### 

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ**

**Департамент программной инженерии и искусственного интеллекта**

**КУРСОВОЙ ПРОЕКТ**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА ADA В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА C++

по дисциплине «Теория языков программирования и компиляторы»

по образовательной программе подготовки бакалавров по направлению 09.03.04 Программная инженерия

Выполнили студенты

гр. Б9120-09.03.04прогин

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Воробьева К.В.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Кассина М.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Лобеева М.Д.

Руководитель:

ассистент департамента ПИиИИ

Глазырин М.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2024 г.

(подпись)

г. Владивосток

2024 г

**Оглавление**

[Введение 3](#_Toc157424617)

[1 Неформальная постановка задачи 5](#_Toc157424618)

[2 Синтаксис входного языка 5](#_Toc157424619)

[3 Контекстные условия языка Ada 7](#_Toc157424620)

[4 Контекстные условия языка C++ 8](#_Toc157424637)

[5 Таблица соответствия языков 9](#_Toc157424638)

[6 Проект лексического анализатора 11](#_Toc157424639)

[6.1 Модель данных 11](#_Toc157424647)

[6.2 Таблица ключевых слов 12](#_Toc157424648)

[6.3 Таблица зарезервированных имен 13](#_Toc157424649)

[6.4 Типы лексем 14](#_Toc157424650)

[6.5 Конечный автомат лексического анализатора 14](#_Toc157424651)

[7 Проект синтаксического анализатора 15](#_Toc157424652)

[7.1 Модель данных 15](#_Toc157424653)

[7.2 Описание алгоритма рекурсивного спуска синтаксического анализа 19](#_Toc157424654)

[7.3 Пример дерева разбора 20](#_Toc157424665)

[7.4 Типы ошибок 20](#_Toc157424667)

[8 Проект семантического анализатора 21](#_Toc157424668)

[8.1 Модель данных 21](#_Toc157424669)

[8.2 Методы проверки нарушений контекстных условия 22](#_Toc157424670)

[8.3 Типы ошибок 23](#_Toc157424671)

[9 Проект генератора кода 24](#_Toc157424672)

[9.1 Модель данных 24](#_Toc157424673)

[9.2 Метод кодогенерации 25](#_Toc157424674)

[9.3 Примеры кодогенерации 26](#_Toc157424675)

[10 Тестирование 27](#_Toc157424676)

[10.1 Лексический анализатор 27](#_Toc157424677)

[10.2 Синтаксический анализатор 29](#_Toc157424678)

[10.3 Семантический анализатор 31](#_Toc157424679)

[10.4 Генератор кода 35](#_Toc157424680)

[Заключение 37](#_Toc157424681)

[Список литературы 38](#_Toc157424682)

# Введение

В настоящее время существует множество разнообразных языков программирования. Их практическое использование невозможно без соответствующей системы программирования, основу которой составляет транслятор или компилятор.

Транслятор – это программа, которая переводит входную программу на исходном (входном) языке в эквивалентную ей выходную программу на результирующем (выходном) языке. Компилятор – это транслятор, который осуществляет перевод исходной программы в эквивалентную ей объектную программу на языке машинных команд или на языке ассемблера. Компилятор отличается от транслятора лишь тем, что его результирующая программа всегда должна быть написана на языке машинных кодов или на языке ассемблера.

Входным языком был выбран язык программирования «Ada», а выходным – язык программирования «C++».

Языки программирования – это искусственные языки. Они отличаются от естественных ограниченным, достаточно малым числом слов, значение которых понятно транслятору, и очень строгими правилами записи команд (*предложений*). Совокупность правил, определяющих допустимые конструкции (*слова, предложения*) языка, т.е. его форму, образуют *синтаксис* языка, а совокупность правил, определяющих смысл синтаксически корректных конструкций языка, т.е. его содержание, – *семантику* языка.

Нарушения формы записи конструкций языка программирования приводят к *синтаксическим* ошибкам, а ошибки, связанные с неправильным содержанием действий и использованием недопустимых значений величин, называются *семантическими*. *Логические* ошибки возникают, когда все конструкции языка записаны правильно, но последовательность их выполнения дает неверный результат.

Трансляторы могут способствовать использованию высокоуровневых языков программирования, минимизируя накладные расходы времени выполнения программ, написанных на этих языках. Они играют важную роль в эффективном использовании высокопроизводительной архитектуры компьютера пользовательскими приложениями.

**Цель курсового проекта:** разработать проект транслятора, который преобразует программу, содержащую подмножество языка Ada, в программу, содержащую подмножество языка C++ генерируя эквивалентный исходный код.

**Задачи:**

1. описать грамматику подмножества входного языка;
2. описать контекстные условия входного и выходного языков;
3. описать соответствие конструкций входного и выходного языков;
4. разработать проект лексического анализатора;
5. разработать проект синтаксического анализатора;
6. разработать проект семантического анализатора;
7. разработать проект генератора кода выходного языка;
8. реализовать транслятор входного языка в выходной;
9. выполнить тестирование разработанного транслятора.

