент в стек с помощью метода Push.

**PopVal():**  
Извлекает элемент из стека с помощью Pop() и возвращает его числовое значение (через GetValue).

**SetVarAndPop(int val):**  
Извлекает элемент из стека и проверяет, что он представляет переменную. Затем устанавливает для этой переменной (в массиве variables) новое значение val. Если извлечённый элемент не является переменной, выбрасывается исключение.

**GetStackState():**

Итеративно проходит по всем элементам стека.

Для каждого элемента получает его числовое значение (с помощью метода GetValue) и добавляет его в строку.

Если стек пуст, возвращает строку "<пусто>".

Результат – строковое представление содержимого стека.

**GetVariablesState():**

Проходит по таблице идентификаторов (список переменных).

Для каждой переменной формирует строку вида "имя=значение", где значение берется из массива variables по индексу переменной.

Возвращает строку, объединяющую состояния всех переменных, что позволяет увидеть текущее состояние переменных.

Метод **InterpretWithLogging** выполняет интерпретацию программы, представленной в виде ПОЛИЗа, с подробным логированием каждого шага. Его основное назначение:

**Инициализация:**

Принимает список элементов ПОЛИЗа, таблицы идентификаторов и констант.

Инициализирует указатель текущей инструкции, стек, массив значений переменных (из таблицы идентификаторов) и объект для накопления вывода.

**Основной цикл интерпретации:**

Последовательно проходит по инструкциям ПОЛИЗа.

Для каждого шага записывает в лог:

Номер текущей инструкции и её содержимое.

Состояние стека до выполнения команды.

Если текущий элемент – команда, то в зависимости от её типа выполняются соответствующие операции (арифметические, логические, условные переходы, присваивания, вывод и т.д.).

После выполнения команды обновляется указатель, и записывается состояние стека и таблицы переменных после операции.

**Формирование итогового вывода:**

После завершения интерпретации к итоговому результату добавляется подробный лог, где отражены все шаги выполнения, изменения в стеке и переменных.

