Минобрнауки России

Федеральное государственное бюджетное образовательное учреждение

высшего образования

**«САРАТОВСКИЙ национальный исследовательский  
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ**

**Н. Г. ЧЕРНЫШЕВСКОГО»**

Факультет компьютерных наук и информационных технологий

**Основы трансляции языков программирования.**

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

Новоярчикова Михаила Андреевича

Саратов 2024

# Задание на лабораторную работу

Реализовать интерпретатор польской инверсной записи. Дополнить анализатор, разработанный в рамках лабораторных работ №1-3 реализованным интерпретатором.

# Описание внутренней формы представления программы

Внутренняя форма программы в нашем интерпретаторе представлена в виде обратной польской записи (ПОЛИЗ). Этот подход позволяет значительно упростить процесс вычислений, исключив необходимость дополнительных скобок и соблюдения приоритетов операторов при интерпретации выражений.

Основные структуры и этапы формирования и обработки внутренней формы программы включают следующие элементы:

**1. Лексический анализатор (Lexer):**

Преобразует входной текст программы в список токенов. Каждый токен представляет собой идентификатор, ключевое слово, оператор или константу.

Лексер проходит по каждому символу входного выражения, определяет тип символа (цифра, буква, оператор и т.д.) и формирует соответствующий токен.

**Пример:**

Входное выражение x = x + 10 \* 2 будет преобразовано в токены:

* ID (x)
* ASSIGN (=)
* ID (x)
* PLUS (+)
* CONST (10)
* MULT (\*)
* CONST (2)

**2. Синтаксический анализатор (Parser):**

Преобразует список токенов в ПОЛИЗ, формируя команды для выполнения.

В процессе анализа строятся выражения, соответствующие синтаксическим правилам, и генерируются инструкции для дальнейшей обработки.

Используются следующие методы:

* ParseAssignment — обрабатывает присваивание переменной значений, основанных на выражениях.
* ParseExpression — анализирует арифметические выражения, такие как сложение и вычитание.
* ParseLogicalExpression — анализирует логические выражения, такие как and, or.
* ParseRelationalExpression — обрабатывает операторы сравнения (например, >, <, ==).

**Пример:**

Выражение: x = x + 10 \* 2 будет преобразовано в ПОЛИЗ:

* Переменная (x)
* Константа (10)
* Константа (2)
* Команда (MUL)
* Команда (ADD)
* Команда (SET)

**3. ПОЛИЗ (PostfixForm):**

Представляет программу в виде списка команд (инструкций) и операндов, которые будут интерпретированы в дальнейшем.

Содержит стек для временного хранения значений и словарь для хранения переменных.

**Пример команд ПОЛИЗ:**

* Переменная (хэш): -842352680
* Константа: 10
* Константа: 2
* Команда: MUL
* Команда: ADD
* Команда: SET

**4. Интерпретация ПОЛИЗ (Interpret):**

Этот этап выполняет последовательное выполнение команд, указанных в ПОЛИЗ.

Каждая команда обрабатывается с помощью метода ExecuteCommand, который выполняет конкретные операции (например, сложение, умножение, логические операции, присваивание и т.д.).

Для выполнения переходов используется указатель на команду (etCmdPtr), позволяющий осуществлять циклы и условия в программе.

Во время выполнения интерпретации поддерживается стек для вычислений и словарь для хранения значений переменных.

**Пример выполнения ПОЛИЗ:**

* Присваивание значения переменной.
* Выполнение арифметической операции умножения и сложения.
* Логическое сравнение и выполнение циклов или условий.

**5. Хранение переменных:**

Переменные хранятся в словаре (\_variables), где ключами являются хэш-коды имён переменных, а значениями — числовые значения этих переменных.

Хэш-коды переменных используются для обеспечения быстрой индексации.

**Пример:**

Переменная с хэшом -842352680 = 100

Таким образом, вся программа сначала преобразуется в список токенов, затем в ПОЛИЗ, который интерпретируется поэтапно, производя вычисления и сохраняя результаты в переменных.