# 1 Неформальная постановка задачи

Подмножество языка Ada включает в себя:

* + - 1. типы данных: integer, real, character, array, boolean, string
      2. операторы:
* арифметические: сложение, вычитание, умножение, деление, остаток от деления;
* логические: И (and), ИЛИ (or), НЕ (not);
* отношений: меньше, равенство, больше, меньше или равно, больше или равно, не равно;
* лево-присваивающие бинарные: присвоение, прибавление к левому операнду правого, вычитание из левого операнда правого, умножение к левому операнду правого, деление левого операнда на правый, остаток от деление левого операнда на правый;
  + - 1. операторы циклов: while
      2. операторы ветвления: if, end if, when, else
      3. стандартные функции: ввод, вывод
      4. операторы прерывания: return

# 2 Синтаксис входного языка

<program> ::= <statements> EOF | EOF

<statements> ::= <statement statements> | <statement>

<statement> ::= <root\_stmt> | <nested\_stmt>

<root\_stmt> ::= <function\_declaration> SEMICOLON | <procedure\_declaration> SEMICOLON

<function\_declaration> ::= FUNCTION ID LPR <formal\_params> RPR RETURN ID IS BEGIN <block> END ID | FUNCTION ID LPR RPR RETURN ID IS BEGIN <block> END ID

<procedure\_declaration> ::= PROCEDURE ID LPR <formal\_params> RPR IS BEGIN block END ID | PROCEDURE ID LPR RPR IS BEGIN <block> END ID

<formal\_params> ::= ID COLON ID SEMICOLON <formal\_params> | ID COLON ID

<block> ::= <nested\_stmt block> | <nested\_stmt>

<nested\_stmt> ::= <compound\_stmt> SEMICOLON | <simple\_stmt> SEMICOLON

<compound\_stmt> ::= <if\_stmt> | <for\_stmt> | < while\_stmt >

<if\_stmt> ::= IF <expression> THEN <block elsif\_stmt> END IF | IF <expression> THEN <block else\_block> END IF | IF <expression> THEN <block> END IF

<elsif\_stmt> ::= ELSIF <expression> THEN <block elsif\_stmt> | ELSIF <expression> THEN <block else\_block> | ELSIF <expression> THEN <block>

<else\_block> ::= ELSE <block>

<while\_stmt> ::= WHILE <expression> LOOP <block> END LOOP

<for\_stmt> ::= FOR ID IN (NUMBER | ID) DOUBLEDOT (NUMBER | ID) LOOP <block> END LOOP

<simple\_stmt> ::= <assignment> | <expression> | <return\_stmt>

<assignment>::= ID ASSIGN <expression>

<return\_stmt> ::= RETURN <expression>

<expression> ::= <disjunction>

<disjunction> ::= <conjunction> OR <disjunction> | <conjunction>

<conjunction>::= <inversion> AND <conjunction> | <inversion>

<inversion> ::= NOT <inversion> | <comparison>

<comparison> ::= <sum> GREATER <comparison> | <sum> LESS <comparison> | <sum> EQUAL <comparison> | <sum> NOTEQ <comparison> | <sum> GREQUAL <comparison> | <sum> LEQUAL <comparison>| <sum>

<sum> ::= <term> PLUS <sum> | <term> MINUS <sum> | <term>

<term> ::= <factor> STAR <term> | <factor> DIV <term>| <factor> MOD <term> | <factor>

<factor> ::= PLUS <factor> | MINUS <factor> | <primary>

<primary> ::= <atom primary> | <atom> | DOT ID <primary> | DOT ID | <func\_call>

<func\_call> ::= LPR <actual\_params> RPR | LPR RPR

<atom> ::= ID | STRING | NUMBER

<actual\_params>::= <expression> COMMA <actual\_params> | <expression>

# 3 Контекстные условия языка Ada

# Контекстные условия о правилах описания идентификаторов в программа:

# 1. Идентификаторы, используемые в программах, должны быть описаны до их использования в программе;

# 2. В идентификаторах не имеет значения регистр;

# 3. Идентификатор может быть описан несколько раз только в том случае, если все различные описания этого идентификатора имеют разные области действия (блок программы, модуль программы, подпрограммы и т.д.);

# 4. Идентификатор – последовательность латинских букв или цифр, начинающаяся с буквы. Нельзя начинать и заканчивать идентификатор знаком «\_» и нельзя использовать два знака «\_» подряд;

# 5. Идентификатор не может являться ключевым (зарезервированным) словом, который служит для оформления конструкций языка (ключевыми словами являются: and, array, as, begin, case, constant и др.), следовательно и контекстно-ключевым словом (стандартным директивом), так как контекстно-ключевые слова являются ключевыми в некотором контексте;