**Контрольный пример**

do until a < 10 a = a + 1; output a loop

=== Лог интерпретации ===

=== Начало интерпретации ===

Инструкция 0: etVar: a

Состояние стека до выполнения: <пусто>

Помещён операнд: etVar: a

Состояние стека после выполнения: 0

Текущее состояние переменных: a=0

----------------------------------------

Инструкция 1: etConst: 10

Состояние стека до выполнения: 0

Помещён операнд: etConst: 10

Состояние стека после выполнения: 0 10

Текущее состояние переменных: a=0

----------------------------------------

Инструкция 2: etCmd: CMPL

Состояние стека до выполнения: 0 10

CMPL: 0 < 10 ? 1

Состояние стека после выполнения: 1

Текущее состояние переменных: a=0

----------------------------------------

Инструкция 3: etCmd: NOT

Состояние стека до выполнения: 1

NOT: !1 = 0

Состояние стека после выполнения: 0

Текущее состояние переменных: a=0

----------------------------------------

Инструкция 4: etCmdPtr: 15

Состояние стека до выполнения: 0

Помещён операнд: etCmdPtr: 15

Состояние стека после выполнения: 0 15

Текущее состояние переменных: a=0

----------------------------------------

Инструкция 5: etCmd: JZ

Состояние стека до выполнения: 0 15

Выполняется команда JZ: условие = 0, адрес = 15

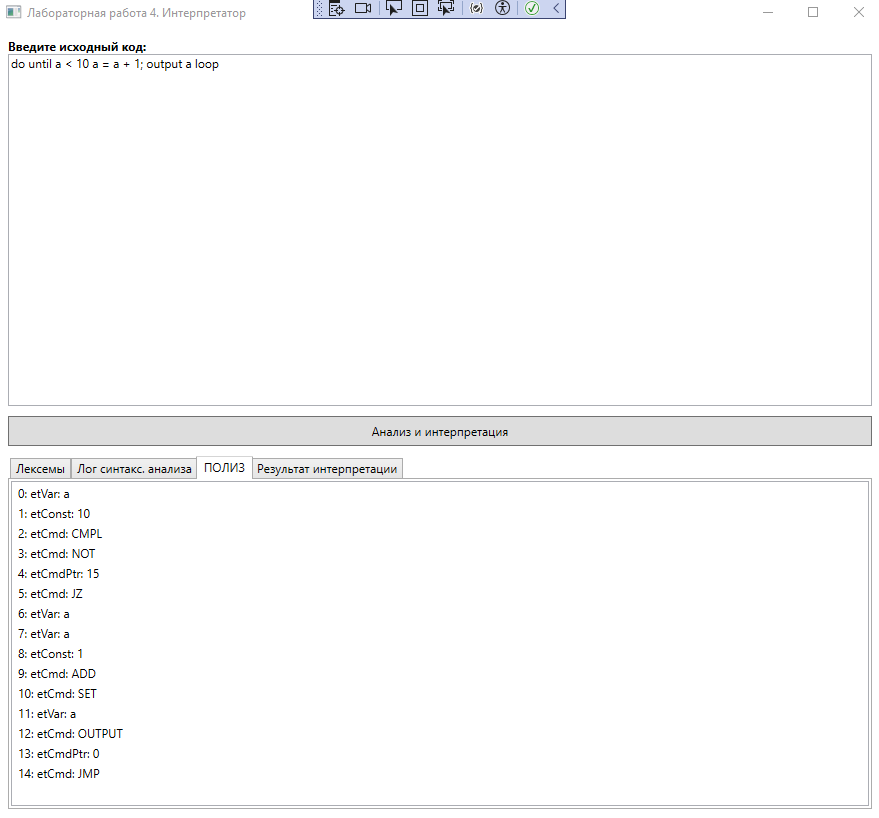
Условие ложно, переход по адресу.

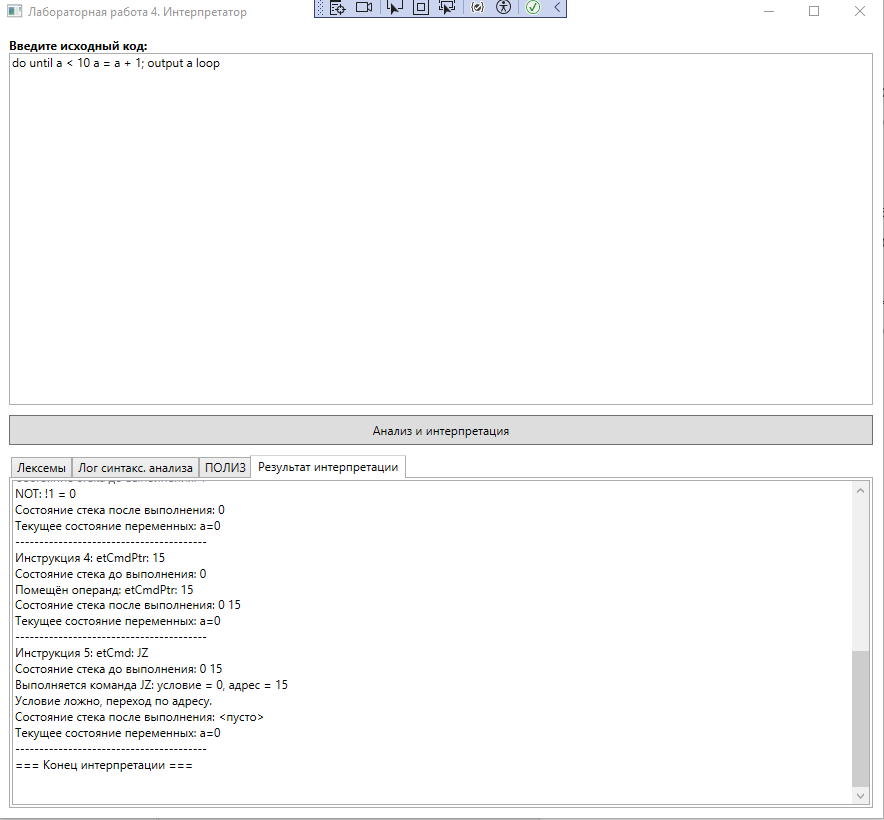
Состояние стека после выполнения: <пусто>