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

**а) Описание алгоритма интерпретации**

Алгоритм интерпретации в данной программе заключается в пошаговом выполнении команд, представленных в виде ПОЛИЗ (обратной польской записи). Интерпретация включает в себя следующие этапы:

1. Лексический анализ: Входное выражение разбивается на токены с помощью лексера. Токены представляют собой отдельные логические единицы программы: идентификаторы, операторы, ключевые слова и константы.
2. Синтаксический анализ: С помощью синтаксического анализатора (парсера) на основе полученных токенов строится ПОЛИЗ. Парсер преобразует выражение в набор команд, которые затем выполняются интерпретатором.
3. Построение ПОЛИЗ: Сформированный ПОЛИЗ представляет собой последовательность операций и операндов, которые будут обработаны в порядке, описанном в ПОЛИЗ.
4. Интерпретация ПОЛИЗ: Интерпретатор выполняет команды в ПОЛИЗ, используя стек для промежуточных вычислений и словарь для хранения значений переменных.
   1. Выполнение арифметических, логических операций.
   2. Присваивание значений переменным.
   3. Переходы по командам (реализация циклов и условий).
   4. Вычисления и промежуточные результаты хранятся в стеке.
5. Окончание интерпретации: Интерпретация завершается, когда все команды в ПОЛИЗ выполнены. После этого выводятся значения всех переменных.

**б) Описание структур данных, используемых при интерпретации**

В программе используются следующие ключевые структуры данных:

**1)** **Массив команд ПОЛИЗ (\_postfix):** Представляет собой список инструкций, которые интерпретатор выполняет. В этот массив записываются команды, операнды (константы) и переменные.

**Пример содержимого:**

* Переменная (хэш): -842352680
* Константа: 10
* Константа: 2
* Команда: MUL
* Команда: ADD

**2)** **Стек (\_stack):** Используется для хранения промежуточных вычислений при выполнении арифметических и логических операций. Например, при выполнении команды сложения или умножения значения операндов помещаются в стек, где выполняются вычисления.

**Пример:**

После выполнения команды MUL стек содержит результат произведения двух чисел.

**3) Словарь переменных (\_variables):** Хранит значения всех переменных, участвующих в программе. В качестве ключа используется хэш-код имени переменной, а значением является текущая величина этой переменной.

**Пример:**

\_variables["-842352680"] = 100 // Значение переменной x

**4) Словарь имён переменных (\_varNames):** Сопоставляет хэш-код переменной с её именем для удобного вывода значений.

**Пример:**

\_varNames[-842352680] = "x"

**в)** **Описание процедур и функций, отвечающих за выполнение команд интерпретатора и работу со структурами данных**

**1) Функция Interpret():** Основная функция интерпретации, которая последовательно обрабатывает команды ПОЛИЗ. Алгоритм заключается в цикле, где на каждой итерации интерпретатор получает текущую команду и выполняет соответствующую операцию.

* Чтение команды: Интерпретатор считывает текущий элемент ПОЛИЗ.
* Выполнение команды: В зависимости от типа команды (etCmd, etVar, etConst, etCmdPtr), вызывается соответствующая функция для выполнения операции (арифметической, логической, перехода и т.д.).

**2) Функция ExecuteCommand(ECmd cmd, int pos):** Эта функция выполняет конкретные команды, которые встречаются в ПОЛИЗ:

* SET — Присваивает переменной значение.
* ADD, SUB, MUL, DIV — Выполняют соответствующие арифметические операции.
* CMP — Логические команды сравнения (равно, больше, меньше и т.д.).
* AND, OR — Логические операции "и", "или".
* JMP, JZ — Операции переходов для циклов и условий.

**3) Функции работы со стеком:**

* PushVal(int value) — добавляет значение в стек.
* PopVal() — извлекает значение из стека для последующей обработки.

**Пример:**

int value1 = \_stack.PopVal();

int value2 = \_stack.PopVal();

\_stack.PushVal(value1 + value2); // Сложение двух значений и возврат в стек

**4) Функции работы с переменными:**

* SetVarValue(int varHash, int value) — записывает значение переменной в словарь \_variables.
* GetVarValue(int varHash) — получает значение переменной по её хэш-коду из словаря.

**5) Функция PrintPostfix():**

Выводит сгенерированный ПОЛИЗ для наглядного отображения результата синтаксического анализа и построения внутренней формы программы.

**6) Функция PrintVariables():**

После завершения интерпретации выводит значения всех переменных, хранящихся в словаре \_variables, с сопоставлением их имён из \_varNames.

