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 №1. Построение лексического анализатора.

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.

Составить автоматную грамматику и на ее основе реализовать лексический анализатор языка, цепочки которого имеют вид, указанный в задании. Лексический анализатор должен преобразовывать исходный текст в последовательность лексем. По результатам работы анализатора должны формироваться таблицы идентификаторов и констант.

# Описание цепочек анализируемого языка.

Лексический анализатор, разработанный в рамках данной лабораторной работы, способен распознавать следующие типы цепочек, которые состоят из базовых элементов языка:

**Цикл:**

Описание: Конструкция цикла с предусловием.

Формат: while <условие> do <оператор> end

Пример: while a < b do b = b + 1 end

**Условие:**

Описание: Логическое выражение, которое может состоять из одного или нескольких сравнений, объединенных логическими операциями.

Формат: <сравнение> | <условие> <логическая операция> <сравнение>

Пример: a < b and b <= c

**Сравнение:**

Описание: Операция сравнения между двумя операндами.

Формат: <операнд> | <операнд> <операция сравнения> <операнд>

Пример: a < b

**Операция сравнения:**

Описание: Операторы, используемые для сравнения двух операндов.

Формат: < | <= | <> | == | =

Пример: a <= b

**Операнд:**

Описание: Операндом может быть либо идентификатор, либо числовая константа.

Формат: <идентификатор> | <константа>

Пример: a, 123

**Логическая операция:**

Описание: Логические операторы для объединения нескольких условий.

Формат: and | or

Пример: a < b and b <= c

**Оператор:**

Описание: Присваивание значения идентификатору с использованием арифметических операций.

Формат: <идентификатор> = <арифметическое выражение>

Пример: b = b + c - 20

**Арифметическое выражение:**

Описание: Выражение, состоящее из операндов и арифметических операций.

Формат: <операнд> | <арифметическое выражение> <арифметическая операция> <операнд>

Пример: b + c - 20

**Арифметическая операция:**

Описание: Операции сложения и вычитания.

Формат: + | -

Пример: +, -

**Примечание:**

Лексический анализатор распознает ключевые слова, операторы сравнения, арифметические операторы, идентификаторы и числовые константы. Элементы языка могут быть разделены произвольным количеством пробельных символов, которые игнорируются при анализе.

# Таблица терминальных символов с подробным описанием.



Описание:

Ключевые слова: while, do, end, and, or — используются для управления потоком выполнения программы.

Операции сравнения: <, <=, <>, == — для проверки условий.

Операторы присваивания и арифметические операторы: =, +, - — для работы с выражениями.

# Описание основных типов лексем с использованием грамматик.

В данной работе лексический анализатор распознает следующие типы лексем:

**Ключевые слова:**

Описание: Ключевые слова языка, зарезервированные для управления потоком выполнения программы.

Тип лексемы: KeywordWhile, KeywordDo, KeywordEnd, KeywordAnd, KeywordOr

Правило грамматики:

<ключевое слово> ::= "while" | "do" | "end" | "and" | "or"

**Идентификаторы:**

Описание: Идентификаторы, которые используются для обозначения переменных или функций в программе. Идентификаторы должны начинаться с буквы и могут содержать буквы и цифры.

Тип лексемы: Identifier

Правило грамматики:

<идентификатор> ::= <буква> {<буква> | <цифра>}

где <буква> — это любой символ латинского алфавита, а <цифра> — это цифры от 0 до 9.

**Константы**

Описание: Числовые константы (целые числа), состоящие только из цифр.

Тип лексемы: Constant

Правило грамматики:

<константа> ::= <цифра> {<цифра>}

где <цифра> — это цифры от 0 до 9.

**Операторы сравнения**

Описание: Операторы, которые используются для сравнения двух значений.

Тип лексемы: RelationalOperator

Правило грамматики:

<оператор сравнения> ::= "<" | "<=" | "<>" | "==" | ">" | ">="

**Арифметические операторы**

Описание: Операторы, которые используются для выполнения арифметических операций (сложение и вычитание).

Тип лексемы: ArithmeticOperator

Правило грамматики:

<арифметическая операция> ::= "+" | "-"

**Операторы присваивания**

Описание: Операторы, которые используются для присваивания значения переменной.

