МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

Высшего профессионального образования

«Чувашский государственный университет имени И. Н. Ульянова»

Факультет информатики и вычислительной техники

Кафедра вычислительной техники

**Курсовая работа**

**Объектно-Ориентированное программирование**

**Лексический анализатор языка Си.**

Выполнил: Андреев Д. A.

студент группы ИВТ-41-21

Проверил:

доц. к.т.н. Обломов И.А.

Чебоксары

2023

Оглавление

[1. ВВЕДЕНИЕ 4](#_Toc136001209)

[2. ПОСТАНОВКА ЗАДАЧИ 5](#_Toc136001210)

[3. ВЫБОР МЕТОДА, СПОСОБА РЕШЕНИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ И ОБОСНОВАНИЕ ВЫБОРА 6](#_Toc136001211)

[4. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ 9](#_Toc136001212)

[Структуры данных для представления объектов 10](#_Toc136001213)

[Алгоритмы работы ключевых частей программы. 21](#_Toc136001214)

[Иерархия лексем: 23](#_Toc136001215)

[Иерархия исключений: 24](#_Toc136001216)

[Иерархия обработчиков регулярных выражений: 24](#_Toc136001217)

[Описание работы программы 24](#_Toc136001218)

[5. Выводы по работе 26](#_Toc136001219)

[6. Выводы по работе 26](#_Toc136001220)

# ВВЕДЕНИЕ

Лексический анализатор – это программа, основной задачей которой является разбиение входного текста, состоящего из последовательности символов, на последовательность слов или лексем. Он часто применяется на первой фазе трансляции языков программирования. На этом этапе входные данные представляют собой цепочку символов, которые на выходе преобразуются в поток лексем, разбираемый синтаксическим анализатором.

Кроме трансляции языков программирования, лексический анализатор также широко используется в средах разработки, где он помогает автоматически выделять ключевые слова, идентификаторы и другие элементы кода, что значительно ускоряет процесс написания программ. Благодаря лексическому анализатору, разработчики могут быстро находить ошибки в коде и устранять их, что повышает качество и эффективность работы.

Лексический анализатор может быть как самостоятельной программой результатами работы которой являются полностью построенная таблицы лексем, имен и констант, так и подпрограммой, вызываемой синтаксическим анализатором для получения ещё одной лексемы. Во втором случае ответственность за построение таблиц ложиться на следующую стадию трансляции.

Обычно все лексемы разбиваются на определенные классы, такие как числа разных типов, идентификаторы и строки, также выделяются ключевые слова и символы пунктуации. Ключевые слова являются частью идентификаторов. В некоторых языках, например, с++, значение лексемы может зависеть от контекста, поэтому лексический анализ не может быть выполнен независимо от синтаксического анализатора.

При построении лексического анализатора важно учитывать дальнейшие фазы компиляции. Для синтаксического анализатора, работающего после лексического, важна информация о последовательности классов лексем, ограничителей и ключевых слов. Для контекстного анализа, работающего вслед за синтаксическим, нужна информация о конкретных значениях отдельных лексем.

# ПОСТАНОВКА ЗАДАЧИ

Разработать модель лексического анализатора на языке высокого уровня С++ для аналитического разбора входной последовательности символов на лексемы, используя принципы объектно-ориентированного программирования, а также исследовать различные методы реализации таких сканеров.

Лексический анализатор должен выполнять следующие основные функции:

1. Распознавание лексем во входной строке и определение их класса.
2. Формирование таблицы имён (идентификаторов).
3. Поиск и вывод лексических ошибок, таких как неверный символ или повторное определение метки.
4. Простейший синтаксический контроль, например, проверка парности операторных скобок и наличия ключевых слов.

Возможности программы:

1. Чтение из фала входной последовательности символов.
2. Разбиение последовательности символов на лексемы.
3. Определение корректности полученных лексем.
4. Вывод списка лексем и таблицы имен в случае успешного сканирования, иначе вывод соответствующей ошибки.

# ВЫБОР МЕТОДА, СПОСОБА РЕШЕНИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ И ОБОСНОВАНИЕ ВЫБОРА

Для решения задачи построения лексического анализатора широко используются три варианта реализации, каждый из которых имеет свои преимущества и недостатки.