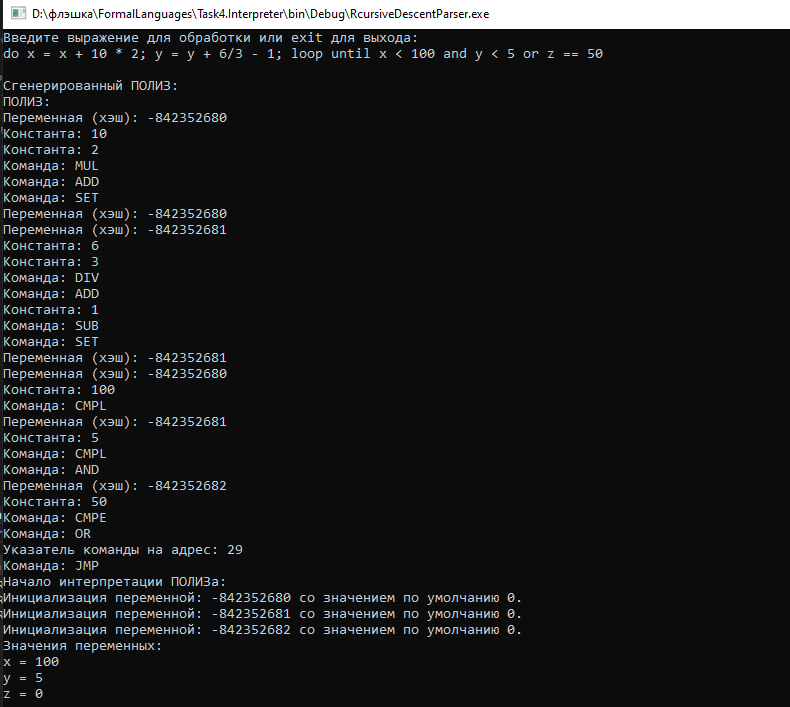
Таким образом, основная логика программы построена на использовании стека и словаря для хранения переменных, а также системы команд для обработки арифметических операций, условий и циклов, представленных в ПОЛИЗ.

# Описание интерфейса пользователя.

Как и в предыдущих лабораторных, вводится выражение для анализа либо exit для выхода.

# Контрольный пример и результаты тестирования.

do x = x + 10 \* 2; y = y + 6/3 - 1; loop until x < 100 and y < 5 or z == 50



# Листинг программы

using Interpreter.Lexer;

using System;

using System.Collections.Generic;

namespace Interpreter

{

class Program

{

static void Main(string[] args)

{

while (true)

{

Console.WriteLine("Введите выражение для обработки или exit для выхода:");

string input = Console.ReadLine();

Console.WriteLine();

if (input == "exit")

{

break;

}

try

{

var lexer = new Lexer.Lexer(input);

List<Token> tokens = lexer.Tokenize();

var parser = new Parser.Parser(tokens);

parser.ParseDoLoopUntil();

Console.WriteLine("Сгенерированный ПОЛИЗ:");

parser.PrintPostfix();

Console.WriteLine("Начало интерпретации ПОЛИЗа:");

var postfix = parser.GetPostfixForm();

postfix.Interpret();

postfix.PrintVariables();

}

catch (Exception ex)

{

Console.WriteLine($"Ошибка: {ex.Message}");

}

Console.WriteLine();

}

}

}

}

using Interpreter.Grammar;

using Interpreter.Postfix;

using Interpreter.Lexer;

using System;

using System.Collections.Generic;

namespace Interpreter.Parser

{

public class Parser

{

private readonly List<Token> \_tokens;

private PostfixForm \_postfix;

private int \_currentTokenIndex;

#region Constructor and Helpers

public Parser(List<Token> tokens)

{

\_tokens = tokens;

\_currentTokenIndex = 0;

\_postfix = new PostfixForm();

}

private Token CurrentToken

{

get

{

if (HasMoreTokens())

{

return \_tokens[\_currentTokenIndex];

}

else

{

throw new Exception("Ошибка: неожиданный конец выражения.");

}

}

}

private bool HasMoreTokens()

{

return \_currentTokenIndex < \_tokens.Count;

}

private void Match(TokenType expectedType)

{

if (CurrentToken.Type == expectedType)

{

\_currentTokenIndex++;

