

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение  
высшего образования

**«Дальневосточный федеральный университет»**

**ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ (ШКОЛА)**

**Департамент программной инженерии и искусственного интеллекта**

РАЗРАБОТКА ТРАНСЛЯТОРА, ПЕРЕВОДЯЩЕГО ПОДМНОЖЕСТВО ЯЗЫКА GOLANG В ЭКВИВАЛЕНТНОЕ ПОДМНОЖЕСТВО ЯЗЫКА PYTHON

**ОТЧЁТ ПО ЛАБОРАТОРНОЙ РАБОТЕ**

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

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Бронников А.А.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Разумов С. И.

Руководитель:

ассистент департамента ПИиИИ

Глазырин М.А.

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

(подпись)

г. Владивосток

2024 г

Оглавление

[Оглавление 2](#_Toc157210910)

[Введение 4](#_Toc157210911)

[1 Неформальная постановка задачи 5](#_Toc157210912)

[2 Синтаксис входного языка 7](#_Toc157210913)

[3 Контекстные условия языка Golang 9](#_Toc157210914)

[3.1 Контекстные условия о правилах описания идентификаторов в программах 9](#_Toc157210915)

[3.2 Контекстные условия о правилах использования идентификаторов в своей области действия 10](#_Toc157210916)

[3.3 Контекстные условия, определяющие правила соответствия видов величин 10](#_Toc157210917)

[4 Контекстные условия языка Python 11](#_Toc157210918)

[4.1 Контекстные условия о правилах описания идентификаторов в программах 11](#_Toc157210919)

[4.2 Контекстные условия о правилах использования идентификаторов в своей области действия 11](#_Toc157210920)

[4.3 Контекстные условия, определяющие правила соответствия видов величин 12](#_Toc157210921)

[5 Таблица соответствия языков 12](#_Toc157210922)

[6 Проект лексического анализатора 14](#_Toc157210923)

[6.1 Модель данных 14](#_Toc157210924)

[6.2 Таблица ключевых слов 15](#_Toc157210925)

[6.3 Таблица зарезервированных слов 15](#_Toc157210926)

[6.4 Типы лексем 16](#_Toc157210927)

[6.5 Типы ошибок 17](#_Toc157210928)

[7 Проект синтаксического анализатора 18](#_Toc157210929)

[7.1 Модель данных 18](#_Toc157210930)

[7.2 Описание алгоритма метода рекурсивного спуска 18](#_Toc157210931)

[7.3 Пример дерева разбора 19](#_Toc157210932)

[7.4 Типы ошибок 20](#_Toc157210933)

[8 Проект семантического анализатора 21](#_Toc157210934)

[8.1 Модель данных 21](#_Toc157210935)

[8.2 Описание алгоритма анализ потока данных 21](#_Toc157210936)

[8.3 Типы ошибок 21](#_Toc157210937)

[9 Проект генератора кода 23](#_Toc157210938)

[9.1 Модель данных 23](#_Toc157210939)

[9.2 Метод шаблонной кодогенерации 23](#_Toc157210940)

[9.3 Пример кодогенерации 23](#_Toc157210941)

[10 Тестирование 25](#_Toc157210942)

[10.1 Лексический анализатор 25](#_Toc157210943)

[10.2 Синтаксический анализатор 26](#_Toc157210944)

[10.3 Семантический анализатор 27](#_Toc157210945)

[10.4 Генератор кода 28](#_Toc157210946)

[Заключение 30](#_Toc157210947)

Введение

**Транслятор** – программа или техническое средство, выполняющее преобразование программы, ставленной на одном из языков программирования, в программу, написанную на другом языке.

Язык программирования – [формальный язык](https://ru.wikipedia.org/wiki/%D0%A4%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), предназначенный для записи [компьютерных программ](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0). Язык программирования определяет набор [лексических](https://ru.wikipedia.org/wiki/%D0%9B%D0%B5%D0%BA%D1%81%D0%B8%D0%BA%D0%B0), [синтаксических](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BD%D1%82%D0%B0%D0%BA%D1%81%D0%B8%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и [семантических](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D0%BA%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) правил, определяющих внешний вид программы и действия, которые выполнит исполнитель под её управлением.