**Способ 1.** Использование конечного автомата.

Конечный автомат - это математическая модель, которая используется для описания поведения систем, которые могут находиться в одном из конечного числа состояний и могут переходить из одного состояния в другое в зависимости от входных данных. Конечный автомат может быть использован для моделирования различных систем, таких как компьютерные программы, электрические цепи, логические схемы и т.д. Конечные автоматы широко применяются в теории автоматов и теории формальных языков.

Преимущества построения лексического анализатора на основе конечного автомата:

1. Простота реализации. Конечный автомат - это простая и интуитивно понятная модель, которую легко реализовать в коде.
2. Высокая скорость работы. Конечный автомат может обрабатывать входные данные очень быстро, так как он работает посимвольно и не требует дополнительных вычислений.
3. Надежность. Конечный автомат является формальной моделью, что означает, что его поведение можно точно определить и проверить.
4. Гибкость: Конечный автомат позволяет легко настраивать правила для анализа входных данных.
5. Универсальность: Конечные автоматы могут использоваться для анализа различных типов данных, таких как текст, числа, символы и т.д.

Недостатки построения лексического анализатора на основе конечного автомата:

1. Сложность: Конечные автоматы сложны в понимании и использовании.
2. Сложность реализации: Реализация конечного автомата может быть сложной и требует большого количества кода.
3. Сложность поддержки. Если требуется изменить лексический анализатор, основанный на конечном автомате, то это может потребовать значительных усилий, так как изменения могут затронуть множество состояний и переходов.

**Способ 2.** Использование контекстно свободной грамматики.

Контекстно-свободная грамматика является одним из основных инструментов для анализа и обработки текста. Она позволяет определить, является ли данная последовательность символов правильной или нет, и может использоваться для различных задач, таких как лексический анализ, синтаксический анализ, распознавание шаблонов и т.д.

Плюсы использования контекстно свободной грамматики для лексического анализатора:

1. Эффективность: контекстно-свободные грамматики позволяют быстро и эффективно обрабатывать большие объемы текста, так как они имеют низкую вычислительную сложность.
2. Гибкость: контекстно-свободные грамматики предоставляют возможность использовать различные структуры данных, такие как деревья, стеки, очереди и т.д., что позволяет адаптировать их к различным задачам.

Минусы использования контекстно-свободной грамматики:

1. Ограничения: контекстно-свободная грамматика неспособна обрабатывать контекст, а также неспособна работать с переменными длинами слов. Эти ограничения могут привести к ошибкам в анализе текста.
2. Сложность реализации: контекстно-свободная грамматика требует более высокого уровня знаний и навыков программирования, чем другие типы грамматик.

**Способ 3**

Регулярные выражения - это специальные символы и шаблоны, которые используются для поиска и сопоставления строк в тексте. Они позволяют находить определенные слова, фразы, числа или другие характеристики в тексте, что может быть полезно, например, при обработке естественного языка или при работе с базами данных. С помощью регулярных выражений можно найти все строки, содержащие определенное слово или фразу, или же найти все числа в заданном диапазоне. Регулярные выражения являются мощным инструментом для работы с текстом и широко используются в различных областях, таких как программирование, анализ данных, веб-разработка и другие.

Преимущества построения лексического анализатора на основе регулярных выражений:

1. Гибкость: Регулярные выражения позволяют создавать сложные и гибкие правила для обработки текста, что может быть полезно при обработке естественного языка или других сложных задач.
2. Эффективность: Регулярные выражения могут быть очень эффективными при поиске и замене текста, особенно если используются оптимальные алгоритмы работы с ними.
3. Простота реализации: Регулярные выражения легко изучать и использовать, что делает их популярным инструментом для работы с текстом.

Минусы:

1. Ограниченность: Регулярные выражения не могут обрабатывать все типы текста и не всегда могут быть использованы для решения сложных задач.

Для реализации лексического анализатора был выбран 3 метод, на основе регулярных выражений, за простоту реализации и легкость представления в виде кода С++.

# ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

**Описание лексического анализатора, построенного на использовании регулярных выражений.**

Для моделирования регулярных выражений был создан абстрактный класс, обязывающий наследника реализовать метод для выделения строки с закрепленным за ним регулярным выражением. Регулярные выражения соответствуют разным классам лексем, таким как числа разных типов, идентификаторы и строки, ключевые слова и символы пунктуации.