}

else

{

throw new Exception($"Ошибка: ожидалось {expectedType}, но найдено {CurrentToken.Type}");

}

}

#endregion

#region Assignment and Expression Parsing

public void ParseAssignment()

{

if (CurrentToken.Type == TokenType.ID)

{

string varName = CurrentToken.Value;

Match(TokenType.ID);

Match(TokenType.ASSIGN);

ParseExpression();

\_postfix.WriteCmd(ECmd.SET);

\_postfix.PushVar(varName);

if (CurrentToken.Type == TokenType.SEMICOLON)

{

Match(TokenType.SEMICOLON);

}

else

{

throw new Exception("Ошибка: Ожидалась ';' после присваивания.");

}

}

}

private void ParseExpression()

{

ParseTerm();

while (CurrentToken.Type == TokenType.PLUS || CurrentToken.Type == TokenType.MINUS)

{

if (CurrentToken.Type == TokenType.PLUS)

{

Match(TokenType.PLUS);

ParseTerm();

\_postfix.WriteCmd(ECmd.ADD);

}

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

{

Match(TokenType.MINUS);

ParseTerm();

\_postfix.WriteCmd(ECmd.SUB);

}

}

}

private void ParseTerm()

{

ParseFactor();

while (CurrentToken.Type == TokenType.MULT || CurrentToken.Type == TokenType.DIV)

{

if (CurrentToken.Type == TokenType.MULT)

{

Match(TokenType.MULT);

ParseFactor();

\_postfix.WriteCmd(ECmd.MUL);

}

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

{

Match(TokenType.DIV);

ParseFactor();

\_postfix.WriteCmd(ECmd.DIV);

}

}

}

private void ParseFactor()

{

if (CurrentToken.Type == TokenType.ID)

{

\_postfix.PushVar(CurrentToken.Value);

Match(TokenType.ID);

}

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

{

\_postfix.PushConst(int.Parse(CurrentToken.Value));

Match(TokenType.CONST);

}

else

{

throw new Exception("Ожидалось арифметическое выражение");

}

}

#endregion

#region Logical and Relational Expressions

private void ParseLogicalExpression()

{

ParseRelationalExpression();

while (CurrentToken.Type == TokenType.AND || CurrentToken.Type == TokenType.OR)

{

if (CurrentToken.Type == TokenType.AND)

{

Match(TokenType.AND);

ParseRelationalExpression();

\_postfix.WriteCmd(ECmd.AND);

}

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

{

Match(TokenType.OR);

ParseRelationalExpression();

\_postfix.WriteCmd(ECmd.OR);

}

if (\_currentTokenIndex == \_tokens.Count)

{

break;

}

}

}

private void ParseRelationalExpression()

{

ParseOperand();

if (CurrentToken.Type == TokenType.REL)

{

string relOp = CurrentToken.Value;

Match(TokenType.REL);

ParseOperand();

switch (relOp)

{

case ">":

\_postfix.WriteCmd(ECmd.CMPG);

break;

case "<":

\_postfix.WriteCmd(ECmd.CMPL);

break;

case ">=":

\_postfix.WriteCmd(ECmd.CMPGE);

break;

case "<=":

\_postfix.WriteCmd(ECmd.CMPLE);

break;

case "==":

\_postfix.WriteCmd(ECmd.CMPE);

break;

default:

throw new Exception($"Неизвестная операция сравнения: {relOp}");

}

}

}

private void ParseOperand()

{

if (CurrentToken.Type == TokenType.ID)

{

\_postfix.PushVar(CurrentToken.Value);

Match(TokenType.ID);

}

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

{

\_postfix.PushConst(int.Parse(CurrentToken.Value));

Match(TokenType.CONST);

}

else

{

throw new Exception("Ошибка: ожидался операнд");

}

}

#endregion

#region Loop Parsing

public void ParseDoLoopUntil()

{

int startLoopIndex = \_postfix.GetCurrentAddress();

Match(TokenType.DO);

while (HasMoreTokens() && CurrentToken.Type != TokenType.LOOP)

{

ParseAssignment();

}

Match(TokenType.LOOP);

Match(TokenType.UNTIL);

ParseLogicalExpression();

int jzIndex = \_postfix.WriteCmd(ECmd.JZ);

\_postfix.WriteCmd(ECmd.JMP);