В качестве входного языка был выбран – Golang.

В качестве выходного языка был выбран – Python.

**Цель курсового проекта:** разработать проект транслятора, который преобразует программу, содержащую подмножество языка Golang, в программу, содержащую подмножество языка Python 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

В ходе была определена грамматика входного языка, определены контекстные условия входного и выходного языка, составлена таблица соответствия языков, создан лексический, синтаксический и семантический анализатор, был создан генератор кода выходного языка, а также проведено тестирование.

Подмножества языка Golang включает в себя:

1. **Типы данных:**
   1. Числовой тип, целочисленный: *int*.
   2. Числовой тип, с плавающей точкой: *float64*.
   3. Логический тип: *bool*.
   4. Строковый тип: *string*.
   5. Произвольный тип, массив: *array*.
2. **Операторы:**
   1. Арифметические операторы: *+, -, \*, /, %, ++, --*.
   2. Операторы сравнения: *==, !=, >, <, >=, <=*.
   3. Логические операторы: *&&, ||, !*.
   4. Операторы присваивания:
      1. Простые операторы присваивания: *=, :=*.
      2. Арифметические операторы присваивания: *+=, -=, \*=, /=, %=*.
3. **Переменные:**
   1. Определение переменной:

var «список переменных» «тип»

* 1. Типизация и объявление:
     1. Статическая типизация:

*var «идентификатор переменной» «тип» = «значение»*

* + 1. Динамическая типизация:

*var «идентификатор переменной» = «значение»*

1. **Константы:**
   1. Определение константы:

const «список переменных» «тип»

* 1. Типизация и объявление:
     1. Статическая типизация:

*const «идентификатор переменной» «тип» = «значение»*

* + 1. Динамическая типизация:

*const «идентификатор переменной» = «значение»*

1. **Операторы ветвления:**
   1. Оператор if:

*if («условие 1») { «…» }*

*if («условие 1») { «…» } else { «…» }*

*if («условие 1») { «…» } else if («условие 2») { «…» } …*

*if («условие 1») { «…» } else if («условие 2») { «…» } … else { «…» }*

* 1. Оператор switch:

*switch («тип») {*

*case «тип»: «…»*

*…*

*default: «…»*

*}*

* 1. Операторы контроля switch: *break*.

1. **Операторы циклов:**
   1. Цикл for:

*for «условие» { «…» }*

*for «инициализация»; «условие»; «приращение» { «…» }*

* 1. Операторы контроля циклов: *break, continue*.

1. **Функции:**
   1. Встроенные стандартные функции:

Функция печати: *func print(args ...Type)*

Функция печати: *func println(args ...Type)*

* 1. Описание функции:

*func «идентификатор функции»(«список параметров») «тип» {*

*«…»*

*}*

* 1. Вызов функции:

*«идентификатор переменной» = «идентификатор функции» («список параметров»)*

# 2 Синтаксис входного языка

В качестве разделителей строк, могут быть использованы следующие символы: ";" и "\n".

Существуют два вида комментариев:

* Строковые комментарии (однострочные) – начинаются с последовательности символов "//" и заканчиваются в конце строки.
* Общие комментарии (многострочные) – начинаются с последовательности символов "/\*" и заканчиваются первой последовательностью символов "\*/".

**Грамматика входного языка Golang.**

Определим правила описания грамматики, основные литеры и нетерминалы, а также грамматику входного языка.