Тип лексемы: AssignmentOperator

Правило грамматики:

<оператор присваивания> ::= "="

# Описание грамматики автоматного языка в виде графа

**Состояния автомата:**

S — начальное состояние.

Ai — состояние для распознавания идентификаторов.

Ac — состояние для распознавания констант.

As — состояние для распознавания операторов сравнения.

Cs — состояние для арифметических операторов.

F — принимающее состояние (конец строки или успешное распознавание лексемы).

E — ошибка (недопустимый символ).

**Переходы автомата:**

Из начального состояния S:

Если символ — пробел или разделитель: остаемся в состоянии S.

Если символ — буква: переходим в состояние Ai (распознавание идентификатора).

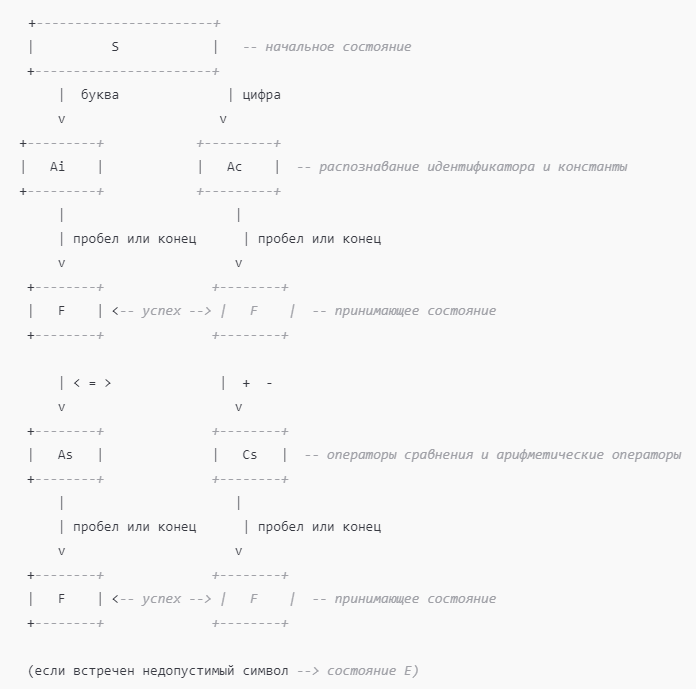
Если символ — цифра: переходим в состояние Ac (распознавание константы).

Если символ — оператор <, =, >: переходим в состояние As (операции сравнения).

Если символ — оператор + или -: переходим в состояние Cs (арифметические операторы).

Если конец строки: переходим в состояние F (успешное завершение).

В остальных случаях: состояние E (ошибка).



# Описание основных алгоритмов и структур данных, используемых в программе

**А) Описание структур данных, используемых для представления последовательности лексем, таблиц идентификаторов и констант**

**Лексемы (Token):**

Структура: Класс Token представляет каждую лексему, распознанную лексическим анализатором. Он содержит следующие поля:

Value: строковое значение лексемы (например, "while", "a", "123").

Type: тип лексемы, представлен через enum TokenType, где указаны типы ключевых слов, операторов, идентификаторов и констант.

Position: позиция лексемы в исходной строке.

Использование: Лексемы собираются в список (List<Token>) после анализа входной строки и используются для дальнейшего синтаксического анализа или вывода.

**Таблица идентификаторов и констант (SymbolTable):**

Структура: Два словаря (Dictionary<string, int>) для хранения:

Идентификаторов (\_identifiers): сопоставляет строковое значение идентификатора с его порядковым номером.

Констант (\_constants): аналогичная структура для числовых констант.

Использование: При распознавании идентификаторов или констант они добавляются в таблицу с уникальным идентификатором (если они еще не были добавлены).

**Таблица терминальных символов (TerminalSymbolTable):**

Структура: Список объектов TerminalSymbol, каждый из которых описывает терминальный символ (например, "while", "+", "<") с указанием индекса, символа, категории, типа и комментария.

Использование: Используется для вывода всех возможных терминальных символов языка. Структура не участвует непосредственно в анализе, но служит для представления возможных символов.