\_postfix.SetCmdPtr(jzIndex, \_postfix.GetCurrentAddress() + 1);

}

#endregion

public void PrintPostfix()

{

\_postfix.PrintPostfix();

}

public PostfixForm GetPostfixForm()

{

return \_postfix;

}

}

}

using Interpreter.Grammar;

using Interpreter.Stack;

using System;

using System.Collections.Generic;

namespace Interpreter.Postfix

{

public class PostfixForm

{

private List<PostfixEntry> \_postfix;

private ExecutionStack \_stack;

private Dictionary<string, int> \_variables;

private Dictionary<int, string> \_varNames = new Dictionary<int, string>();

public PostfixForm()

{

\_postfix = new List<PostfixEntry>();

\_stack = new ExecutionStack();

\_variables = new Dictionary<string, int>();

}

public void Interpret()

{

int pos = 0;

while (pos < \_postfix.Count)

{

var entry = \_postfix[pos];

switch (entry.Type)

{

case EEntryType.etCmd:

var cmd = (ECmd)entry.Index;

pos = ExecuteCommand(cmd, pos);

break;

case EEntryType.etVar:

\_stack.PushVal(GetVarValue(entry.Index));

pos++;

break;

case EEntryType.etConst:

\_stack.PushVal(entry.Index);

pos++;

break;

case EEntryType.etCmdPtr:

int condition = \_stack.PopVal();

if (condition == 0)

{

pos = entry.Index;

}

else

{

pos = pos+ 1;

// TODO возвращаемся в начало цикла, хардкод

\_stack.PushVal(0);

}

break;

default:

throw new Exception($"Неизвестный тип записи: {entry.Type}");

}

}

}

public int ExecuteCommand(ECmd cmd, int pos)

{

switch (cmd)

{

case ECmd.SET:

int value = \_stack.PopVal();

int varHash = \_postfix[++pos].Index;

SetVarValue(varHash, value);

return pos + 1;

case ECmd.ADD:

\_stack.PushVal(\_stack.PopVal() + \_stack.PopVal());

return pos + 1;

case ECmd.SUB:

int subtrahend = \_stack.PopVal();

\_stack.PushVal(\_stack.PopVal() - subtrahend);

return pos + 1;

case ECmd.MUL:

\_stack.PushVal(\_stack.PopVal() \* \_stack.PopVal());

return pos + 1;

case ECmd.DIV:

int divisor = \_stack.PopVal();

if (divisor == 0)

throw new DivideByZeroException("Ошибка: деление на ноль.");

\_stack.PushVal(\_stack.PopVal() / divisor);

return pos + 1;

case ECmd.CMPL:

\_stack.PushVal(\_stack.PopVal() > \_stack.PopVal() ? 1 : 0);

return pos + 1;

case ECmd.CMPLE:

\_stack.PushVal(\_stack.PopVal() >= \_stack.PopVal() ? 1 : 0);

return pos + 1;

case ECmd.CMPG:

\_stack.PushVal(\_stack.PopVal() < \_stack.PopVal() ? 1 : 0);

return pos + 1;

case ECmd.CMPGE:

\_stack.PushVal(\_stack.PopVal() <= \_stack.PopVal() ? 1 : 0);

return pos + 1;

case ECmd.CMPE:

\_stack.PushVal(\_stack.PopVal() == \_stack.PopVal() ? 1 : 0);

return pos + 1;

case ECmd.AND:

\_stack.PushVal((\_stack.PopVal() != 0 && \_stack.PopVal() != 0) ? 1 : 0);

return pos + 1;

case ECmd.OR:

\_stack.PushVal((\_stack.PopVal() != 0 || \_stack.PopVal() != 0) ? 1 : 0);

return pos + 1;

case ECmd.JMP:

return \_stack.PopVal();

case ECmd.JZ:

int address = \_stack.PopVal();

int condition = \_stack.PopVal();

return condition == 0 ? address : pos + 1;

default:

throw new Exception($"Неизвестная команда: {cmd}");

}

}

private int GetVarValue(int varHash)

{

string varKey = varHash.ToString();

if (!\_variables.ContainsKey(varKey))

{

Console.WriteLine($"Инициализация переменной: {varKey} со значением по умолчанию 0.");

\_variables[varKey] = 0;

}

return \_variables[varKey];