# 6. Идентификатор не может переноситься со строки на строку.

# Контекстные условия о правилах использования идентификаторов в своей области действия:

# 1. Идентификатор не может быть использован при его объявлении, кроме рекурсии;

# 2. Использующее вхождение идентификаторов может быть в любой конструкции.

# Контекстные условия, определяющие правила соответствия видов величин, входящих в синтаксические конструкции программ:

# 1. Нельзя передавать в метод или функцию количество параметров, не соответствующее объявленному количеству;

# 2. Запрещены неявные преобразования при использовании разных типов;

# 3. Использование и описание массивов (и других типов) должно соответствовать выделенной им области памяти.

# Контекстные условия, задающие различные количественные ограничения:

# 1. –

# 4 Контекстные условия языка C++

**Контекстные условия о правилах описания идентификаторов в программах:**

1. Идентификаторы, используемые в программах, должны быть описаны до их использования в программе;

2. Первым символом идентификатора может быть только буква, следующими символами идентификатора могут быть буквы, цифры и \_, длина идентификатора не ограничена;

3. В идентификаторах учитывается регистр;

4. Идентификатор не может быть ключевым словом. Ключевые слова зарезервированы. (Пример ключевых слов: const, else, int, return…);

5. Идентификатор не может переноситься с одной строки на другую.

**Контекстные условия о правилах использования идентификаторов в своей области действия:**

1. Идентификатор не может быть использован при его объявлении, кроме рекурсии;

2. Использующее вхождение идентификаторов может быть в любой конструкции.

**Контекстные условия, определяющие правила соответствия видов величин, входящих в синтаксические конструкции программ:**

1. Нельзя передавать в функцию количество параметров, не соответствующее объявленному;

типа) при использовании разных типов;

3. Использование массивов должно соответствовать выделенной им памяти;

**Контекстные условия, задающие различные количественные ограничения:**

1. –

# 5 Таблица соответствия языков

Таблица 1 – Соответствие Ada – C++

|  |  |  |
| --- | --- | --- |
|  | **Ada** | **C++** |
| Структура программы | <Объявление переменных>  <0-блок> | <Объявление переменных>  <0-блок> |
| Циклы | Стандартный цикл:  loop  <Действия>;  exit when <eсловие>  end loop;  Цикл while:  while <условие> loop  <действия>;  end loop;  Цикл for:  for i in BufferRange loop  <действия>;  end loop; | 1. Цикл while:   while (<условие>) {  <тело цикла>;  }   1. Цикл for:   for (<счетчик> = <значение>; <счетчик> < <значение>; <шаг цикла>) {  <тело цикла>;  }   1. Цикл do while:   do {  <тело цикла>;  } while (<условие>); |
| Ветвление | if <условие\_1> then  <действие\_1>;  elsif <условие\_2> then  <действие\_2>;  else  <действие\_3>;  end if; | If (<условие\_1>) {  <действие\_1>;  } else if (<условие\_2>) {  <действие\_2>;  } else {  <действие\_3>;  } |
| Функции | function Func  (Var : Integer) return Integer  is  begin  return Var + 1;  end Func; | int Func (int Var) {  return Var + 1;  } |
| Процедуры | procedure Proc  (Var1 : Integer;  Var2 : out Integer;  Var3 : in out Integer)  is  begin  Var2 := Func (Var1);  Var3 := Var3 + 1;  end Proc; | void Proc  (int Var1,  int & Var2,  int & Var3) {  Var2 = Func (Var1);  Var3 = Var3 + 1;  } |

# 6 Проект лексического анализатора

## Лексер (или лексический анализатор) в трансляторе - это компонент, который отвечает за разбор входной последовательности символов на лексемы, то есть на минимальные смысловые единицы программы, такие как ключевые слова, идентификаторы, операторы, числа, строки и так далее.

## Лексический анализатор работает по следующему принципу:

## 1. Сканирование: лексер сканирует входной поток символов по одному символу за раз, образуя лексемы.

## 2. Классификация: каждая лексема классифицируется на тип (например, ключевое слово, идентификатор, оператор и т.д.) с помощью определенных правил.

## 3. Генерация токенов: после классификации лексемы, лексер генерирует токены, представляющие собой пары "тип лексемы" и "значение" (например, для чисел - тип "число" и значение самого числа).

## Эти токены затем передаются синтаксическому анализатору для дальнейшей обработки.

## Цель лексического анализа - подготовить входные данные для синтаксического анализа, выделяя из потока символов лексемы с их значениями и типами.

## 6.1 Модель данных

Были реализованы классы **Token**, **Lexer**, а также перечислимый тип **LexerStates**.

**LexerStates** содержит типы для получаемых лексем. Это множество определяется на основе подмножества входного языка.

Класс **Token** – класс, определяющий пару вида «тип-лексема».

Класс **Lexer** – класс, позволяющий провести лексический анализ текста.