Текущее состояние переменных: a=0

----------------------------------------

=== Конец интерпретации ===





# **Листинг программы**

using System;

using System.Collections.Generic;

using System.Text;

using Lab4.Models;

namespace Lab4.Services

{

public static class Interpreter

{

private static List<PostfixEntry> Postfix;

private static List<string> Identifiers;

private static List<string> Constants;

private static int pos;

private static List<StackElement> stack;

private static int[] variables;

private static StringBuilder output;

private struct StackElement

{

public bool IsVar;

public int VarIndex;

public int Value;

}

private static int GetValue(StackElement se)

{

return se.IsVar ? variables[se.VarIndex] : se.Value;

}

private static void Push(StackElement se)

{

stack.Add(se);

}

private static StackElement Pop()

{

if (stack.Count == 0)

throw new Exception("Ошибка: переполнение стека");

StackElement se = stack[stack.Count - 1];

stack.RemoveAt(stack.Count - 1);

return se;

}

private static void PushElm(PostfixEntry entry)

{

StackElement se = new StackElement();

switch (entry.Type)

{

case EEntryType.etConst:

se.IsVar = false;

se.Value = int.Parse(Constants[entry.Index]);

break;

case EEntryType.etVar:

se.IsVar = true;

se.VarIndex = entry.Index;

se.Value = variables[entry.Index];

break;

case EEntryType.etCmdPtr:

se.IsVar = false;

se.Value = entry.Index;

break;

}

Push(se);

}

private static int PopVal()

{

return GetValue(Pop());

}

private static void SetVarAndPop(int val)

{

StackElement se = Pop();

if (!se.IsVar)

throw new Exception("SET: Ожидалась ссылка на переменную.");

variables[se.VarIndex] = val;

}

public static string InterpretWithLogging(List<PostfixEntry> postfix, List<string> idTable, List<string> constTable)

{

Postfix = postfix;

Identifiers = idTable;

Constants = constTable;

pos = 0;

stack = new List<StackElement>();

variables = new int[Identifiers.Count];

output = new StringBuilder();

StringBuilder log = new StringBuilder();

log.AppendLine("=== Начало интерпретации ===");

while (pos < Postfix.Count)

{

log.AppendLine($"Инструкция {pos}: {Postfix[pos]}");

log.AppendLine($"Состояние стека до выполнения: {GetStackState()}");

int prevPos = pos;

if (Postfix[pos].Type == EEntryType.etCmd)

{

ECmd cmd = (ECmd)Postfix[pos].Index;

switch (cmd)

{

case ECmd.JMP:

{

int jumpAddr = PopVal();

log.AppendLine($"Выполняется команда JMP, переход на адрес {jumpAddr}");

pos = jumpAddr;

}

break;

case ECmd.JZ:

{

int addr = PopVal();

int cond = PopVal();

log.AppendLine($"Выполняется команда JZ: условие = {cond}, адрес = {addr}");

if (cond != 0)

{

pos++;

log.AppendLine("Условие истинно, переходим к следующей инструкции.");

}

else

{

pos = addr;

log.AppendLine("Условие ложно, переход по адресу.");

}

}

break;

case ECmd.SET:

{

int value = PopVal();

SetVarAndPop(value);

log.AppendLine($"Выполняется SET, присваиваем значение {value}");

pos++;

}

break;

case ECmd.ADD:

{

int b = PopVal();

int a = PopVal();

int res = a + b;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"ADD: {a} + {b} = {res}");

pos++;

}

break;

case ECmd.SUB:

{

int b = PopVal();

int a = PopVal();

int res = a - b;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"SUB: {a} - {b} = {res}");

pos++;

}

break;

case ECmd.MUL:

{

int b = PopVal();

int a = PopVal();

int res = a \* b;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"MUL: {a} \* {b} = {res}");

pos++;

}

break;

case ECmd.DIV:

{

int b = PopVal();

int a = PopVal();

if (b == 0)

throw new Exception("Деление на ноль!");

int res = a / b;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"DIV: {a} / {b} = {res}");

pos++;

}

break;

case ECmd.AND:

{

int b = PopVal();

int a = PopVal();

int res = (a != 0 && b != 0) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"AND: {a} && {b} = {res}");

pos++;

}

break;

case ECmd.OR:

{

int b = PopVal();

int a = PopVal();

int res = (a != 0 || b != 0) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"OR: {a} || {b} = {res}");

pos++;

}

break;

case ECmd.NOT:

{

int a = PopVal();

int res = (a == 0) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"NOT: !{a} = {res}");

pos++;

}

break;

case ECmd.CMPE:

{

int b = PopVal();

int a = PopVal();

int res = (a == b) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"CMPE: {a} == {b} ? {res}");

pos++;

}

break;

case ECmd.CMPNE:

{

int b = PopVal();

int a = PopVal();

int res = (a != b) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"CMPNE: {a} <> {b} ? {res}");

pos++;

}

break;

case ECmd.CMPL:

{

int b = PopVal();

int a = PopVal();

int res = (a < b) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"CMPL: {a} < {b} ? {res}");

pos++;

}

break;

case ECmd.CMPLE:

{

int b = PopVal();

int a = PopVal();

int res = (a <= b) ? 1 : 0;

Push(new StackElement { IsVar = false, Value = res });

log.AppendLine($"CMPLE: {a} <= {b} ? {res}");

pos++;

}

break;

case ECmd.OUTPUT:

{

int val = PopVal();

output.AppendLine(val.ToString());

log.AppendLine($"OUTPUT: выведено значение {val}");

pos++;

}

break;

default:

throw new Exception("Неизвестная команда: " + cmd.ToString());

}

}

else

{

PushElm(Postfix[pos]);

log.AppendLine($"Помещён операнд: {Postfix[pos]}");

pos++;

}

log.AppendLine($"Состояние стека после выполнения: {GetStackState()}");

log.AppendLine($"Текущее состояние переменных: {GetVariablesState()}");

log.AppendLine(new string('-', 40));

}

log.AppendLine("=== Конец интерпретации ===");

output.AppendLine("\n=== Лог интерпретации ===");

output.AppendLine(log.ToString());

return output.ToString();

}

private static string GetStackState()

{

if (stack.Count == 0)

return "<пусто>";

StringBuilder sb = new StringBuilder();

foreach (var se in stack)

{

sb.Append(GetValue(se) + " ");

}

return sb.ToString();

}

private static string GetVariablesState()

{

StringBuilder sb = new StringBuilder();

for (int i = 0; i < Identifiers.Count; i++)

{

sb.Append($"{Identifiers[i]}={variables[i]} ");

}

return sb.ToString();

}

}

}

using System;

using System.Collections.Generic;

using Lab4.Models;

namespace Lab4.Services

{

public class LexAnalyzer

{

public List<Token> Tokens { get; private set; } = new List<Token>();

public List<string> Identifiers { get; private set; } = new List<string>();

public List<string> Constants { get; private set; } = new List<string>();

private Dictionary<string, TokenType> keywords = new Dictionary<string, TokenType>()

{

{ "do", TokenType.Do },

{ "until", TokenType.Until },

{ "loop", TokenType.Loop },

{ "output", TokenType.Output },

{ "not", TokenType.Not },

{ "and", TokenType.And },

{ "or", TokenType.Or }

};

public bool Analyze(string input)

{

int pos = 0;

while (pos < input.Length)

{

char c = input[pos];

if (char.IsWhiteSpace(c))

{

pos++;

continue;

}

if (char.IsLetter(c))

{

int start = pos;

while (pos < input.Length && char.IsLetterOrDigit(input[pos]))

{

pos++;

}

string lexeme = input.Substring(start, pos - start);

string lowerLexeme = lexeme.ToLower();

if (keywords.ContainsKey(lowerLexeme))

{

Token token = new Token { Type = keywords[lowerLexeme], Lexeme = lexeme, StartPos = start, EndPos = pos - 1 };