}

private void SetVarValue(int varHash, int value)

{

string varKey = varHash.ToString();

\_variables[varKey] = value;

}

public int WriteCmd(ECmd cmd)

{

\_postfix.Add(new PostfixEntry(EEntryType.etCmd, (int)cmd));

return \_postfix.Count - 1;

}

public int PushVar(string varName)

{

int index = varName.GetHashCode();

\_postfix.Add(new PostfixEntry(EEntryType.etVar, index));

if (!\_varNames.ContainsKey(index))

{

\_varNames.Add(index, varName);

}

return \_postfix.Count - 1;

}

public int PushConst(int value)

{

\_postfix.Add(new PostfixEntry(EEntryType.etConst, value));

return \_postfix.Count - 1;

}

public int WriteCmdPtr(int ptr)

{

\_postfix.Add(new PostfixEntry(EEntryType.etCmdPtr, ptr));

return \_postfix.Count - 1;

}

public void SetCmdPtr(int index, int ptr)

{

\_postfix[index] = new PostfixEntry(EEntryType.etCmdPtr, ptr);

}

public int GetCurrentAddress()

{

return \_postfix.Count - 1;

}

public void PrintPostfix()

{

Console.WriteLine("ПОЛИЗ:");

foreach (var entry in \_postfix)

{

string entryDescription;

switch (entry.Type)

{

case EEntryType.etCmd:

entryDescription = $"Команда: {((ECmd)entry.Index)}";

break;

case EEntryType.etVar:

entryDescription = $"Переменная (хэш): {entry.Index}";

break;

case EEntryType.etConst:

entryDescription = $"Константа: {entry.Index}";

break;

case EEntryType.etCmdPtr:

entryDescription = $"Указатель команды на адрес: {entry.Index}";

break;

default:

entryDescription = "Неизвестный тип записи";

break;

}

Console.WriteLine(entryDescription);

}

}

public void SetVarAndPop(string varName)

{

int varHash = varName.GetHashCode();

int value = \_stack.PopVal();

SetVarValue(varHash, value);

}

public void PrintVariables()

{

Console.WriteLine("Значения переменных:");

foreach (var kvp in \_variables)

{

int varHash = int.Parse(kvp.Key);

int value = kvp.Value;

if (\_varNames.ContainsKey(varHash))

{

string varName = \_varNames[varHash];

Console.WriteLine($"{varName} = {value}");

}

else

{

Console.WriteLine($"Переменная с хэшом {varHash} = {value}");

}

}

}

}

}

using System;

using System.Collections.Generic;

namespace Interpreter.Stack

{

public class ExecutionStack

{

private Stack<int> \_stack;

public ExecutionStack()

{

\_stack = new Stack<int>();

}

public int PopVal()

{

if (\_stack.Count == 0)

throw new InvalidOperationException("Ошибка: попытка извлечь значение из пустого стека.");

return \_stack.Pop();

}

public void PushVal(int value)

{

\_stack.Push(value);

}

public void PrintStack()

{

Console.WriteLine("Текущее состояние стека: ");

foreach (var item in \_stack)

{

Console.WriteLine(item);

}

}

}

}

namespace Interpreter.Grammar

{

public enum ECmd

{

JMP,

JZ,

SET,

ADD,

SUB,

AND,

OR,

CMPE,

CMPNE,

CMPL,

CMPLE,

MUL,

DIV,

CMPG,

CMPGE

}

}

namespace Interpreter.Grammar

{

public enum EEntryType

{

etCmd, // Команда (например, ADD, SUB, MUL, DIV, JZ, JMP и т.д.)

etVar, // Переменная (хэш-значение имени переменной, которая используется в выражении)

etConst, // Константа (например, числовое значение, которое напрямую участвует в выражении)

etCmdPtr // Указатель на команду (например, для команд перехода: JZ, JMP; адрес перехода в ПОЛИЗ)

}

}

namespace Interpreter.Grammar

{

public enum TokenType

{

ID,

CONST,

ASSIGN,

SEMICOLON,

COLON,

PLUS,

MINUS,

MULT,

OPENPAREN,

CLOSEPAREN,

WHILE,

DO,

END,

AND,

OR,

REL,

CLOSEBRACE,

OPENBRACE,

INCREMENT,

FOR,

LOOP,

UNTIL,

DIV

}

}

using Interpreter.Grammar;

using System;

using System.Collections.Generic;

namespace Interpreter.Lexer

{

public class Lexer

{

private readonly string \_input;

private int \_position;

public Lexer(string input)