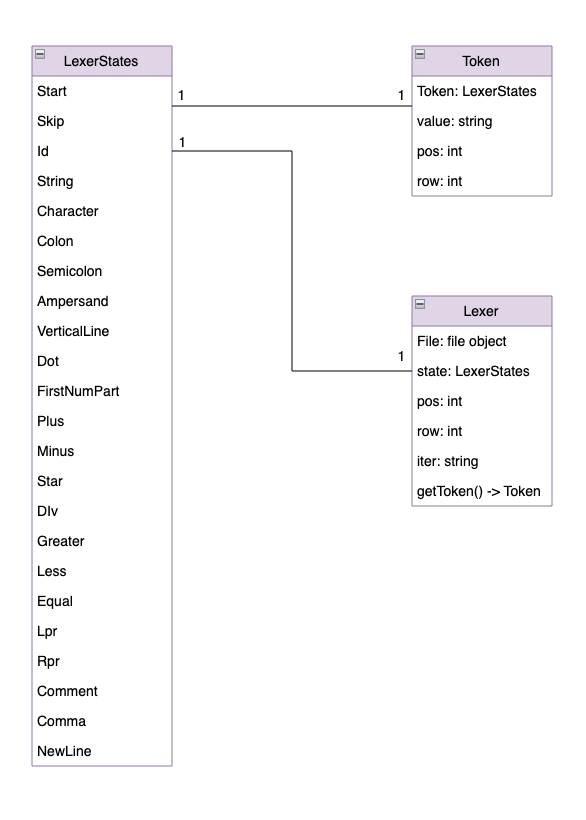


Рисунок 1 – Диаграмма классов для лексического анализатора

## 6.2 Таблица ключевых слов

В контексте языков программирования «ключевые слова» представляют собой предварительно определенные зарезервированные слова, имеющие специальное значение в рамках синтаксиса языка. Эти слова обычно не могут использоваться в качестве идентификаторов (имен переменных, функций, процедур и т. д.) и предназначены для выполнения специфических задач в программе. Ниже приведена таблица ключевых слов языка Ada.

Таблица 2 – Ключевые слова

| **Ключевые слова** | **Ключ** |
| --- | --- |
| begin | K1 |
| else | K2 |
| end | K3 |
| for | K4 |
| goto | K5 |
| if | K6 |
| loop | K7 |
| return | K8 |
| then | K9 |
| while | K10 |

## 6.3 Таблица зарезервированных имен

Зарезервированное слово в языках программирования – это слово, которое выделено для использования в качестве ключевого слова или определенной конструкции в синтаксисе языка, и поэтому не может быть использовано как идентификатор (название переменной, функции, класса и т.д.). Зарезервированные слова обеспечивают языку программирования набор ключевых конструкций, который определяет его синтаксическую структуру и функциональность. По сути, они предназначены для использования в определенных контекстах, и использование их вне этих контекстов приведет к ошибке компиляции или выполнения.

Таблица 3 – Зарезервированные слова

| **Зарезервированное имя** | **Значение** |
| --- | --- |
| array | R1 |
| and | R2 |
| case | R3 |
| else | R4 |
| is | R5 |
| in | R6 |
| function | R7 |
| or | R8 |

## 6.4 Типы лексем

В контексте программирования и компиляции, лексемы представляют собой минимальные смысловые единицы исходного кода (такие как ключевые слова, операторы, идентификаторы, константы и т.д.). Различные типы лексем используются для классификации этих элементов в соответствии с их синтаксическим или семантическим значением.

Таблица 4 – Типы лексем

| **Идентификатор** | **Ключевое слово, зарезервированное имя** |
| --- | --- |
| Оператор | (О1) \*  (О2) /  (О3) +  (О4) - |
| Оператор | (О5) =  (О6) /=  (О7) >  (О8) <  (О9) >=  (О10) <=  (О11) not  (О12) &  (О13) |  (О14) := |
| Разделитель | (D1) (  (D2) )  (D3) ;  (D5) . |