Tokens.Add(token);

}

else

{

Token token = new Token { Type = TokenType.Identifier, Lexeme = lexeme, StartPos = start, EndPos = pos - 1 };

Tokens.Add(token);

if (!Identifiers.Contains(lexeme))

Identifiers.Add(lexeme);

}

continue;

}

else if (char.IsDigit(c))

{

int start = pos;

while (pos < input.Length && char.IsDigit(input[pos]))

{

pos++;

}

string lexeme = input.Substring(start, pos - start);

Token token = new Token { Type = TokenType.Constant, Lexeme = lexeme, StartPos = start, EndPos = pos - 1 };

Tokens.Add(token);

if (!Constants.Contains(lexeme))

Constants.Add(lexeme);

continue;

}

else

{

if (c == '<')

{

if (pos + 1 < input.Length && input[pos + 1] == '>')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "<>", StartPos = pos, EndPos = pos + 1 };

Tokens.Add(token);

pos += 2;

continue;

}

else

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "<", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

}

else if (c == '>')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = ">", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '=')

{

if (pos + 1 < input.Length && input[pos + 1] == '=')

{

Token token = new Token { Type = TokenType.Rel, Lexeme = "==", StartPos = pos, EndPos = pos + 1 };

Tokens.Add(token);

pos += 2;

continue;

}

else

{

Token token = new Token { Type = TokenType.As, Lexeme = "=", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

}

else if (c == '+')

{

Token token = new Token { Type = TokenType.Plus, Lexeme = "+", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '-')

{

Token token = new Token { Type = TokenType.Minus, Lexeme = "-", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '\*')

{

Token token = new Token { Type = TokenType.Multiply, Lexeme = "\*", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == '/')

{

Token token = new Token { Type = TokenType.Divide, Lexeme = "/", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else if (c == ';')

{

Token token = new Token { Type = TokenType.Delimiter, Lexeme = ";", StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

else

{

Token token = new Token { Type = TokenType.Unknown, Lexeme = c.ToString(), StartPos = pos, EndPos = pos };

Tokens.Add(token);

pos++;

continue;

}

}

}

return true;

}

}

}

using System;

using System.Collections.Generic;

using System.Text;

using Lab4.Models;

namespace Lab4.Services

{

public static class Parser

{

private static List<Token> tokens;

private static int currentIndex;

private static StringBuilder parseLog = new StringBuilder();

private static int indentLevel = 0;

public static List<PostfixEntry> Postfix { get; private set; } = new List<PostfixEntry>();

public static List<string> Identifiers { get; set; }

public static List<string> Constants { get; set; }

private static Token CurrentToken => currentIndex < tokens.Count ? tokens[currentIndex] : null;

public static string GetParseLog() => parseLog.ToString();

private static void Log(string message)

{

parseLog.AppendLine(new string(' ', indentLevel \* 2) + message);

}

private static void Error(string msg, int pos)

{

throw new Exception($"Ошибка на позиции {pos}: {msg}");

}

private static int WriteCmd(ECmd cmd)

{

Postfix.Add(new PostfixEntry { Type = EEntryType.etCmd, Index = (int)cmd });

return Postfix.Count - 1;

}

private static int WriteVar(int varIndex)

{

Postfix.Add(new PostfixEntry { Type = EEntryType.etVar, Index = varIndex });

return Postfix.Count - 1;

}

private static int WriteConst(int constIndex)

{

Postfix.Add(new PostfixEntry { Type = EEntryType.etConst, Index = constIndex });

return Postfix.Count - 1;

}

private static int WriteCmdPtr(int ptr)

{

Postfix.Add(new PostfixEntry { Type = EEntryType.etCmdPtr, Index = ptr });

return Postfix.Count - 1;

}

private static void SetCmdPtr(int ind, int ptr)

{

Postfix[ind] = new PostfixEntry { Type = EEntryType.etCmdPtr, Index = ptr };

}

private static void SwapLastTwoEntries() // для обработки оператора >

{

int count = Postfix.Count;

if (count < 2) return;

var temp = Postfix[count - 1];

