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.

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

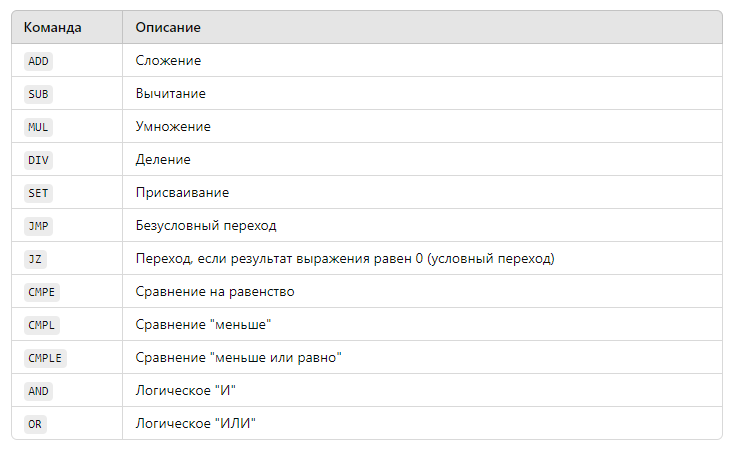
# Описание внутренней формы представления программы и набора операций

Внутренняя форма представления программы в данной работе реализована с помощью ПОЛИЗа (польской инверсной записи). ПОЛИЗ представляет собой удобное для интерпретации или трансляции в машинный код промежуточное представление программы, основанное на стековой машине. В ПОЛИЗе операции записываются после операндов (постфиксная запись), что упрощает выполнение операций с использованием стека.

**Основные элементы ПОЛИЗа:**

* Операции (Команды): В ПОЛИЗе представлены базовые команды для арифметических и логических операций, таких как сложение, умножение, сравнение, условные и безусловные переходы. Эти команды отображаются как ECmd в программе.
* Операнды: Операндами могут быть переменные или константы. Переменные и константы передаются в команду и обрабатываются в стеке. Переменные кодируются через хэш-коды их имен.
* Указатели на команды: Используются для указания на начало цикла или условные/безусловные переходы.

**Пример набора команд:**



Пример структуры ПОЛИЗа:

**В контексте разбираемого выражения:**

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

Сгенерированный ПОЛИЗ будет выглядеть следующим образом:

**Присваивание:**

x x 10 2 \* + = // x = x + 10 \* 2

y y 3 / 1 - = // y = y / 3 - 1

**Условие выхода из цикла:**

x 100 > y 5 < AND z 50 = OR JZ // Условие выхода

**Переходы:**

JMP // Безусловный переход в начало цикла

# Описание типов данных, используемых для хранения внутреннего представления программы.

Для хранения внутреннего представления программы используется несколько основных типов данных, которые обеспечивают представление элементов программы (переменные, константы, команды) в виде ПОЛИЗа. Эти данные организованы в структуры, которые упрощают построение, хранение и обработку программы на этапе семантического анализа.

Основные типы данных:

**PostfixEntry:** Это основной класс, представляющий единичный элемент ПОЛИЗа. Каждый элемент ПОЛИЗа может быть:

* Командой (арифметическая операция, условный или безусловный переход).
* Переменной (сохраненной как хэш-код имени переменной).
* Константой (значение, например, целое число).
* Указателем на команду (для реализации переходов, например, в циклах или условных операторах).

Этот тип данных используется для каждого элемента ПОЛИЗа и сохраняет информацию о том, что это за элемент (команда, переменная и т.д.) и какой его индекс/значение.

**Перечисление** **EEntryType:** Это перечисление определяет возможные типы элементов ПОЛИЗа:

* etCmd — команда (например, арифметическая операция).
* etVar — переменная.
* etConst — константа.
* etCmdPtr — указатель на команду (для условных и безусловных переходов).

**Перечисление ECmd:** Это перечисление определяет набор возможных команд, которые могут быть использованы в ПОЛИЗе:

* Арифметические операции (ADD, SUB, MUL, DIV).
* Логические операции (AND, OR).
* Операции сравнения (CMPE, CMPL, CMPLE).
* Управляющие команды для циклов и условий (JMP, JZ).
* Операция присваивания (SET).