**Б) Описание процедур и функций, отвечающих за работу с введенными структурами данных**

**Процедуры для работы с лексемами (класс Lexer):**

**Analyze():** Главная функция, которая выполняет лексический анализ входной строки. Она последовательно читает символы и формирует лексемы на основе правил:

* Пробелы и разделители игнорируются.
* Буквы распознаются как идентификаторы или ключевые слова.
* Цифры распознаются как константы.
* Символы <, =, > и т.д. распознаются как операторы сравнения.
* Символы +, - — как арифметические операторы.
* Неизвестные символы классифицируются как Unknown.

**ReadWhile():** Читает символы строки, пока они удовлетворяют заданному условию (например, пока символ — буква или цифра).

**IdentifyKeyword():** Определяет, является ли строка ключевым словом (например, "while", "do"). Если не является, строка считается идентификатором.

**Процедуры для работы с таблицей идентификаторов и констант (класс SymbolTable):**

* AddIdentifier(): Добавляет новый идентификатор в таблицу, если его там еще нет.
* AddConstant(): Добавляет новую константу в таблицу.
* GetIdentifierId(): Возвращает порядковый номер идентификатора, если он существует в таблице, или -1.
* GetConstantId(): Возвращает порядковый номер константы.

**Процедуры для работы с таблицей терминальных символов (класс TerminalSymbolTable):**

* GetTable(): Возвращает список всех терминальных символов.
* PrintTable(): Выводит таблицу терминальных символов в консоль.

**В) Описание алгоритма анализа автоматного языка**

Алгоритм лексического анализа реализован в классе Lexer и работает как конечный автомат, переходя из одного состояния в другое на основе прочитанных символов:

**Инициализация:**

Входная строка передается в конструктор класса Lexer, где сохраняется как \_input, а индекс текущей позиции устанавливается в 0.

**Цикл анализа:**

* Алгоритм начинается с начального состояния и перебирает символы входной строки.
* Для каждого символа проверяется, является ли он пробелом, буквой, цифрой, оператором или неизвестным символом.
* В зависимости от символа, автомат переходит в одно из состояний:
  + Идентификаторы и ключевые слова: если символ — буква, запускается процесс распознавания идентификатора или ключевого слова.
  + Константы: если символ — цифра, запускается процесс распознавания числовой константы.
  + Операторы: если символ — один из операторов сравнения или арифметических операций, он классифицируется как соответствующий оператор.
  + Неизвестные символы: если символ не соответствует ни одному правилу, он классифицируется как Unknown.

**Создание токенов:**

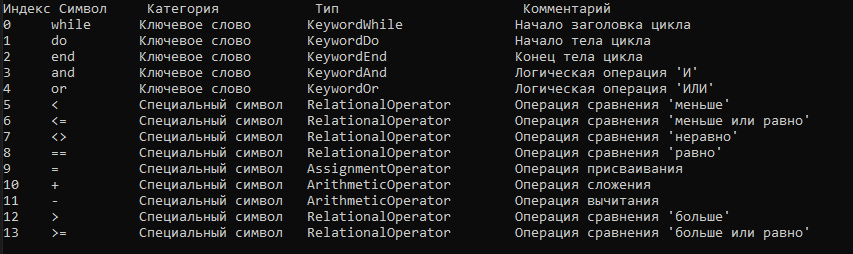
* Когда очередная лексема распознана, создается объект класса Token, содержащий значение лексемы, ее тип и позицию в строке.
* Все токены добавляются в список, который возвращается после завершения анализа.

**Завершение анализа:**

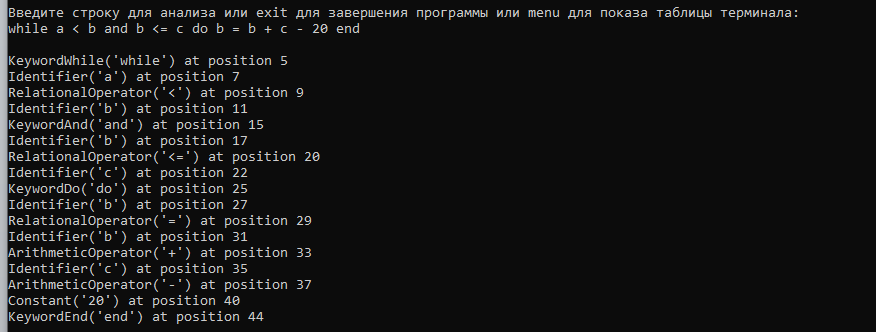
Когда все символы входной строки обработаны, цикл завершает работу, и возвращается список токенов.