## 6.5 Конечный автомат лексического анализатора

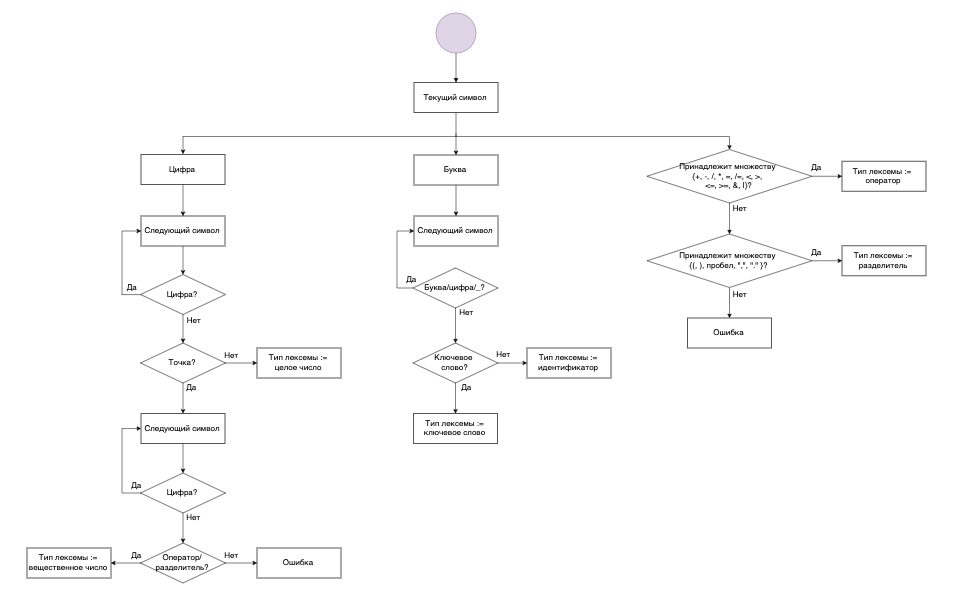


Рисунок 2 – Конечный автомат лексического анализатора

# 7 Проект синтаксического анализатора

Синтаксический анализ – это процесс, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка. Процесс синтаксического анализа – построение дерева грамматического разбора для транслируемых предложений.

Для реализации синтаксического анализатора выбран метод рекурсивного спуска.

## 7.1 Модель данных

Был реализован модуль **ASTNode**, который состоит из классов, определяющих узлы дерева разбора.

Для реализации был выбран язык C++.

Таблица 5 – Классы для описания узлов синтаксического дерева

| **Класс** | **Суперкласс** | **Краткое описание** |
| --- | --- | --- |
| BaseASTNode | *-* | Используется для общего описания узла дерева |
| VariableDeclarationNode | BaseASTNode | Используется для представления узлов объявления переменных |
| ExpressionNode | BaseASTNode | Используется для представления узлов абстрактного синтаксического дерева для выражений |
| Leaf | ExpressionNode | Используется для представления символов, идентификаторов или конкретных значений в выражениях |
| FormalParamsNode | BaseASTNode | Используется для представления формальных параметров функции. Он содержит методы для добавления параметров, вывода информации о них, а также принятия посетителей для выполнения дополнительных операций |
| ActualParamsNode | BaseASTNode | Используется для представления фактических параметров (или аргументов) функции |
| CallNode | ExpressionNode | Используется для хранения информации о вызове функции или процедуры |
| BinaryNode | ExpressionNode | Используется для хранения информации о бинарной операции |
| UnaryNode | ExpressionNode | Используется для хранения информации об унарной операции |
| AssignmentNode | BaseASTNode | Используется для хранения информации об операции присваивания |
| ReturnNode | BaseASTNode | Используется для хранения информации об операторе return |
| BlockNode | BaseASTNode | Используется для хранения информации о блоке кода или области видимости |
| ProgramNode | BlockNode | Используется для представления программы в целом |
| FunctionNode | BaseASTNode | Используется для представления определения функции |
| ProcedureNode | BaseASTNode | Используется для хранения информации об определении процедуры, включая её идентификатор, формальные параметры, тело процедуры и объявления переменных |
| ElseNode | BaseASTNode | Используется для представления блока «else» в структуре условного оператора |
| ElifNode | BaseASTNode | Используется для представления блока «elif» в структуре условного оператора |
| IfNode | BaseASTNode | Используется для представления оператора условия "if", включая его условие, тело и блоки |
| WhileNode | BaseASTNode | Используется для представления цикла "while" с заданным условием и телом |
| ForNode | BaseASTNode | Используется для представления цикла "for" с заданным итератором, начальным и конечным значениями и телом |

Для формирования дерева разбора был реализован класc **Parser**,включающий методы, позволяющие построить дерево на основе метода рекурсивного спуска. Для формирования дерева используется модуль **ASTNode**,а также лексический анализатор.