Общий вид регулярного выражения для строки - /".\*?"/, где двойные кавычки обозначают начало и конец строки, а .\*? означает любые символы между ними (включая пустую строку). Однако так же надо учитывать возможность экранирования двойных кавычек внутри строки.

Регулярное выражение для обозначения символов имеет аналогичный регулярным выражениям строки вид, за исключением того что вместо двойных кавычек используются одинарные - /’.\*?’/.

Операторы могут соответствовать следующим последовательностям символов: "+", "-", "\*", "/", "%", "=" , "==", "!=" , "<" ">", "<=" , ">=". И пунктуаторы соответствуют: "[", "]", "(", ")", "{", "}", "#", ".", ",", ";", "::",":".

С числами все немного сложнее. Числа могут быть представлены как целой записью, к примеру, 12, так десятичной записью 12.5. Помимо этого десятичные числа могут быть представлены в шестнадцатеричной системе счисления: 0xC, восьмеричной: 014 и бинарной: 0b1100.

Ключевое слово - это зарезервированное слово в языке программирования, которое имеет специальное значение и используется для определенных целей, например, для объявления переменных, функций, циклов и т.д. Ключевые слова не могут использоваться в качестве идентификаторов. Ключевым словом может быть одна из следующих последовательностей символов:

"auto"

"break"

"case"

"char"

"const"

"continue"

"default"

"do"

"double"

"else"

"enum"

"extern"

"float"

"for"

"goto"

"if"

"inline"

"int"

"long"

"register"

"restrict"

"return"

"short"

"signed"

"sizeof"

"static"

"struct"

"switch"

"typedef"

"union"

"unsigned"

"void"

"volatile"

"while"

"\_Alignas"

"\_Alignof"

"\_Atomic"

"\_Bool1"

"\_Complex"

"\_Generic"

"\_Imaginary"

"\_Noreturn"

"\_Static\_assert"

"\_Thread\_local"

Идентификатор - это имя, которое используется для обозначения переменных, функций, классов, модулей и других элементов программы. Идентификаторы могут состоять из букв, цифр и символа подчеркивания, но должны начинаться с буквы или символа подчеркивания. Идентификаторы чувствительны к регистру, то есть идентификаторы "foo" и "Foo" считаются разными. Первый символ идентификатора должен принадлежать к английскому алфавиту либо являться нижним подчеркиванием. Последующие символы могут быть как символами английского алфавита, так цифрами или символом нижнего подчеркивания.

Первый шаг лексического анализатора - это удаление идущих перед кодом пробельных символов, таких как пробелы, символы табуляции и переводы строк. Это позволяет проигнорировать ненужные символы и упростить дальнейшую обработку входной строки.

Затем лексический анализатор проверяет входную строку на наличие комментариев. Комментарии - это часть кода, которая игнорируется компилятором или интерпретатором и используется только для пояснения кода. Комментарии могут быть однострочными (начинаются с символа // и продолжаются до конца строки) или многострочными (начинаются с символов /\* и заканчиваются символами \*/).

После удаления пробельных символов и комментариев лексический анализатор последовательно проверяет входную строку на соответствие регулярным выражениям, пытаясь выделить последовательность символов соответствующую регулярному выражению. В случае отсутствия такой последовательности лексический анализатор возвращает последовательность символов не являющиеся частью алфавита.

## Структуры данных для представления объектов

**Класс Pointer:**

class Pointer

{

private:

const char\* \_headPtr;

const char\* \_currentPtr;

size\_t \_length;

public:

Pointer(const char\* headPtr, size\_t length);

size\_t GetLength();

size\_t GetRemainingLength();

size\_t GetPassedLength();

bool IsReachedEnd();

char GetCurrent();

const char\* GetHeadPtr();

const char\* GetCurrentPtr();

bool CompareFollowing(std::string string);

bool MoveNext();

bool MoveNext(size\_t length);

bool MovePrevious();

bool MovePrevious(size\_t length);

};

*Описание:* класс предназначен для упрощения работы со входными данными, предоставляет методы для индексирования по входным данными.