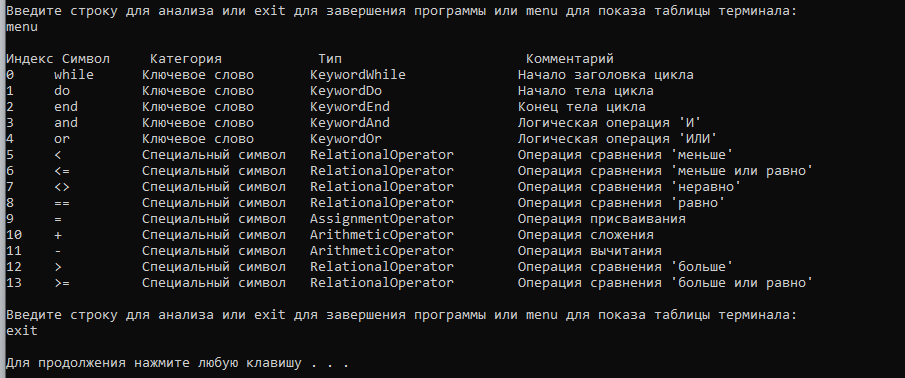
# Описание интерфейса пользователя программы.

Изначально выводятся все наши ключевые слова:



Дальше можно ввести 2 слова menu для показа таблицы терминала или exit для выхода. Либо ввести строку для анализа.





# Листинг программы

using LexicalAnalyzer.SymbolTable;

using System;

namespace LexicalAnalyzer

{

public class Program

{

static void Main(string[] args)

{

TerminalSymbolTable.PrintTable();

Console.WriteLine();

while (true)

{

Console.WriteLine("Введите строку для анализа или exit для завершения программы или menu для показа таблицы терминала:");

string input = Console.ReadLine();

Console.WriteLine();

if (input.ToLower() == "exit")

{

break;

}

else if (input.ToLower() == "menu")

{

TerminalSymbolTable.PrintTable();

}

else

{

var lexer = new Lexer.Lexer(input);

var tokens = lexer.Analyze();

foreach (var token in tokens)

{

Console.WriteLine(token);

}

}

Console.WriteLine();

}

}

}

}

using LexicalAnalyzer.Grammar;

namespace LexicalAnalyzer.Lexer

{

public class Token

{

public string Value { get; }

public TokenType Type { get; }

public int Position { get; }

public Token(string value, TokenType type, int position)

{

Value = value;

Type = type;

Position = position;

}

public override string ToString()

{

return $"{Type}('{Value}') at position {Position}";

}

}

}

namespace LexicalAnalyzer.Grammar

{

public static class GrammarRules

{

public static bool IsKeyword(string value)

{

return value == "while" || value == "do" || value == "end" || value == "and" || value == "or";

}

public static bool IsRelationalOperator(char c)

{

return c == '<' || c == '=' || c == '>';

}

public static bool IsArithmeticOperator(char c)

{

return c == '+' || c == '-';

}

public static bool IsIdentifierStart(char c)

{

return char.IsLetter(c);

}

public static bool IsIdentifierPart(char c)

{

return char.IsLetterOrDigit(c);

}

}

}

namespace LexicalAnalyzer.Grammar

{

public enum TokenType

{

KeywordWhile,

KeywordDo,

KeywordEnd,

KeywordAnd,

KeywordOr,

RelationalOperator,

ArithmeticOperator,

AssignmentOperator,

Identifier,

Constant,

Unknown

}

}

using System;

using System.Collections.Generic;

using LexicalAnalyzer.Grammar;

namespace LexicalAnalyzer.Lexer

{

public class Lexer

{

private readonly string \_input;

private int \_position;

public Lexer(string input)

{

\_input = input;

\_position = 0;

}

public List<Token> Analyze()

{

var tokens = new List<Token>();

while (\_position < \_input.Length)

{

if (char.IsWhiteSpace(CurrentChar()))

{

\_position++;

}

else if (IsLetter(CurrentChar()))

{

var identifier = ReadWhile(IsLetterOrDigit);

tokens.Add(new Token(identifier, IdentifyKeyword(identifier), \_position));