Postfix[count - 1] = Postfix[count - 2];

Postfix[count - 2] = temp;

}

public static bool DoUntilStatement()

{

Log("Вход в DoUntilStatement, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null || CurrentToken.Type != TokenType.Do)

{

Error("Ожидалось 'do'", CurrentToken != null ? CurrentToken.StartPos : -1);

return false;

}

currentIndex++;

if (CurrentToken == null || CurrentToken.Type != TokenType.Until)

{

Error("Ожидалось 'until'", CurrentToken != null ? CurrentToken.StartPos : -1);

return false;

}

currentIndex++;

int loopStart = Postfix.Count;

if (!LogicalExpression())

return false;

WriteCmd(ECmd.NOT);

int exitJumpIndex = WriteCmdPtr(-1);

WriteCmd(ECmd.JZ);

if (!Operators())

return false;

WriteCmdPtr(loopStart);

int jmpIndex = WriteCmd(ECmd.JMP);

SetCmdPtr(exitJumpIndex, jmpIndex + 1);

if (CurrentToken == null || CurrentToken.Type != TokenType.Loop)

{

Error("Ожидалось 'loop'", CurrentToken != null ? CurrentToken.StartPos : -1);

return false;

}

currentIndex++;

indentLevel--;

Log("Выход из DoUntilStatement, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

if (currentIndex < tokens.Count)

{

Error("Лишние токены после 'loop'", CurrentToken != null ? CurrentToken.StartPos : -1);

return false;

}

return true;

}

public static bool LogicalExpression()

{

Log("Вход в LogicalExpression, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

bool hasNot = false;

if (CurrentToken != null && CurrentToken.Type == TokenType.Not)

{

hasNot = true;

Log("Найден 'not'");

currentIndex++;

}

if (!ComparisonExpression())

return false;

if (hasNot)

{

WriteCmd(ECmd.NOT);

}

while (CurrentToken != null &&

(CurrentToken.Type == TokenType.And || CurrentToken.Type == TokenType.Or))

{

Token op = CurrentToken;

Log("Найден логический оператор: " + op.ToString());

currentIndex++;

if (!ComparisonExpression())

return false;

if (op.Type == TokenType.And)

WriteCmd(ECmd.AND);

else if (op.Type == TokenType.Or)

WriteCmd(ECmd.OR);

}

indentLevel--;

Log("Выход из LogicalExpression, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool ComparisonExpression()

{

Log("Вход в ComparisonExpression, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!Operand())

return false;

if (CurrentToken != null && CurrentToken.Type == TokenType.Rel)

{

string op = CurrentToken.Lexeme;

Log("Найден оператор сравнения: " + op);

currentIndex++;

if (!Operand())

return false;

ECmd cmd;

if (op == "<")

cmd = ECmd.CMPL;

else if (op == ">")

{

SwapLastTwoEntries();

cmd = ECmd.CMPL;

}

else if (op == "==")

cmd = ECmd.CMPE;

else if (op == "<>")

cmd = ECmd.CMPNE;

else

cmd = ECmd.CMPL;

WriteCmd(cmd);

}

indentLevel--;

Log("Выход из ComparisonExpression, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool OperatorStmt()

{

Log("Вход в OperatorStmt, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null)

{

Error("Ожидался оператор", -1);

return false;

}

if (CurrentToken.Type == TokenType.Identifier)

{

string varName = CurrentToken.Lexeme;

int varIndex = Identifiers.IndexOf(varName);

Log("Обнаружено присваивание, идентификатор: " + CurrentToken.ToString());

WriteVar(varIndex);

currentIndex++;

if (CurrentToken == null || CurrentToken.Type != TokenType.As)

{

Error("Ожидалось '=' в присваивании", CurrentToken != null ? CurrentToken.StartPos : -1);

return false;

}

Log("Найден '='");

currentIndex++;

if (!ArithExpr())

return false;

WriteCmd(ECmd.SET);

indentLevel--;

Log("Выход из OperatorStmt (присваивание), токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

else if (CurrentToken.Type == TokenType.Output)