**List<PostfixEntry>:** Для хранения самого ПОЛИЗа используется список элементов PostfixEntry. Этот список динамически заполняется по мере синтаксического разбора программы, и каждый элемент списка представляет собой либо команду, либо операнд (переменную или константу).

Этот список реализован как коллекция, которая хранит последовательность команд, переменных, констант и указателей в порядке выполнения программы

**PostfixForm:** Это класс, который управляет созданием ПОЛИЗа и предоставляет методы для добавления команд, переменных, констант и указателей. Он также отвечает за вывод сгенерированного ПОЛИЗа для отладки и интерпретации.

**Основные методы:**

* WriteCmd(ECmd cmd) — добавляет команду в ПОЛИЗ.
* WriteVar(string varName) — добавляет переменную в ПОЛИЗ (по хэш-коду имени).
* WriteConst(int value) — добавляет константу в ПОЛИЗ.
* WriteCmdPtr(int ptr) — добавляет указатель на команду.
* SetCmdPtr(int index, int newPtr) — обновляет указатель на команду для реализации условных и безусловных переходов.

# Функции и процедуры, в которые осуществлялось включение семантики с подробным описанием.

В процессе работы семантика программы (то есть выполнение логических и арифметических операций, работа с переменными и переходами) включалась в нескольких ключевых функциях и процедурах. Эти функции ответственны за генерацию ПОЛИЗа, построение внутреннего представления программы и обеспечение корректной работы с выражениями, условиями и циклами.

Ниже приведены основные функции и процедуры, в которые осуществлялось включение семантики.

**ParseAssignment() — Обработка присваивания**

Эта функция отвечает за обработку операторов присваивания в программе. Она разбирает выражение вида x = выражение; и генерирует команды ПОЛИЗа для присваивания значения переменной.

**Основные шаги:**

* Сначала функция проверяет наличие идентификатора переменной (например, x).
* Затем она разбирает выражение после оператора присваивания (=) и вызывает функцию для генерации ПОЛИЗа для этого выражения.
* В конце функция добавляет команду присваивания SET в ПОЛИЗ, которая присваивает результат вычисленного выражения переменной.

**Пример вызова семантики:**

* Генерация команды присваивания SET.
* Вызов функций для разбора арифметических операций и переменных.

public void ParseAssignment()

{

if (CurrentToken.Type == TokenType.ID)

{

\_postfix.WriteVar(CurrentToken.Value);

Match(TokenType.ID);

Match(TokenType.ASSIGN);

ParseExpression();

\_postfix.WriteCmd(ECmd.SET);

if (CurrentToken.Type == TokenType.SEMICOLON)

{

Match(TokenType.SEMICOLON);

}

else

{

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

}

}

}

**Включение семантики:**

* Переменная добавляется в ПОЛИЗ через команду WriteVar().
* Присваивание обрабатывается через команду SET, которая сохраняет результат вычисленного выражения в переменной.

**ParseExpression() — Разбор арифметических выражений**

Функция отвечает за разбор арифметических выражений. Она генерирует команды для выполнения операций сложения, вычитания, умножения и деления, которые используются для вычисления значений в выражении.

**Основные шаги:**

* Функция разбирает термы (числа или переменные) и операции между ними (например, +, -, \*, /).
* Для каждой арифметической операции генерируется соответствующая команда ПОЛИЗа (ADD, SUB, MUL, DIV).

**Включение семантики:**

В зависимости от оператора (+, -), генерируются команды ADD или SUB для выполнения арифметических операций над операндами.

**ParseTerm() — Разбор термов**

Эта функция разбирает составные части выражений, такие как переменные, константы или подвыражения. Она также обрабатывает операции умножения и деления.

**Включение семантики:**

При нахождении оператора умножения или деления добавляются команды MUL или DIV, которые обеспечивают выполнение соответствующей операции над операндами.