*Методы:*

size\_t GetLength(); - метод возвращает длину входной последовательности символов.

size\_t GetRemainingLength(); - метод возвращает не пройденную длину последовательности символов.

size\_t GetPassedLength(); - метод возвращает пройденную длину последовательности символов.

bool IsReachedEnd(); - метод возвращает true, если входная последовательность символов было пройдена до конца.

bool GetCurrent(); - возвращает символ на который указывает текущий символ, на котором остановился указатель

const char\* GetHeadPtr(); - возвращает указатель на начало входной. последовательности символов.

const char\* GetCurrentPtr(); - возвращает указатель на текущий символ.

bool CompareFollowing(std::string string); - сравнивает аргумент string с последующими символами.

bool MoveNext(); - увеличивает указатель на текущим символ на 1, в случае успеха возвращает true, если достигли конца то false.

bool MoveNext(size\_t length); - увеличивает указатель на текущий символ на length.

bool MovePrevious(); - уменьшает указатель на текущим символ на 1, в случае успеха возвращает true, если достигли начала то false.

bool MovePrevious(size\_t length); - уменьшает указатель на текущим символ на length.

**Класс Analyzer:**

class Analyzer

{

private:

NamesTabel\* \_namesTabel;

std::list<Token\*> \_tokens;

Lexer\* \_lexer;

BracketСounter\* \_brackets;

public:

Analyzer(const char\* input, size\_t length);

void Parse();

NamesTabel\* GetNamesTabel();

std::list<Token\*> GetTokens();

};

*Описание:* класс предназначен для построения таблицы имен и списка лексем.

*Методы:*

void Parse(); - Разбивает входную последовательность символов на лексемы и строит таблицу имён.

NamesTabel\* GetNamesTabel(); - возвращает таблицу имён.

std::list<Token\*> GetTokens(); - возвращает список токенов.

**Класс Lexer:**

class Lexer

{

private:

Pointer\* \_pointer;

std::vector<BaseTokenRegex\*> \_parsers;

public:

Lexer(std::string string);

Lexer(const char\*, size\_t length);

private:

bool TrySkipSpaces();

bool TrySkipComments();

public:

bool TryLex(Token\*&);

};

*Описание:* класс предназначен для распознавания лексем.

*Методы:*

bool TryLex(Token\*&); - Возвращает следующий распознанный токе, в случае успешного сканирования токена вернется true иначе false

**Класс BaseException:**

class BaseException

{

private:

std::string \_message;

public:

BaseException(std::string message);

std::string GetMessage();

};

*Описание:* Родительский класс для всех исключений возникающих при лексическом анализе.

*Методы:*

std::string GetMessage(); - Вернет сообщение исключения. В сообщении должна содержаться основная информация об ошибке

**Класс IncorrectBracketSequencesException:**

class IncorrectBracketSequencesException: public BaseException

{

public:

IncorrectBracketSequencesException();

};

*Описание:* Исключение возникает при обнаружении неправильной последовательности скобок.

**Класс InvalidTokenException:**

class InvalidTokenException : public BaseException

{

private:

InvalidToken\* \_token;

public:

InvalidTokenException(InvalidToken\* token);

InvalidToken\* GetInvalidToken();

};

*Описание:* Исключение возникает при обнаружении недопустимой последовательности символов.

*Методы:*

InvalidToken\* GetInvalidToken(); - Возвращает лексему из за которой было вызвано исключение.

**Класс** **MultipleInitializationException:**

class MultipleInitializationException : public BaseException

{

private:

Identifier\* \_identifier;

public:

MultipleInitializationException(Identifier\* identifier);

Identifier\* GetIdentifier();

};

*Описание:* Исключение возникает при попытке множественной инициализации.

**Класс NamesTabel:**

class NamesTabel

{

private:

Scope\* \_global;

Scope\* \_current;

public:

NamesTabel();

Scope\* GetGlobalScope();

void LinkToken(Identifier\* identifier);

void AddIdentifier(Identifier\* identifier, IdentifierInfo\* info);

int GetCurrentInverstmentLevel();

void UpScope();

void DownScope();

};

*Описание:* Предоставляет методы для формирования и работы с таблицей имен. Таблица имен представляет древовидную структуру.

*Методы:*

Scope\* GetGlobalScope(); - Возвращает глобальную область видимости

void LinkToken(Identifier\* identifier); - Связывает идентификатор с его полем в таблице имен

void AddIdentifier(Identifier\* identifier, IdentifierInfo\* info); - Добавляет идентификатор в таблицу имен, в текущую область видимости;

int GetCurrentInverstmentLevel(); - Возвращает текущий уровень вложенности.

void UpScope(); - Повышает уровень вложенности.

void DownScope(); - Понижает уровень вложенности.