{

Log("Обнаружен оператор вывода: " + CurrentToken.ToString());

currentIndex++;

if (!Operand())

return false;

WriteCmd(ECmd.OUTPUT);

indentLevel--;

Log("Выход из OperatorStmt (вывод), токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

else

{

Error("Ожидалось присваивание или оператор вывода", CurrentToken.StartPos);

return false;

}

}

public static bool ArithExpr()

{

Log("Вход в ArithExpr, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!Operand())

return false;

while (CurrentToken != null &&

(CurrentToken.Type == TokenType.Plus ||

CurrentToken.Type == TokenType.Minus ||

CurrentToken.Type == TokenType.Multiply ||

CurrentToken.Type == TokenType.Divide))

{

Token op = CurrentToken;

Log("Найден арифметический оператор: " + op.ToString());

currentIndex++;

if (!Operand())

return false;

ECmd cmd;

if (op.Type == TokenType.Plus)

cmd = ECmd.ADD;

else if (op.Type == TokenType.Minus)

cmd = ECmd.SUB;

else if (op.Type == TokenType.Multiply)

cmd = ECmd.MUL;

else

cmd = ECmd.DIV;

WriteCmd(cmd);

}

indentLevel--;

Log("Выход из ArithExpr, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool Operand()

{

Log("Вход в Operand, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (CurrentToken == null)

{

Error("Ожидался операнд (идентификатор или константа)", -1);

return false;

}

if (CurrentToken.Type == TokenType.Identifier)

{

string varName = CurrentToken.Lexeme;

int varIndex = Identifiers.IndexOf(varName);

Log("Найден идентификатор: " + CurrentToken.ToString());

WriteVar(varIndex);

currentIndex++;

}

else if (CurrentToken.Type == TokenType.Constant)

{

string constVal = CurrentToken.Lexeme;

int constIndex = Constants.IndexOf(constVal);

Log("Найдена константа: " + CurrentToken.ToString());

WriteConst(constIndex);

currentIndex++;

}

else

{

Error("Ожидался идентификатор или константа", CurrentToken.StartPos);

return false;

}

indentLevel--;

Log("Выход из Operand, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool Operators()

{

Log("Вход в Operators, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

indentLevel++;

if (!OperatorStmt())

return false;

while (CurrentToken != null && CurrentToken.Type == TokenType.Delimiter)

{

Log("Найден символ ';'");

currentIndex++;

if (!OperatorStmt())

return false;

}

indentLevel--;

Log("Выход из Operators, токен: " + (CurrentToken != null ? CurrentToken.ToString() : "null"));

return true;

}

public static bool Parse(List<Token> tokenList, List<string> idTable, List<string> constTable)

{

tokens = tokenList;

Identifiers = idTable;

Constants = constTable;

currentIndex = 0;

parseLog.Clear();

indentLevel = 0;

Postfix.Clear();

Log("=== Начало синтаксического анализа ===");

bool result = DoUntilStatement();

Log("=== Конец синтаксического анализа ===");

return result;

}

}

}

namespace Lab4.Models

{

public enum EEntryType

{

etCmd,

etVar,

etConst,

etCmdPtr

}

public enum ECmd

{

JMP,

JZ,

SET,

ADD,

SUB,

MUL,

DIV,

AND,

OR,

NOT,

CMPE,

CMPNE,

CMPL,

CMPLE,

OUTPUT

}

public struct PostfixEntry

{

public EEntryType Type;

public int Index;

public static List<string> ConstTable { get; set; }

public static List<string> IdentifierTable { get; set; }

public override string ToString()

{

switch (Type)

{

case EEntryType.etCmd:

return $"etCmd: {((ECmd)Index)}";

case EEntryType.etVar:

if (IdentifierTable != null && Index >= 0 && Index < IdentifierTable.Count)

return $"etVar: {IdentifierTable[Index]}";

else

return $"etVar: {Index}";

case EEntryType.etConst:

if (ConstTable != null && Index >= 0 && Index < ConstTable.Count)

return $"etConst: {ConstTable[Index]}";

else

return $"etConst: {Index}";

case EEntryType.etCmdPtr:

return $"etCmdPtr: {Index}";

default:

return $"{Type} : {Index}";