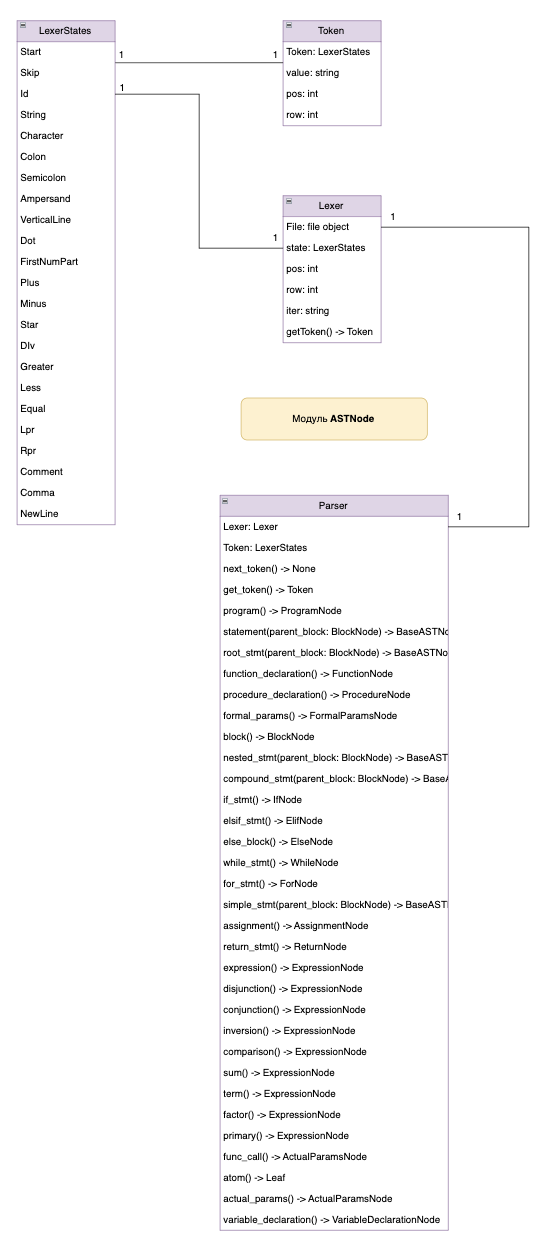


Рисунок 3 – Диаграмма классов для синтаксического анализатора

## 7.2 Описание алгоритма рекурсивного спуска синтаксического анализа

## Синтаксический анализатор методом рекурсивного спуска выстраивается набором отдельных функций, каждая из которых нужна для исследования отдельного символа языка, обозначенного в грамматике.

## Таким анализаторам свойственны следующие принципы:

## производится последовательный перебор символов входной строки, начиная с левой части строки;

## каждый символ — это основание, чтобы выбрать подходящее к нему правило из группы правил;

## присутствует «взаимное уничтожение» символов входной строки и соответствующих символов из правил «правой» части;

## для каждой группы правил создается собственная функция;

## реализован процесс «процедура-распознаватель», когда сравнивают ожидаемый символ из правой части и символ входной строки;

## когда символ из правой части полностью совпадает с символом входной строки, то символ входной строки игнорируется и функция переходит к другому символу из правой части;

## когда нет совпадения между терминальным символом и очередным символом входной строки — такая ситуация трактуется как синтаксическая ошибка;

## когда в правой части возникает нетерминальный символ, то для его распознавания нужно создать соответствующую функцию.

## 7.3 Пример дерева разбора

## 

## 7.4 Типы ошибок

1. Непредвиденный символ - Unexpected token pos=n row=n type=… value=…

# 8 Проект семантического анализатора

Семантический анализатор - это компонент, ответственный за проверку правильности семантики программы после успешного лексического и синтаксического анализа. Он обеспечивает более высокий уровень анализа, связанный со смыслом программы, и может выполнять следующие функции:

1. Проверка типов: Семантический анализатор проверяет совместимость типов данных в выражениях и операциях. Например, он может обнаруживать ошибки, если происходит попытка сложить целое число и строку.

2. Проверка объявлений и областей видимости: Анализатор убеждается, что переменные, функции и другие идентификаторы использованы в соответствии с их объявлениями и областями видимости.

3. Обработка функций и процедур: Семантический анализатор проверяет правильность вызовов функций, количество и типы передаваемых аргументов, а также корректность возвращаемых значений.

4. Оптимизация и контроль ошибок: Некоторые семантические анализаторы также могут выполнять оптимизации кода и контроль ошибок, такие как неиспользуемые переменные, недостижимый код и т. д.

5. Генерация промежуточного представления: Во многих компиляторах семантический анализатор также выполняет создание промежуточного представления программы, которое будет использоваться на следующем этапе компиляции.

Семантический анализатор играет ключевую роль в обеспечении корректности программ и разрешении многих сложных аспектов, связанных с семантикой программирования.

# 8.1 Модель данных

Были реализованы классы **SemanticAnalyzer** и **SemanticVisitor***.*

|  |  |  |
| --- | --- | --- |
| **Класс** | **Суперкласс** | **Краткое описание** |
| BaseASTNode | - | Используется для общего описания узла дерева |
| SemanticAnalyzer | - | Используется для проведения семантического анализа на основе синтаксического дерева |
| SemanticVisitor | BaseАSTNode- | Представляет собой посетителя (visitor), который может посещать различные узлы абстрактного синтаксического дерева (AST) и выполнять определенные действия для каждого узла. |