**Класс Scope:**

class Scope

{

private:

Scope\* \_parent;

std::map<std::string, IdentifierInfo\*> \_identifiers;

std::list<Scope\*> \_childs;

int \_investmentLevel;

public:

Scope();

Scope(Scope\* parent);

int GetInverstmentLevel();

std::list<Scope\*> GetChilds();

Scope\* CreateChildScope();

Scope\* GetParentScope();

void AddIdentifier(Identifier\*, IdentifierInfo\*);

std::map<std::string, IdentifierInfo\*> GetIdentifiers();

};

*Описание:* Позволяет создать древовидную структуру таблицы имен.

*Методы:*

int GetInverstmentLevel(); - Возвращает уровень вложенности текущей области видимости.

std::list<Scope\*> GetChilds(); - Возвращает список вложенных областей видимости.

Scope\* CreateChildScope(); - Добавляет новую вложенную область видимости.

Scope\* GetParentScope(); - Возвращает область видимости, в которую вложена текущая область видимости.

void AddIdentifier(Identifier\*, IdentifierInfo\*); - добавляет идентификатор в текущую область видимости.

**Класс IdentifierInfo:**

class IdentifierInfo

{

private:

IdentifierType \_type;

Token\* \_typeToken;

public:

IdentifierInfo(Token\* typeToken, IdentifierType type);

IdentifierType GetType();

std::string ToString();

};

*Описание:* Информация об идентификаторе хранящаяся в таблице имен.

*Методы:*

IdentifierType GetType(); - вернет тип идентификатора, тип может быть функцией, структурой, переменной и неизвестный.

std::string ToString(); - вернет строковое представление этого класса.

**Класс BaseTokenRegex:**

class BaseTokenRegex abstract

{

protected:

Pointer\* \_pointer;

public:

BaseTokenRegex(Pointer\* pointer);

virtual bool TryParse(Token\*& token) abstract;

};

*Описание:* Представляет базовый класс для всех регулярных выражений.

*Методы:*

virtual bool TryParse(Token\*& token) abstract; - Если входная последовательность символов совпадает с регулярным выражением, то метод вернет true и проинициализирует значение параметра token, соответствующей лексемой.

*Классы наследники:*

CharConstantRegex, NumericConstantRegex, KeywordOrIdentifierRegex, LiteralRegex, OperatorRegex.

**Класс Token:**

class Token

{

private:

TokenType \_TokenType;

const char\* \_startPtr;

size\_t \_length;

public:

Token(TokenType id, const char\* startPtr, size\_t length);

size\_t GetLength();

const char\* GetStartPtr();

TokenType GetTokenType();

std::string GetOriginalString();

virtual std::string ToString() abstract;

};

*Описание:* Хранит лексему.

*Методы:*

size\_t GetLength(); - вернет длину лексемы

const char\* GetStartPtr(); - вернет указатель на первый символ лексемы

TokenType GetTokenType(); - вернет тип токена.

std::string GetOriginalString(); - вернет строковое представление лексемы.

virtual std::string ToString() abstract; - строковое представление токена (для вывода в консоль).

**Класс InvalidToken:**

class InvalidToken: public Token

{

std::string \_reason;

public:

InvalidToken(const char\* startPtr, size\_t length);

InvalidToken(const char\* startPtr, size\_t length, std::string reason);

std::string ToString() override;

};

*Описание:* Представляет неверную лексему.

**Класс Punctuator:**

class Punctuator : public Token

{

private:

int \_punctuatorId;

std::string \_value;

public:

Punctuator(int id, const char\* startPtr, size\_t length);

std::string ToString() override;

bool IsBracket();

bool IsOpenBracket();

bool IsCloseBracket();

int GetPunctuatorId();

};

*Описание:* Хранит одну из следующих лексем: "[", "]", "(", ")", "{", "}", "#", ".", ",", ";", "::", ":".