{

\_input = input.ToLower();

\_position = 0;

}

public List<Token> Tokenize()

{

var tokens = new List<Token>();

while (\_position < \_input.Length)

{

var currentChar = \_input[\_position];

if (char.IsWhiteSpace(currentChar))

{

\_position++;

continue;

}

if (char.IsLetter(currentChar))

{

var identifier = ReadIdentifier();

if (identifier == "for")

{

tokens.Add(new Token(TokenType.FOR, identifier));

}

else if (identifier == "while")

{

tokens.Add(new Token(TokenType.WHILE, identifier));

}

else if (identifier == "and")

{

tokens.Add(new Token(TokenType.AND, identifier));

}

else if (identifier == "or")

{

tokens.Add(new Token(TokenType.OR, identifier));

}

else if (identifier == "do")

{

tokens.Add(new Token(TokenType.DO, identifier));

}

else if (identifier == "loop")

{

tokens.Add(new Token(TokenType.LOOP, identifier));

}

else if (identifier == "until")

{

tokens.Add(new Token(TokenType.UNTIL, identifier));

}

else if (identifier == "end")

{

tokens.Add(new Token(TokenType.END, identifier));

}

else

{

tokens.Add(new Token(TokenType.ID, identifier));

}

continue;

}

if (char.IsDigit(currentChar))

{

var number = ReadNumber();

tokens.Add(new Token(TokenType.CONST, number));

continue;

}

switch (currentChar)

{

case '=':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, "=="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.ASSIGN, "="));

\_position++;

}

break;

case '<':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, "<="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.REL, "<"));

\_position++;

}

break;

case '>':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '=')

{

tokens.Add(new Token(TokenType.REL, ">="));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.REL, ">"));

\_position++;

}

break;

case '+':

if (\_position + 1 < \_input.Length && \_input[\_position + 1] == '+')

{

tokens.Add(new Token(TokenType.INCREMENT, "++"));

\_position += 2;

}

else

{

tokens.Add(new Token(TokenType.PLUS, "+"));

\_position++;

}

break;

case '-':

tokens.Add(new Token(TokenType.MINUS, "-"));

\_position++;

break;

case '\*':

tokens.Add(new Token(TokenType.MULT, "\*"));

\_position++;

break;

case '(':

tokens.Add(new Token(TokenType.OPENPAREN, "("));

\_position++;

break;

case ')':

tokens.Add(new Token(TokenType.CLOSEPAREN, ")"));

\_position++;

break;

case '{':

tokens.Add(new Token(TokenType.OPENBRACE, "{"));

\_position++;

break;

case '}':

tokens.Add(new Token(TokenType.CLOSEBRACE, "}"));

\_position++;

break;

case ';':

tokens.Add(new Token(TokenType.SEMICOLON, ";"));

\_position++;

break;

case ':':

tokens.Add(new Token(TokenType.COLON, ":"));

\_position++;

break;

case '/':

tokens.Add(new Token(TokenType.DIV, "/"));

\_position++;

break;

default:

throw new Exception($"Неизвестный символ: {currentChar}");

}

}

return tokens;

}

private string ReadIdentifier()

{

var start = \_position;

while (\_position < \_input.Length && char.IsLetterOrDigit(\_input[\_position]))

{

\_position++;

}

return \_input.Substring(start, \_position - start);

}

private string ReadNumber()

{

var start = \_position;

while (\_position < \_input.Length && char.IsDigit(\_input[\_position]))

{

\_position++;

}

return \_input.Substring(start, \_position - start);

}

}

}

using Interpreter.Grammar;

namespace Interpreter.Lexer

{

public class Token

{

public TokenType Type { get; }

public string Value { get; }

public Token(TokenType type, string value)

{

Type = type;

Value = value;

}

public override string ToString()

{

return $"{Type} ({Value})";

}

}

}

using Interpreter.Grammar;

namespace Interpreter.Postfix

{

public class PostfixEntry

{

public EEntryType Type { get; set; }

public int Index { get; set; }

public PostfixEntry(EEntryType type, int index)

{

Type = type;

Index = index;

}

}

}