}

else if (char.IsDigit(CurrentChar()))

{

var constant = ReadWhile(char.IsDigit);

tokens.Add(new Token(constant, TokenType.Constant, \_position));

}

else if (CurrentChar() == '<' || CurrentChar() == '=')

{

var symbol = ReadWhile(c => c == '<' || c == '=');

tokens.Add(new Token(symbol, TokenType.RelationalOperator, \_position));

}

else if (CurrentChar() == '+' || CurrentChar() == '-')

{

var symbol = ReadWhile(c => c == '+' || c == '-');

tokens.Add(new Token(symbol, TokenType.ArithmeticOperator, \_position));

}

else

{

tokens.Add(new Token(CurrentChar().ToString(), TokenType.Unknown, \_position));

\_position++;

}

}

return tokens;

}

private char CurrentChar()

{

return \_input[\_position];

}

private string ReadWhile(Func<char, bool> condition)

{

var start = \_position;

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

{

\_position++;

}

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

}

private bool IsLetter(char ch) => char.IsLetter(ch);

private bool IsLetterOrDigit(char ch) => char.IsLetterOrDigit(ch);

private TokenType IdentifyKeyword(string value)

{

switch (value)

{

case "while": return TokenType.KeywordWhile;

case "do": return TokenType.KeywordDo;

case "end": return TokenType.KeywordEnd;

case "and": return TokenType.KeywordAnd;

case "or": return TokenType.KeywordOr;

default: return TokenType.Identifier;

}

}

}

}

using System.Collections.Generic;

namespace LexicalAnalyzer.SymbolTable

{

public class SymbolTable

{

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

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

public void AddIdentifier(string identifier)

{

if (!\_identifiers.ContainsKey(identifier))

{

\_identifiers[identifier] = \_identifiers.Count + 1;

}

}

public void AddConstant(string constant)

{

if (!\_constants.ContainsKey(constant))

{

\_constants[constant] = \_constants.Count + 1;

}

}

public int GetIdentifierId(string identifier)

{

return \_identifiers.ContainsKey(identifier) ? \_identifiers[identifier] : -1;

}

public int GetConstantId(string constant)

{

return \_constants.ContainsKey(constant) ? \_constants[constant] : -1;

}

}

}

using System;

using System.Collections.Generic;

namespace LexicalAnalyzer.SymbolTable

{

public class TerminalSymbol

{

public int Index { get; set; }

public string Symbol { get; set; }

public string Category { get; set; }

public string Type { get; set; }

public string Comment { get; set; }

}

public class TerminalSymbolTable

{

public static List<TerminalSymbol> GetTable()

{

return new List<TerminalSymbol>

{

new TerminalSymbol { Index = 0, Symbol = "while", Category = "Ключевое слово", Type = "KeywordWhile", Comment = "Начало заголовка цикла" },

new TerminalSymbol { Index = 1, Symbol = "do", Category = "Ключевое слово", Type = "KeywordDo", Comment = "Начало тела цикла" },

new TerminalSymbol { Index = 2, Symbol = "end", Category = "Ключевое слово", Type = "KeywordEnd", Comment = "Конец тела цикла" },

new TerminalSymbol { Index = 3, Symbol = "and", Category = "Ключевое слово", Type = "KeywordAnd", Comment = "Логическая операция 'И'" },

new TerminalSymbol { Index = 4, Symbol = "or", Category = "Ключевое слово", Type = "KeywordOr", Comment = "Логическая операция 'ИЛИ'" },

new TerminalSymbol { Index = 5, Symbol = "<", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'меньше'" },

new TerminalSymbol { Index = 6, Symbol = "<=", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'меньше или равно'" },

new TerminalSymbol { Index = 7, Symbol = "<>", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'неравно'" },

new TerminalSymbol { Index = 8, Symbol = "==", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'равно'" },

new TerminalSymbol { Index = 9, Symbol = "=", Category = "Специальный символ", Type = "AssignmentOperator", Comment = "Операция присваивания" },

new TerminalSymbol { Index = 10, Symbol = "+", Category = "Специальный символ", Type = "ArithmeticOperator", Comment = "Операция сложения" },

new TerminalSymbol { Index = 11, Symbol = "-", Category = "Специальный символ", Type = "ArithmeticOperator", Comment = "Операция вычитания" },

new TerminalSymbol { Index = 12, Symbol = ">", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'больше'" },

new TerminalSymbol { Index = 13, Symbol = ">=", Category = "Специальный символ", Type = "RelationalOperator", Comment = "Операция сравнения 'больше или равно'" }