}

}

}

}

namespace Lab4.Models

{

public class Token

{

public TokenType Type { get; set; }

public string Lexeme { get; set; }

public int StartPos { get; set; }

public int EndPos { get; set; }

public override string ToString() => $"{Lexeme} ({Type})";

}

}

namespace Lab4.Models

{

public enum TokenType

{

Do,

Until,

Loop,

Output,

Not,

And,

Or,

Identifier,

Constant,

Rel,

As,

Plus,

Minus,

Multiply,

Divide,

Delimiter,

Unknown

}

}

<Window x:Class="Lab4.Views.MainWindow"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Title="Лабораторная работа 4. Интерпретатор" Height="700" Width="900">

<Grid Margin="10">

<Grid.RowDefinitions>

<RowDefinition Height="Auto"/>

<RowDefinition Height="3\*"/>

<RowDefinition Height="Auto"/>

<RowDefinition Height="3\*"/>

</Grid.RowDefinitions>

<TextBlock Text="Введите исходный код:" FontWeight="Bold"/>

<TextBox x:Name="InputTextBox" Grid.Row="1" AcceptsReturn="True" VerticalScrollBarVisibility="Auto" TextWrapping="Wrap" />

<Button x:Name="InterpretButton" Grid.Row="2" Content="Анализ и интерпретация" Margin="0,10,0,10" Height="30" Click="InterpretButton\_Click"/>

<TabControl Grid.Row="3">

<TabItem Header="Лексемы">

<ListBox x:Name="TokensListBox"/>

</TabItem>

<TabItem Header="Лог синтакс. анализа">

<TextBox x:Name="ParseLogTextBox" IsReadOnly="True" TextWrapping="Wrap" VerticalScrollBarVisibility="Auto"/>

</TabItem>

<TabItem Header="ПОЛИЗ">

<ListBox x:Name="PostfixListBox"/>

</TabItem>

<TabItem Header="Результат интерпретации">

<TextBox x:Name="InterpretOutputTextBox" IsReadOnly="True" TextWrapping="Wrap" VerticalScrollBarVisibility="Auto"/>

</TabItem>

</TabControl>

</Grid>

</Window>

using System;

using System.Linq;

using System.Windows;

using Lab4.Models;

using Lab4.Services;

namespace Lab4.Views

{

public partial class MainWindow : Window

{

public MainWindow()

{

InitializeComponent();

InputTextBox.Text = "do until a < 10 a = a + 1; output a loop";

}

private void InterpretButton\_Click(object sender, RoutedEventArgs e)

{

try

{

TokensListBox.Items.Clear();

ParseLogTextBox.Clear();

PostfixListBox.Items.Clear();

InterpretOutputTextBox.Clear();

string input = InputTextBox.Text;

LexAnalyzer lex = new LexAnalyzer();

bool lexOk = lex.Analyze(input);

PostfixEntry.ConstTable = lex.Constants;

PostfixEntry.IdentifierTable = lex.Identifiers;

if (!lexOk)

{

MessageBox.Show("Лексический анализ завершился с ошибками.");

return;

}

foreach (var token in lex.Tokens)

{

TokensListBox.Items.Add(token.ToString());

}

bool parseOk = Parser.Parse(lex.Tokens, lex.Identifiers, lex.Constants);

ParseLogTextBox.Text = Parser.GetParseLog();

if (!parseOk)

{

MessageBox.Show("Синтаксический анализ завершился с ошибками.");

return;

}

int i = 0;

foreach (var entry in Parser.Postfix)

{

PostfixListBox.Items.Add($"{i++}: {entry}");

}

string interpResult = Interpreter.InterpretWithLogging(Parser.Postfix, lex.Identifiers, lex.Constants);

InterpretOutputTextBox.Text = interpResult;

MessageBox.Show("Анализ и интерпретация завершены успешно.");

}

catch (Exception ex)

{

MessageBox.Show("Ошибка: " + ex.Message);

}

}

}

}