**Аксиома:** SourceFile

|  |  |  |
| --- | --- | --- |
| **Правила описания и обозначения** | | |
| *|* | Альтернатива. | |
| *( )* | Группировка. | |
| *[ ]* | Вариант (0 или 1 раз)  A ::= B[C] ⬄ A ::= B | BC | |
| *{ }* | Повторение (от 0 до N раз)  A ::= B{C} ⬄ A ::= B | BC | BCC | … | |
| *…* | Диапозон.  A ::= "0" … "5" ⬄ A ::= "0" | "1" | "2" | "3" | "4" | "5" | |
| **Основные литеры и нетерминалы** | | |
| ***letter*** | | *"a" … "z" | "A" … "Z" | "\_"* |
| ***digit*** | | *"0" … "9"* |
| ***char*** | | *Любой символ кроме "\n"* |
| ***newline*** | | *"\n"* |
| ***int\_lit*** | | *"0" | "1" … "9" [ digits ]* |
| ***float\_lit*** | | *int\_lit "." digits* |
| ***digits*** | | *digit { digit }* |
| ***string\_lit*** | | *"`" { char | newline } "`" | """ { char } """* |
| ***binary\_op*** | | *"||" | "&&" | rel\_op | add\_op | mul\_op* |
| ***unary\_op*** | | *add\_op | "!"* |
| ***assign\_op*** | | *[ add\_op | mul\_op ] "="* |
| ***rel\_op*** | | *"==" | "!=" | "<" | "<=" | ">" | ">="* |
| ***add\_op*** | | *"+" | "-"* |
| ***mul\_op*** | | *"\*" | "/" | "%"* |
| **Грамматика языка** | | |
| ***SourceFile*** | | *PackageClause ";" { TopLevelDecl ";" }* |
| ***PackageClause*** | | *"package" Identifier* |
| ***TopLevelDecl*** | | *Declaration | FunctionDecl* |
| ***Declaration*** | | *ConstDecl | VarDecl* |
| ***Type*** | | *Identifier | ArrayType | "(" Type ")"* |
| ***ArrayType*** | | *"[" Expression "]" Type* |
| ***ConstDecl*** | | *"const" Identifier [ Type ] "=" Expression* |
| ***VarDecl*** | | *"var" Identifier [ Type ] "=" Expression* |
| ***FunctionDecl*** | | *"func" Identifier Signature Block* |
| ***Signature*** | | *"(" [ ParameterList ] ")" [ Type ]* |
| ***ParameterList*** | | *ParameterDecl { "," ParameterDecl }* |
| ***ParameterDecl*** | | *IdentifierList Type* |
| ***Block*** | | *"{" StatementList "}"* |
| ***StatementList*** | | *{ Statement ";" }* |
| ***Statement*** | | *Declaration | SimpleStmt | "continue" | "break" |"return" [ Expression ] |*  *IfStmt | SwitchStmt | ForStmt* |
| ***SimpleStmt*** | | *Expression ( [ "++" | "--" ] | assign\_op Expression | ":=" Expression )* |
| ***Expression*** | | *UnaryExpr [ binary\_op Expression ]* |
| ***ExpressionList*** | | *Expression { "," Expression }* |
| ***Identifier*** | | *letter { letter | digit }* |
| ***IdentifierList*** | | *Identifier { "," Identifier }* |
| ***Operand*** | | *Identifier | int\_lit | float\_lit | string\_lit | "(" Expression ")"* |
| ***PrimaryExpr*** | | *Operand [ { "." Identifier | "[" Expression "]" |*  *"(" [ ExpressionList ] ")" } ]* |
| ***UnaryExpr*** | | *PrimaryExpr | unary\_op UnaryExpr* |
| ***IfStmt*** | | *"if" Expression Block [ "else" ( IfStmt | Block ) ]* |
| ***SwitchStmt*** | | *"switch" Expression "{" { ExprCaseClause } "}"* |
| ***ExprCaseClause*** | | *( "case" ExpressionList | "default" ) ":" StatementList* |
| ***ForStmt*** | | *"for" ( Expression | SimpleStmt ";" Expression ";" SimpleStmt ) Block* |

# 3 Контекстные условия языка Golang

3.1 Контекстные условия о правилах описания идентификаторов в программах

* Все используемые в программах идентификаторы должны быть описаны до их использования в программе.
* Каждый из идентификаторов, используемых в программе, должен быть описан один раз в рамках одного блока программы, модуля программы, подпрограммы, описание в формальных параметрах функций и другого.
* Каждый из идентификаторов, используемых в программе, может быть описан несколько раз при условии, что различные описания этого идентификатора имеют разные области действия (блок программы, модуль программы, подпрограмма, описание в формальных параметрах функций и другого).
* Идентификатор – последовательность букв (латинского алфавита), цифр и специального символа «\_». Идентификатор может начинаться с буквы или специального символа «\_».
* Регистр символов в идентификаторе значение имеет.
* Некоторые слова являются ключевыми, то есть служат для оформления конструкций языка и не могут использоваться как идентификаторы (const, var, if, …).