};

}

public static void PrintTable()

{

var table = GetTable();

Console.WriteLine("{0,-5} {1,-10} {2,-20} {3,-25} {4,-40}", "Индекс", "Символ", "Категория", "Тип", "Комментарий");

foreach (var symbol in table)

{

Console.WriteLine("{0,-5} {1,-10} {2,-20} {3,-25} {4,-40}",

symbol.Index, symbol.Symbol, symbol.Category, symbol.Type, symbol.Comment);

}

}

}

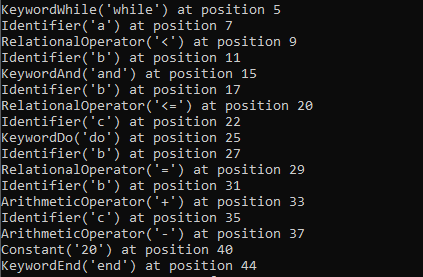
}

# Контрольный пример и результаты тестирования.

Пример:

while a < b and b <= c do b = b + c - 20 end

Результат работы программы:



# Контрольные вопросы.

**1) Назовите основные этапы компиляции.**

**Лексический анализ (Lexical Analysis)**

* Этот этап заключается в разбиении исходного кода на минимальные значимые единицы — токены. Компилятор читает исходный код построчно, определяя ключевые слова, идентификаторы, операторы, константы и другие элементы программы. Лексический анализатор игнорирует незначащие элементы, такие как пробелы и комментарии.
* Например, выражение int x = 42; будет разбито на следующие токены: int (тип), x (идентификатор), = (оператор присваивания), 42 (константа), ; (конец выражения).

**Синтаксический анализ (Syntax Analysis)**

* На этом этапе происходит проверка правильности структуры кода в соответствии с синтаксическими правилами языка программирования. Компилятор создает синтаксическое дерево (или дерево разбора), которое отображает иерархическую структуру программы.
* Цель синтаксического анализа — убедиться, что программа состоит из правильных грамматических конструкций. Например, если компилятор встречает выражение int x = 42 42;, он обнаружит ошибку, так как это выражение нарушает синтаксис языка (ожидается один операнд после оператора =).

**Семантический анализ (Semantic Analysis)**

* Этот этап проверяет семантическую корректность программы, т.е. соответствие ее логики правилам языка. Например, компилятор проверяет типы данных, совместимость типов, объявление переменных перед их использованием и корректность операций над данными.
* Пример: если вы попытаетесь присвоить строку числовой переменной (например, int x = "hello";), на этапе семантического анализа будет обнаружена ошибка типа данных.

**Генерация промежуточного кода (Intermediate Code Generation)**

* На этом этапе компилятор преобразует синтаксическое дерево в промежуточное представление, которое находится между исходным кодом и объектным кодом. Это представление может быть проще для анализа и оптимизации, и часто оно независимое от платформы.
* Пример промежуточного кода может выглядеть как набор простых инструкций или трехадресный код, где каждая инструкция выполняет одно элементарное действие.

**Оптимизация промежуточного кода (Intermediate Code Optimization)**

* На этом этапе компилятор пытается улучшить промежуточный код, уменьшая количество операций, устраняя дублирование и делая программу более эффективной по времени и/или памяти. Оптимизации могут включать:
  + Удаление неиспользуемого кода.
  + Замена медленных операций быстрыми.
  + Преобразование циклов для улучшения производительности.
* Например, если компилятор видит выражение x = x + 0, оно может быть оптимизировано и удалено, так как эта операция не меняет значение переменной.

**Генерация объектного кода (Code Generation)**

* На этом этапе происходит преобразование оптимизированного промежуточного кода в объектный код, который зависит от конкретной архитектуры процессора. Этот код будет исполняться на целевой машине. Результатом является машинный код или код, который может быть интерпретирован на уровне аппаратного обеспечения.
* На этом этапе компилятор также может добавить инструкции для управления памятью, вызова функций и других низкоуровневых операций.