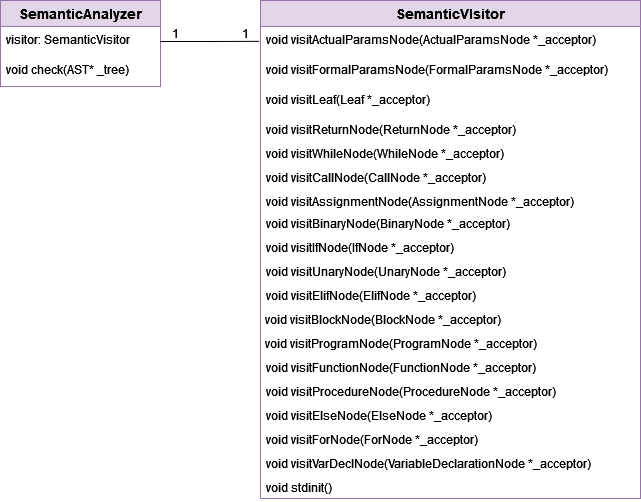
**

Рисунок 4 – Диаграмма классов для семантического анализатора

## 8.2 Методы проверки нарушений контекстных условия

Для проверки соблюдения контекстуальных условий применяются таблицы переменных, которые задаются для каждой области видимости:

* При объявлении переменной происходит проверка на наличие переменной с таким же идентификатором в таблице переменных текущей области видимости. Если переменная уже присутствует в таблице, то при выполнении семантического анализа возникает ошибка. В противном случае проверка повторяется в "родительской" области видимости. Если переменная с таким идентификатором не найдена в глобальной области видимости, то она добавляется в таблицу переменных текущей области видимости и может быть использована в дальнейшем;
* При использовании переменной в языковых конструкциях происходит аналогичная проверка наличия переменной в таблице переменных, но в этом случае ошибка возникает, если идентификатор переменной отсутствует в таблице переменных;
* Для обнаружения несовместимости типов данных переменных в операциях производится поиск соответствующих переменных, участвующих в операции, в таблице переменных, и проверяется указанный в таблице тип. Если типы не совпадают, то возникает соответствующая ошибка.

## 8.3 Типы ошибок

1. Идентификатор не объявлен – Name *name* is undefined. Occurred at row: n position: n

2. Несовпадение типов выражения и переменной в присваивании – Type mismatch occured at row: n position:n

3. Несовпадение количества аргументов в вызове функции – Parameter quantity mismatch occured at row:n position:n

4. Несовпадение типов аргументов в вызове функции - Parameter type mismatch occured at row: n position:n

5. Идентификатор уже был объявлен - Defined second time at row : n position:n

6. Условие не логического типа - Condition type is not Bool. Occurred at row: n position: n

# 9 Проект генератора кода

Генератор кода - это один из ключевых компонентов компилятора или интерпретатора, который отвечает за создание целевого исполняемого кода на основе внутреннего представления программы.  
Вот несколько важных аспектов проекта генератора кода в трансляторе:  
1.Промежуточное представление: Генератор кода работает с промежуточным представлением программы, которое было создано на предыдущих этапах компиляции.  
2. Кодогенерация: Генератор кода отображает промежуточное представление программы на целевой язык или инструкции целевой платформы.

## 9.1 Модель данных

Были реализованы классы **CodeGenerator** и **CodeEmittingNodeVisitor**.

| **Класс** | **Суперкласс** | **Краткое описание** |
| --- | --- | --- |
| BaseASTNode | *-* | Используется для общего описания узла дерева. |
| CodeGenerator | - | Используется для генерации кода на выходном языке на основе синтаксического дерева. |
| CodeEmittingNodeVisitor | АSTNode | Используется для посещения различных узлов абстрактного синтаксического дерева (AST) и генерации соответствующего кода. |



Рисунок 5 – Диаграмма классов для генератора кода

## 9.2 Метод кодогенерации

Код генерируется на основе входящих в дерево узлов: для каждого узла определена функция, которая позволяет в необходимой форме представить информацию. На основе проведенного семантического анализа определяются узлы, в рассмотрении которых при генерации кода нет необходимости. Таким образом, выходной текст будет представлять собой оптимизированный код на выходном языке.

## 9.3 Примеры кодогенерации

|  |  |
| --- | --- |
|  |  |

# 10 Тестирование

## 10.1 Лексический анализатор

|  |  |  |
| --- | --- | --- |
| Тестовая ситуация | Входные данные | Выходные данные |
| Пустой файл | Пустой файл |  |
| Объявление функции и ее вызов | function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  rangeSum(2, 10); |  |
| Объявление процедуры | procedure print(s: String)  is  begin  Ada.Text\_IO.Put\_Line(s);  end print; |  |
| Цикл for | s:=0;  for i in 1 .. 10 loop  s := s + i;  end loop; |  |
| Присваивание | s := 0;  b := 0; |  |
| Неожидаемый символ | dfvewrfgwew |  |

## 10.2 Синтаксический анализатор

|  |  |  |
| --- | --- | --- |
| Тестовая ситуация | Входные данные | Выходные данные |
| Пустой файл | Пустой файл |  |
| Объявление функции и ее вызов | function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  rangeSum(2, 10); |  |
| Объявление процедуры | procedure print(s: String)  is  begin  Ada.Text\_IO.Put\_Line(s);  end print; |  |
| Цикл for | s:=0;  for i in 1 .. 10 loop  s := s + i;  end loop; |  |
| Присваивание | s := 0;  b := 0; |  |
| Неожидаемый символ | dfvewrfgwew |  |