**ParseLogicalExpression() — Разбор логических выражений**

Функция отвечает за разбор логических выражений, таких как x > 100 and y < 5. Логические выражения могут состоять из сравнений и логических операций AND и OR.

**Основные шаги**:

* Сначала разбираются отношения (например, x > 100), затем логические операторы.
* Генерируются команды для выполнения логических операций, таких как AND и OR.

**Включение семантики:**

В зависимости от оператора генерируются команды AND или OR, которые выполняют логические операции в ПОЛИЗе.

**ParseDoLoopUntil() — Разбор цикла do ... loop until**

Эта функция разбирает цикл do ... loop until и генерирует ПОЛИЗ для выполнения операций внутри цикла, а также для проверки условия выхода.

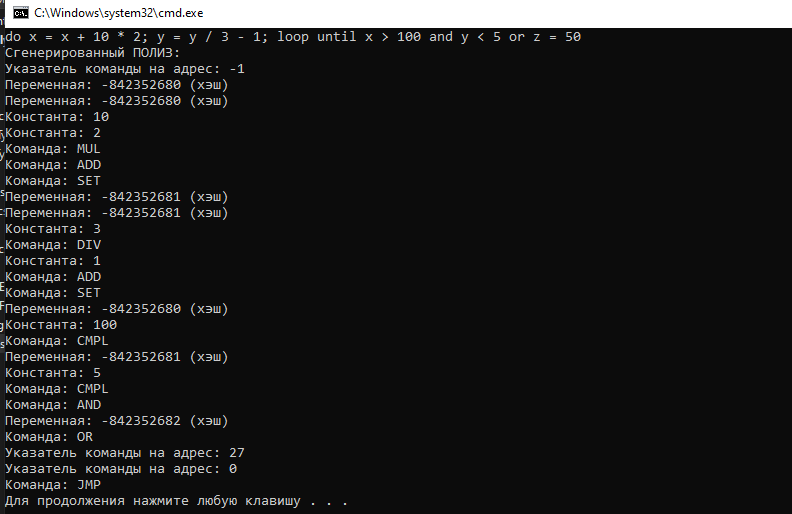
**Основные шаги:**

* Генерируется ПОЛИЗ для тела цикла.
* Генерируются команды для проверки условия выхода из цикла (JZ — условный переход).
* В конце цикла генерируется безусловный переход на начало цикла (JMP).

**Включение семантики:**

* Тело цикла добавляется в ПОЛИЗ с операциями внутри цикла.
* Условие выхода обрабатывается через команду JZ (переход, если условие истинно).
* Переходы: Генерируются команды переходов, чтобы вернуться к началу цикла или выйти из него.

# Контрольный пример и результаты тестирования



# Листинг программы.

using SemanticAnalyzer.Lexer;

using SemanticAnalyzer.Parser;

using System;

using System.Collections.Generic;

namespace SemanticAnalyzer

{

class Program

{

static void Main(string[] args)

{

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

string input = Console.ReadLine();

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();

}

catch (Exception ex)

{

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

}

}

}

}

using RcursiveDescentParser.Grammar;

namespace RcursiveDescentParser.Postfix

{

public class PostfixEntry

{

public EEntryType Type { get; set; }

public int Index { get; set; }

public PostfixEntry(EEntryType type, int index)

{

Type = type;

Index = index;

}

}

}

namespace RcursiveDescentParser.Grammar

{

public enum ECmd

{

JMP,

JZ,

SET,

ADD,

SUB,

AND,

OR,

CMPE,

CMPNE,

CMPL,

CMPLE,

MUL,

DIV

}

}

namespace RcursiveDescentParser.Grammar

{

public enum EEntryType

{

etCmd,

etVar,

etConst,

etCmdPtr

}

}

using RcursiveDescentParser.Grammar;

using System;

using System.Collections.Generic;

namespace RcursiveDescentParser.Postfix

{

public class PostfixForm

{

private List<PostfixEntry> \_postfix;

public PostfixForm()