3.2 Контекстные условия о правилах использования идентификаторов в своей области действия

* Идентификатор не может быть использован при его объявлении.
* Идентификатор может быть использован только внутри определенных областей действия.
* Область действия идентификатора может быть на уровне модуля (глобальные идентификаторы), внутри функции (локальные идентификаторы) или внутри блока кода (идентификаторы, ограниченные блоком).

3.3 Контекстные условия, определяющие правила соответствия видов величин

* Количество объявленных в методе или функции параметров должно совпадать с количеством переданных.
* Использование и описание массивов должно соответствовать выделенной им области памяти.
* Неявные преобразования при использовании разных типов запрещены.

# 4 Контекстные условия языка Python

4.1 Контекстные условия о правилах описания идентификаторов в программах

* Все используемые в программах идентификаторы должны быть описаны до их использования в программе.
* Идентификатор – последовательность букв (латинского алфавита), цифр и специального символа «\_». Идентификатор может начинаться с буквы или специального символа «\_».
* Регистр символов в идентификаторе значение имеет.
* Некоторые слова являются ключевыми, то есть служат для оформления конструкций языка и не могут использоваться как идентификаторы (def, class, if, …).

4.2 Контекстные условия о правилах использования идентификаторов в своей области действия

* Идентификатор не может быть использован при его объявлении.
* Идентификатор может быть использован только внутри определенных областей действия.
* Область действия идентификатора может быть на уровне модуля (глобальные идентификаторы), внутри функции (локальные идентификаторы) или внутри блока кода (идентификаторы, ограниченные блоком).

4.3 Контекстные условия, определяющие правила соответствия видов величин

* Количество объявленных в методе или функции параметров должно совпадать с количеством переданных.
* Переменные могут быть связаны с объектами разных типов в течение жизни программы. Нет необходимости предварительно объявлять типы и устанавливать их на этапе компиляции.
* Неявные преобразования типов могут производить некоторые операции, например при выполнении математических операций целые числа могут неявно преобразовываться в числа с плавающей точкой, чтобы выполнить операцию.

# 5 Таблица соответствия языков

Для наглядного представления о соответствиях отдельных лексем одного языка (Golang) другому (Python) составим таблицу соответствий Golang – Python.

Таблица 1 – Соответствие Golang – Python

|  |  |
| --- | --- |
| **Golang** | **Python** |
| Структура программы | |
| *package <Identifier>*  *<Top Level Decl>* | *<Top Level Decl>* |
| Объявление констант и переменных | |
| *const <Identifier> <[Type]> = <Expression>*  *var <Identifier> <[Type]> = <Expression>*  *<Identifier> := <Expression>* | *<Identifier> = <Expression>* |
| *Продолжение таблицы 1* | |
| **Golang** | **Python** |
| Функции | |
| Объявление функции  *func <Identifier> ( <[ Parameter List]> ) <[Type]> {*  *<Block>*  *}*  Вызов  *<Identifier> ( <[ Parameter List]> )* | Объявление функции  *def <Identifier> ( <[ Parameter List]> ):*  *<Block>*  Вызов  *<Identifier> ( <[ Parameter List]> )* |
| Ветвление | |
| *if <Condition> { <Block> }*  *else if <Condition> { <Block> }*  *else { <Block> }* | *if <Condition>:*  *<Block>*  *elif <Condition>:*  *<Block>*  *else:*  *<Block>* |
| Выбор | |
| *switch <Condition> {*  *case <Expression>: <Block>*  *default: <Block>*  *}* | *match <Condition>:*  *case <Expression>: <Block>*  *case \_: <Block>* |
| Цикл | |
| *for <Simple Stmt>; <Expression>; <Simple Stmt> {*  *<Block>*  *}* | *for <Identifier> in <Range>:*  *<Block>* |

# 6 Проект лексического анализатора

Лексический анализ – процесс аналитического разбора входной последовательности символов с целью получения на выходе последовательности символов, называемых «токенами».