**2)** **Дайте краткую характеристику этапу лексического анализа.**

* Лексический анализатор разбивает исходный код на последовательности символов (лексемы), которые соответствуют определенным токенам, и передает их на следующий этап обработки. Это первичная форма анализа исходного текста программы.

**3)** **Что такое лексема? Приведите примеры.**

* Лексема — это минимальная значимая единица исходного текста программы. Примеры: ключевые слова (например, if, for), идентификаторы (переменные, функции), операторы (например, +, =).

**4)** **С какими таблицами осуществляется работа на этапе лексического анализа?**

* Таблица символов (хранит информацию о переменных, функциях).
* Таблица ключевых слов.
* Таблица литералов (констант).
* Таблица токенов.

5) **Дайте определение грамматики.**

* Грамматика — это формальное описание синтаксиса языка программирования, определяющее правила формирования корректных последовательностей символов.

**6)** **Дайте определение автоматной грамматики.**

* Автоматная грамматика — это грамматика, которая описывает язык, распознаваемый конечным автоматом, и имеет ограничение на форму правил (например, регулярные грамматики).

**7)** **Назовите основные этапы создания анализатора по автоматной грамматике.**

* Определение правил грамматики.
* Построение конечного автомата.
* Определение состояний и переходов автомата.
* Разработка программы, распознающей язык, описанный грамматикой.

**8)** **Какие способы задания автоматного языка Вы знаете?**

**1. Регулярные выражения**

Описание: Регулярные выражения (regex) — это способ описания последовательностей символов, которые могут быть распознаны конечным автоматом. Они используются для описания шаблонов, по которым можно проверить, соответствует ли строка определенному образцу.

Пример: Регулярное выражение a\*b+ описывает строки, состоящие из любого количества символов a, за которыми следует хотя бы один символ b. Например, строки "aaab", "ab", "bbbb" будут соответствовать этому шаблону.

Применение: Регулярные выражения широко используются в системах поиска и обработки текста, а также в лексических анализаторах компиляторов.

**2. Контекстно-свободные грамматики (Context-Free Grammars, CFG)**

Описание: Контекстно-свободные грамматики — это набор правил, которые описывают структуру языка. В отличие от регулярных выражений, они способны описывать более сложные структуры, такие как вложенные выражения (например, скобки или блоки кода).

Пример: Правило S -> aSb | ε описывает язык, в котором каждая строка имеет равное количество символов a и b, например: "ab", "aabb", "aaabbb".

Применение: Контекстно-свободные грамматики используются для описания синтаксиса большинства языков программирования. Они важны на этапе синтаксического анализа при компиляции.

**3. Сети Петри (Petri Nets)**

Описание: Сети Петри — это математическая модель, используемая для описания систем, состоящих из взаимодействующих параллельных процессов. Они представляют собой графы, где узлы обозначают события (переходы) и состояния (позиции). Связи между ними задают возможные пути переходов между состояниями системы.

Пример: В сети Петри переход между состояниями может быть выполнен при выполнении определённых условий, что позволяет описывать сложные системы с параллельным выполнением процессов.

Применение: Сети Петри часто используются для моделирования систем с параллельными процессами, таких как системы управления и взаимодействия компонентов в распределенных системах.

**4. Автоматные диаграммы переходов (State Transition Diagrams)**

Описание: Диаграммы переходов состояний — это графическое представление автоматов, где узлы представляют состояния, а стрелки — переходы между состояниями, которые происходят в ответ на события или ввод символов.

Пример: Для описания простого конечного автомата можно использовать диаграмму, где начальное состояние переходит в другое состояние при определённом входном символе. Например, автомата, распознающего строку "ab", можно представить диаграммой с тремя состояниями (начальное, промежуточное и конечное).

Применение: Такие диаграммы широко применяются при проектировании систем автоматизации, интерфейсов и алгоритмов, а также в программировании для создания конечных автоматов.

**5. Формальные грамматики (Formal Grammars)**

Описание: Формальная грамматика — это более общая категория, которая включает регулярные и контекстно-свободные грамматики. Она состоит из множества правил, которые описывают последовательности символов для построения слов в языке. Различают несколько типов формальных грамматик (по классификации Хомского):

Тип 0: Неограниченные грамматики.