{

\_postfix = new List<PostfixEntry>();

}

public int WriteCmd(ECmd cmd)

{

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

return \_postfix.Count - 1;

}

public int WriteVar(string varName)

{

int index = varName.GetHashCode();

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

return \_postfix.Count - 1;

}

public int WriteConst(int ind)

{

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

return \_postfix.Count - 1;

}

public int WriteCmdPtr(int ptr)

{

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

return \_postfix.Count - 1;

}

public void SetCmdPtr(int ind, int ptr)

{

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

}

public int GetCurrentAddress()

{

return \_postfix.Count - 1;

}

public void PrintPostfix()

{

foreach (var entry in \_postfix)

{

string entryDescription = "";

switch (entry.Type)

{

case EEntryType.etCmd:

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

break;

case EEntryType.etVar:

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

break;

case EEntryType.etConst:

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

break;

case EEntryType.etCmdPtr:

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

break;

default:

entryDescription = "Неизвестный тип";

break;

}

Console.WriteLine(entryDescription);

}

}

}

}

using RcursiveDescentParser.Grammar;

using RcursiveDescentParser.Postfix;

using SemanticAnalyzer.Grammar;

using SemanticAnalyzer.Lexer;

using System;

using System.Collections.Generic;

namespace SemanticAnalyzer.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)

{

\_postfix.WriteVar(CurrentToken.Value);

Match(TokenType.ID);

Match(TokenType.ASSIGN);

ParseExpression();

\_postfix.WriteCmd(ECmd.SET);

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.WriteVar(CurrentToken.Value);

Match(TokenType.ID);

}

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

{

\_postfix.WriteConst(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);

}

}

}

private void ParseRelationalExpression()

{

ParseOperand();

if (CurrentToken.Type == TokenType.REL)

{

string relOp = CurrentToken.Value;

Match(TokenType.REL);

ParseOperand();

switch (relOp)

{

case ">":

\_postfix.WriteCmd(ECmd.CMPL);

break;

case "<":

\_postfix.WriteCmd(ECmd.CMPL);

break;

case ">=":

\_postfix.WriteCmd(ECmd.CMPLE);

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.WriteVar(CurrentToken.Value);

Match(TokenType.ID);

}

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

{

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

Match(TokenType.CONST);

}

else

{

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

}

}

#endregion

#region Loop Parsing

public void ParseDoLoopUntil()

{

int startLoopIndex = \_postfix.WriteCmdPtr(-1);

Match(TokenType.DO);

while (CurrentToken.Type != TokenType.LOOP)

{

ParseAssignment();

}

Match(TokenType.LOOP);

Match(TokenType.UNTIL);

ParseLogicalExpression();

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

\_postfix.WriteCmdPtr(startLoopIndex);

\_postfix.WriteCmd(ECmd.JMP);

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

}

#endregion

public void PrintPostfix()

{

\_postfix.PrintPostfix();

}

}

}

using SemanticAnalyzer.Grammar;

using System;

using System.Collections.Generic;

namespace SemanticAnalyzer.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.PLUS, "-"));

\_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);

}

}

}

namespace SemanticAnalyzer.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 SemanticAnalyzer.Grammar;

namespace SemanticAnalyzer.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})";

}

}

}

# Контрольные вопросы.

**1) Что такое внутренняя форма представления программы? Зачем она используется?**

Внутренняя форма представления программы — это промежуточная форма, в которую трансформируется исходный код программы после лексического и синтаксического анализа. Она представляет собой удобную для обработки и оптимизации структуру, которая может быть использована компилятором или интерпретатором для генерации машинного кода.

Используется она для упрощения дальнейших этапов трансляции и выполнения программы, таких как семантический анализ, оптимизация и генерация кода для конкретной платформы.

**2) Какие формы внутреннего представления программ Вы знаете?**

* **ПОЛИЗ** (польская инверсная запись) — постфиксная форма представления программы.
* **Абстрактное синтаксическое дерево** (AST) — древовидная структура, представляющая синтаксис программы.
* **Трехадресный код** — промежуточная форма представления, в которой каждая инструкция имеет до трех операндов.
* **Таблицы символов** — содержат информацию о переменных, функциях и других элементах программы.