**Класс Operator:**

class Operator : public Token

{

private:

int \_operatorId;

public:

Operator(int OpeartorId, const char\* startPtr, size\_t length);

std::string ToString() override;

};

*Описание:* Хранит одну из следующих лексем: "...", ">>=", "<<=","::", "==", "%>", "&&", "||", "\*=", "/=", "%=", "-=", ">=", "&=", "^=", "|=", "##", "<:", ":>", "<%", "+=", "<=", "!=", ">>", "<<", "->", "++", "--", "<", "&", "\*", "+", "-", "~", "=", "!", "?", "/", "%", "|", "^", ">"

**Класс Literal:**

class Literal: public Token

{

public:

Literal(const char\* startPtr, size\_t length);

std::string ToString() override;

};

*Описание:* Хранит литералу.

**Класс Keyword.**

class Keyword : public Token

{

private:

int \_keywordId;

public:

Keyword(int id, const char\* startPtr, size\_t length);

int GetKeywordId();

bool IsType();

std::string ToString() override;

};

*Описание:* Хранит ключевое слово.

**Класс Identifier.**

class Identifier : public Token

{

private:

IdentifierInfo\* \_info;

public:

Identifier(const char\* startPtr, size\_t length);

bool IsType();

void LinkToNamesTabel(IdentifierInfo\* info);

std::string ToString() override;

};

*Описание:* Хранит идентификатор.

**Класс** **Constant .**

class Constant :public Token

{

public:

Constant(const char\* startPtr, size\_t length);

std::string ToString() override;

};

*Описание:* Базовый класс для всех констант.

**Класс IntegerConstant.**

class IntegerConstant : public Constant

{

private:

NumberSystem \_numberSystem;

public:

IntegerConstant(const char\* startPtr, size\_t length, NumberSystem numberSystem);

std::string ToString() override;

};

*Описание:* Хранит целочисленную константу.

**Класс FloatConstant.**

class FloatConstant: public Constant

{

public:

FloatConstant(const char\* startPtr, size\_t length);

std::string ToString() override;

};

*Описание:* Хранит константу с плавающей точкой.

**Класс CharConstant.**

class CharConstant: public Constant

{

public:

CharConstant(const char\* startPtr, size\_t length);

std::string ToString() override;

};

*Описание:* Хранит символьную константу.

**Класс BracketСounter.**

class BracketСounter

{

private:

std::list<Punctuator\*> \_brackets;

public:

BracketСounter();

void Add(Punctuator\* bracket);

size\_t GetDepth();

};

*Описание:* Класс подсчитывает скобки.

*Методы:*

void Add(Punctuator\* bracket); - Учитывает новую скобку, вызывает исключение в случае обнаружения неправильной последовательности скобок.

size\_t GetDepth(); - возвращает текущий уровень вложенности на основе количества фигурных скобок.

## Алгоритмы работы ключевых частей программы.

*Алгоритм разбиения входной последовательности на лексемы (метод Lexer::Lex()).*

bool Lexer::TryLex(Token\*& token, bool catchInvalidToken)

{

while (true)

{

//Подгоняем к строке с кодом убирая табы и прочее

//Пропускаем комментарии

if (TrySkipSpaces() && TrySkipComments())

continue;

//Пытаемся запарсить различные виды лексем

for (BaseTokenRegex\* parser : \_parsers)

{

if(parser->TryParse(token))

return true;

}

//Если пройдя через все фильтры, мы не нашли лексему, значит лексема неверна

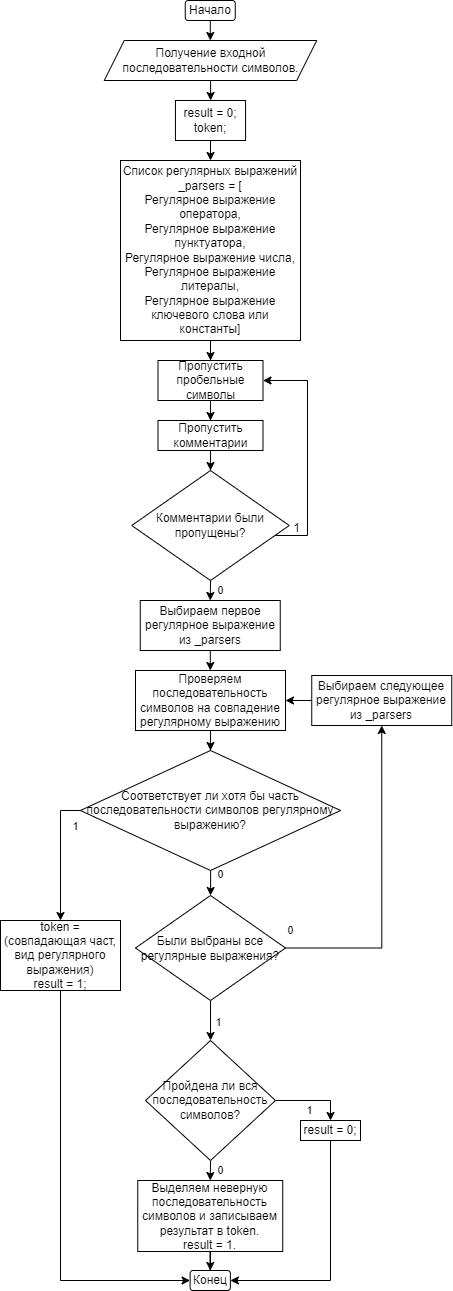
if (catchInvalidToken && TryParseInvalidToken(token))