Тип 1: Контекстно-зависимые грамматики.

Тип 2: Контекстно-свободные грамматики.

Тип 3: Регулярные грамматики.

Пример: Контекстно-зависимая грамматика может описывать такие языки, в которых правила зависимы от контекста символов (например, множественное число в некоторых языках).

Применение: Формальные грамматики используются в компьютерной лингвистике, компиляторах и других системах анализа и синтеза языков.

**6. Pushdown Automata (Автоматы с магазинной памятью)**

Описание: Автоматы с магазинной памятью — это расширение конечных автоматов, которые могут использовать стек для хранения промежуточных данных. Это позволяет им распознавать контекстно-свободные языки, которые не могут быть распознаны обычными конечными автоматами.

Пример: Такие автоматы могут использоваться для распознавания вложенных скобок в выражениях, что недоступно для конечного автомата.

Применение: Автоматы с магазинной памятью широко используются для анализа синтаксиса программ и выражений, особенно когда требуется учитывать контексты и вложенные структуры.

**Дополнительные способы:**

**Turing Machines (Машины Тьюринга):** Описание языков, которые можно вычислить при помощи машин Тьюринга, что делает этот способ самым мощным, но менее применимым для повседневных задач синтаксического анализа. Машина Тьюринга может моделировать любые вычислимые функции, но её моделирование сложно для практического использования в описании языков.

**Графы переходов (Transition Graphs):** Могут быть использованы для задания автоматов и описания всех возможных переходов между состояниями.

**9)** **Дайте определение конечного автомата.**

* Конечный автомат — это математическая модель, представляющая собой устройство, которое находится в одном из конечного числа состояний и переходит из одного состояния в другое в зависимости от входных данных.

**10)** **Что такое детерминированная, недетерминированная и полнодетерминированная формы автоматных грамматик? Приведите примеры.**

* **Детерминированная грамматика** описывает автомат, в котором для каждого состояния и символа входа существует единственный возможный переход. Пример: ДКА.
* **Недетерминированная грамматика** допускает наличие нескольких переходов для одного состояния и символа. Пример: НКА.
* **Полнодетерминированная грамматика:**

Это расширение понятия детерминированной грамматики. Здесь под "вполне" подразумевается, что автомат имеет определённый переход для каждого возможного символа алфавита из каждого состояния.

Вполне детерминированная грамматика означает, что:

Переходы определены для каждого символа алфавита.

Нет ситуаций, когда для какого-либо символа алфавита переход не определён (это означает, что в грамматике нет "неопределённых состояний").

Эпсилон-переходы также исключены.

То есть, в таком автомате для каждого входного символа и каждого состояния всегда существует ровно один переход.

Пример: Если для каждого состояния автомата при любом символе из входного алфавита существует чётко определённый переход в другое состояние — это и есть вполне детерминированная граммат

**11)** **Дайте определение детерминированному (ДКА) и недетерминированному конечному автомату (НКА).**

* ДКА — это автомат, в котором для каждого состояния и символа входа определён ровно один возможный переход в другое состояние.
* НКА — это автомат, где возможны несколько переходов для одного состояния и символа входа, или же не определено ни одного перехода.

**12)** **В чем заключается алгоритм перехода от НКА к ДКА. Приведите пример.**

* Алгоритм включает создание множества состояний ДКА, каждое из которых является подмножеством состояний НКА. По каждому входному символу для каждого состояния НКА определяется набор состояний, и они группируются в новые состояния ДКА. В результате создается ДКА с эквивалентной функциональностью.

**13)** **Как перейти от грамматики в детерминированной форме к грамматике во вполне детерминированной форме?**

* Для перехода необходимо дополнить грамматику так, чтобы для каждого символа входа в любом состоянии был определён переход. Это может быть сделано добавлением "ловушечного" состояния (состояние, из которого невозможно выйти) для неразрешённых переходов.

**14)** **Как по вполне детерминированной автоматной грамматике составить программу анализа?**

* Для создания программы анализа по вполне детерминированной грамматике можно использовать таблицу переходов автомата, где каждая строка соответствует состоянию, а каждая колонка — входному символу. Программа последовательно обрабатывает входные символы, переходя по состояниям на основе этой таблицы.