## 10.3 Семантический анализатор

|  |  |  |
| --- | --- | --- |
| Тестовая ситуация | Входные данные | Выходные данные |
| Идентификатор не объявлен | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + ibd;  end loop;  return s;  end rangeSum; | Name ibd is undefined  Occured at row: 16 position: 18 |
| Несовпадение типов выражения и переменной в присваивании | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: String;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum; | Type mismatch occured at row: 15 position: 9 |
| Несовпадение количества аргументов в вызове функции | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  procedure main()  is  skibidi: array(1 .. 50) of Integer;  gg: Integer;  bombom: String;  begin  (1 + 2) \* (3 - 4) / 5;  if 1 + 2 = 5 and true or not false then  print("Helloworld!");  elsif false then  print("Godbyeworld?");  else  gg := rangeSum(1000);  print("rangeSum(100, 1000)");  end if;  end main; | Parameter quantity mismatch occured at row: 33 position: 15 |
| Несовпадение типов аргументов в вызове функции | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  procedure main()  is  skibidi: array(1 .. 50) of Integer;  gg: Integer;  bombom: String;  begin  (1 + 2) \* (3 - 4) / 5;  if 1 + 2 = 5 and true or not false then  print("Helloworld!");  elsif false then  print("Godbyeworld?");  else  gg := rangeSum(bombom, 1000);  print("rangeSum(100, 1000)");  end if;  end main; | Parameter type mismatch occured at row: 33 position: 24 |
| Идентификатор уже был объявлен | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum; | Name b is already defined  Defined second time at row : 11 position: 2 |
| Условие не логического типа | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  procedure main()  is  skibidi: array(1 .. 50) of Integer;  gg: Integer;  bombom: String;  begin  (1 + 2) \* (3 - 4) / 5;  if 1 + 2 = 5 and true or not false then  print("Helloworld!");  elsif gg then  print("Godbyeworld?");  else  gg := rangeSum(100, 1000);  print("rangeSum(100, 1000)");  end if;  end main; | Condition type is not Bool.  Occured at row: 30 position: 11 |

## 10.4 Генератор кода

|  |  |  |
| --- | --- | --- |
| Тестовая ситуация | Входные данные | Выходные данные |
| Генерация кода, в котором есть ошибка | function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: String;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum; | Пустой файл |
| Генерация кода, в котором содержатся всевозможные узлы | procedure print(s: String)  is  begin  Put\_Line(s);  end print;  function rangeSum(rangeStart: Integer; rangeEnd: Integer) return Integer  is  s: Integer;  b: Integer;  i: Integer;  begin  s := 0;  b := 0;  for i in rangeStart .. rangeEnd loop  s := s + i;  end loop;  return s;  end rangeSum;  procedure main()  is  skibidi: array(1 .. 50) of Integer;  gg: Integer;  bombom: String;  begin  (1 + 2) \* (3 - 4) / 5;  if 1 + 2 = 5 and true or not false then  print("Helloworld!");  elsif false then  print("Godbyeworld?");  else  gg := rangeSum(100, 1000);  print("rangeSum(100, 1000)");  end if;  end main; | #include <bits/stdc++.h>  void print(std::string *s*)  {  Put\_Line(s);  };  int rangeSum(int *rangeStart*, int *rangeEnd*)  {  int s;  int b;  int i;  s = 0;  b = 0;  for (int i = rangeStart; i < rangeEnd; i++)  {  s = (s + i);  };  return s;  };  void main()  {  int skibidi[50];  int gg;  std::string bombom;  (((1 + 2) \* (3 - 4)) / 5);  if (((((1 + 2) == 5) && true) || !false))  {  print("Helloworld!");  }else if (false)  {  print("Godbyeworld?");  }else{  gg = rangeSum(100, 1000);  print("rangeSum(100, 1000)");  };  };  ; |

# Заключение

В рамках курсовой работы был разработан транслятор, который преобразует программу, содержащую подмножество языка Ada, в программу, содержащую подмножество языка C++ генерируя эквивалентный исходный код.

Для этого были выполнены следующие задачи:

1. описана грамматика подмножества входного языка;
2. описаны контекстные условия входного и выходного языков;
3. описано соответствие конструкций входного и выходного языков;
4. разработан проект лексического анализатора;
5. разработан проект синтаксического анализатора;
6. разработан проект семантического анализатора;
7. разработан проект генератора оптимизированного кода выходного языка;
8. реализован транслятор входного языка в выходной;
9. выполнено тестирование разработанного транслятора.

# Список литературы

1. Ada Reference Manual // URL: <http://www.ada-auth.org/standards/22rm/html/RM-TOC.html> (дата обращения: 10.04.2023)
2. Документация по Microsoft C++ // Microsoft Learn URL: <https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170> (дата обращения: 10.04.2023)