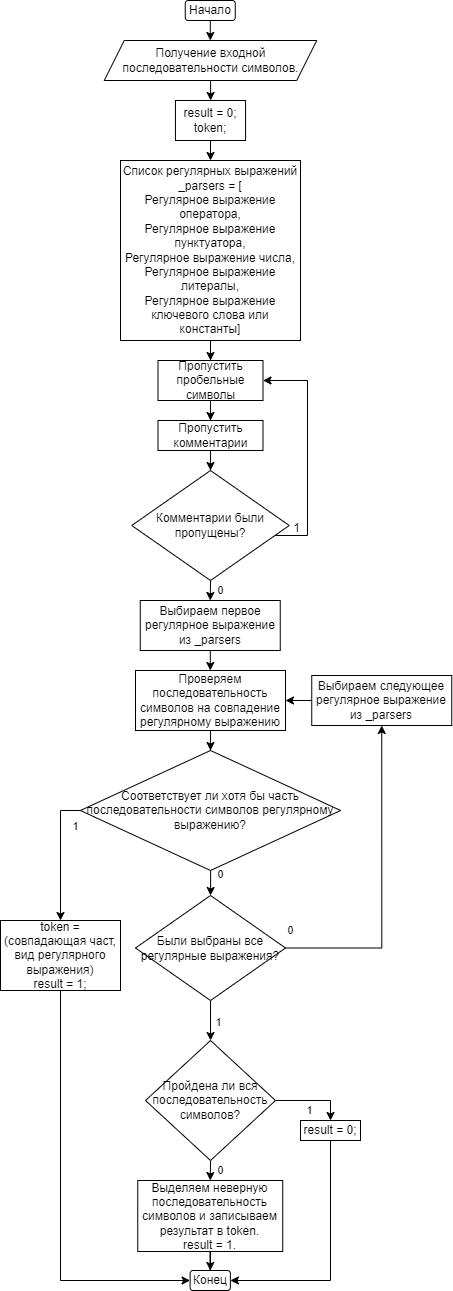
return true;

return false;

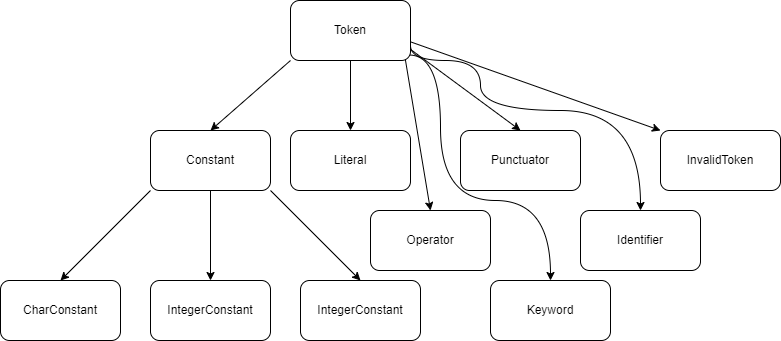
}

}



****

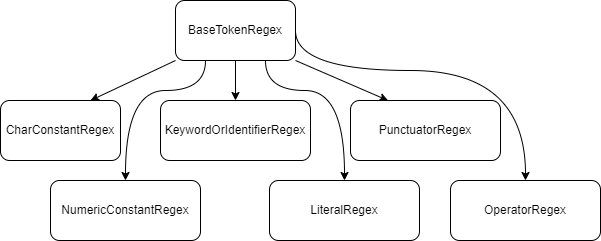
## Иерархия лексем:



## Иерархия исключений:

****

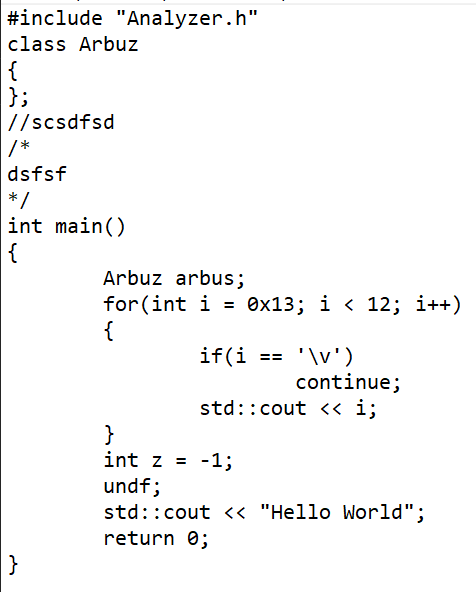
## Иерархия обработчиков регулярных выражений:



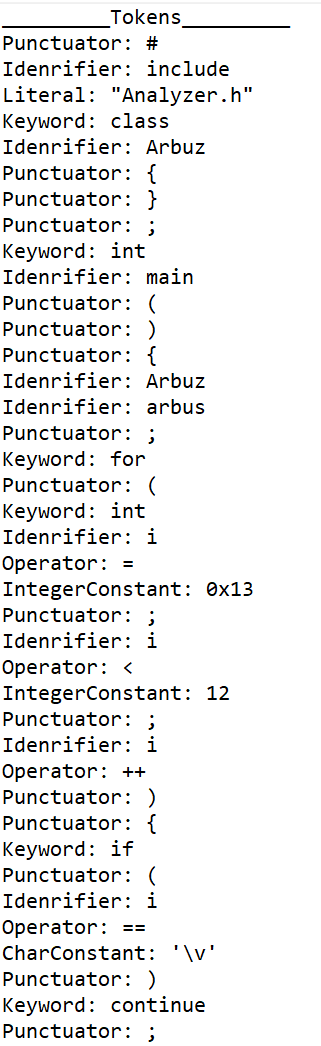
## Описание работы программы

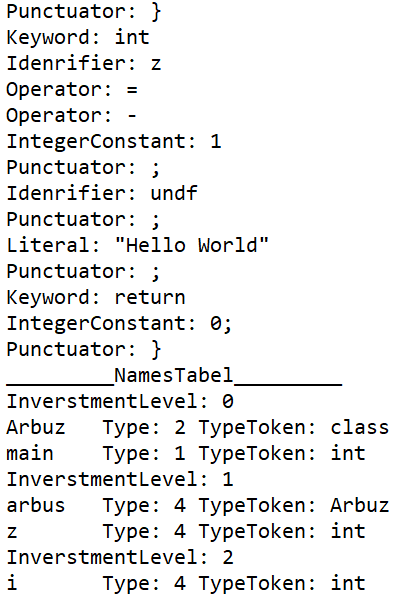
В файл input.txt вводится тестовая последовательность символов. При запуске программа считывает содержимое файла и выводи в консоль список лексем и таблицу имен.

Содержимое файла input.txt



Вывод программы:





# Выводы по работе.

В ходе выполнения работы был реализован лексический анализатор на основе регулярных выражений, построенный с использованием принципов объектно-ориентированного программирования.

Были рассмотрены различные методы построения лексических анализаторов, в том числе на основе конечных автоматов, регулярных выражений и контекстно свободной грамматики. Было решено использовать регулярные выражения, так как они имеют более простую реализацию.

Была разработана программа на языке C++, которая считывает исходный код программы и выделяет из него лексемы с помощью регулярных выражений, а также строит таблицу имен.

В результате работы был получен функциональный лексический анализатор, который может быть использован для анализа исходного кода программ на языке СИ. Были выявлены некоторые проблемы при работе с регулярными выражениями, такие как ограниченность регулярных выражений в задаче построения таблицы имен.

В результате работы над данной курсовой работой, было получено глубокое понимание принципов объектно-ориентированного программирования и их применения при разработке лексических анализаторов.