**3) Что такое трехадресный код? Приведите пример.**

Трехадресный код — это форма промежуточного представления программы, в которой каждая инструкция содержит не более трех операндов: два операнда для операции и один для результата.

Пример:

t1 = 10 \* 2

t2 = x + t1

t3 = y / 3

t4 = t3 – 1

**4) В чем отличие триад и тетрад? Приведите пример.**

**Триады** — форма представления, где каждая операция имеет три компонента: оператор, левый и правый операнд. Результат неявно хранится в промежуточной переменной.

**Тетрады** — в отличие от триад, в тетрадах явным образом указывается результат операции.

Пример триады:

(\*, 10, 2) // умножение 10 на 2

(+, x, t1) // сложение x и результата предыдущей операции

Пример тетрады:

t1 = 10 \* 2

t2 = x + t1

**5) В чем особенности постфиксной формы записи?**

Постфиксная форма записи (ПОЛИЗ) упрощает выполнение выражений, так как операции выполняются над операндами сразу после их появления. В ПОЛИЗе нет необходимости в скобках для определения приоритетов операций, так как они определяются порядком записи.

Пример:

Инфиксное выражение: x = 10 + 2 \* 3 Постфиксное: x 10 2 3 \* + =

**6) Как вычислить выражение в ПОЛИЗе вручную?**

Вычисление выражения в ПОЛИЗе выполняется с помощью стека:

* Операнды (переменные и константы) помещаются в стек.
* Как только встречается операция, из стека извлекаются соответствующие операнды, производится операция, и результат снова помещается в стек.

Пример: ПОЛИЗ: 10 2 3 \* +

1. Помещаем 10 в стек.
2. Помещаем 2 в стек.
3. Помещаем 3 в стек.
4. Выполняем \*: 2 \* 3 = 6, заменяем в стеке.
5. Выполняем +: 10 + 6 = 16.

**7) Как перевести выражение в ПОЛИЗ вручную?**

Для перевода выражения в ПОЛИЗ (постфиксную запись) используется метод стеков:

* Операнды записываются непосредственно в выходную строку.
* Операции помещаются в стек и извлекаются в момент, когда встречается операция с более низким приоритетом.

Пример: Инфиксное выражение: x = 10 + 2 \* 3 ПОЛИЗ: x 10 2 3 \* + **=**

**8) Как представить в ПОЛИЗе операции присваивания и обращения по индексу?**

* Присваивание в ПОЛИЗе представляется операцией SET, которая выполняется после вычисления правой части выражения
* **Обращение по индексу** представляется как операция получения значения переменной или массива по индексу, которое затем используется для вычисления.

Пример:

x 10 2 \* + SET

**9) Как представить в ПОЛИЗе условный оператор?**

Условный оператор в ПОЛИЗе представляется с помощью команды условного перехода JZ (jump if zero) или аналогичной. После вычисления условия, если оно ложно, выполняется переход по указанному адресу.

Пример:

x 10 > JZ метка\_выхода

**10) Как представить в ПОЛИЗе операторы цикла?**

Операторы цикла в ПОЛИЗе представлены двумя основными частями:

* Условие выхода (генерируется как условный переход JZ).
* Переход на начало цикла (безусловный переход JMP).

Пример цикла:

метка\_начала:

x 10 +

y 5 > JZ метка\_выхода

JMP метка\_начала

**11) Существуют ли преимущества у постфиксной формы записи перед традиционной?**

Да, постфиксная форма записи (ПОЛИЗ) имеет несколько преимуществ:

* Отсутствие необходимости в скобках: порядок операций определяется автоматически.
* Удобство для выполнения: ПОЛИЗ легко интерпретируется или выполняется с помощью стековых машин.
* Простота реализации: операции выполняются сразу при их встрече, что упрощает обработку выражений без дополнительных шагов по приоритизации операций.