При этом, группа символов входной последовательности, идентифицируемая на выходе процесса как токен, называется лексема, т.е. в процессе лексического анализа производится распознавание и выделение лексем из входной последовательности символов.

Лексический анализатор решает следующие задачи:

* Выделение токенов (лексем) из текста.
* Отделение лексем от разделительных символов.
* Удаление комментариев.

## 6.1 Модель данных

Лексический анализатор был реализован на языке программирования python в виде класса "Lexer" в котором реализовано несколько функций, обеспечивающих работу анализатора.

Lexer.tokenization – принимает на вход строку содержащую исходный код и запускает алгоритм выделения лексем из текста.

Lexer.curren\_token – возвращает лексему на которую указывает метка.

Lexer.next\_token – перекидывает метку на следующую лексему и возвращает ее.

Lexer.last\_token – перекидывает метку на предыдущую лексему и возвращает ее.

Lexer.next – перекидывает метку на следующую лексему.

Lexer.last – перекидывает метку на предыдущую лексему.

## 6.2 Таблица ключевых слов

Ключевое слово – лексема, обладающая особым семантическим значением в контексте языка программирования. Оно представляет собой зарезервированное слово, за которым закреплен какой-либо функционал.

Множество ключевых слов представлено в таблице 2.

Таблица 2 – Ключевые слова

|  |  |
| --- | --- |
| **Ключевое слово** | **Ключ** |
| package | KEYWORD\_1 |
| func | KEYWORD\_2 |
| const | KEYWORD\_3 |
| var | KEYWORD\_4 |
| if | KEYWORD\_5 |
| else | KEYWORD\_6 |
| for | KEYWORD\_7 |
| switch | KEYWORD\_8 |
| case | KEYWORD\_9 |
| default | KEYWORD\_10 |
| continue | KEYWORD\_11 |
| break | KEYWORD\_12 |
| return | KEYWORD\_13 |

## 6.3 Таблица зарезервированных слов

Зарезервированное слово – лексема, зарезервированная в коде языка. Зарезервированные слова не могут быть использованы в роли констант, имён переменных или любых других идентификаторов.

Множество зарезервированных слов представлено в таблице 3.

Таблица 3 – Зарезервированные слова

|  |  |
| --- | --- |
| **Зарезервированное слово** | **Значение** |
| true | RESERVEDWORD\_1 |
| false | RESERVEDWORD\_2 |
| Print | RESERVEDWORD\_3 |
| Println | RESERVEDWORD\_4 |

**6.4 Типы лексем**

Лексема – минимальная единица языка, имеющая собственный смысл и определенную грамматическую структуру.

Тип лексемы – это категория, к которой относится определенная единица языка.

Типы лексем представлены в таблице 4.

Таблица 4 – Типы лексем

|  |  |  |
| --- | --- | --- |
| **Название** | **Представление** | |
| Идентификатор | Ключевое слово  Зарезервированное слово  const  var  func | |
| Комментарий | Однострочный комментарий  Многострочный комментарий | |
| Оператор | ADD | *"+", "–"* |
| MUL | *"\*", "/", "%"* |
| INCREMENT | *"++", "––"* |
| REL | *"==", "!=", ">", "<", ">=", "<="* |
| BINARY | *"||", "&&"* |
| UNARY | *"!"* |
| EQUALS | *"=", ":="* |
| *Продолжение таблицы 4* | | |
| **Название** | **Представление** | |
| Разделитель | *"*(*"*, *"*)*"*  *"*{*"*, *"*}*"*  *"*[*"*,*"*]*"*  *"*.*"*  *","*  *":"*  *";", "\n"* | |
| Число | Число без точки (int)  Число с точкой (float) | |
| Строка | Последовательность символов, ограниченных кавычками (string) | |

## 6.5 Типы ошибок

Были выделены следующие типы ошибок представленные в таблице 4.

Таблица 4 – Типы ошибок лексического анализатора

|  |  |
| --- | --- |
| **Тип ошибки** | **Пример** |
| Incorrect identifier | var 123\_ident int = 0  =>  Лексический анализ:  - Инициализация лексера  - Лексический анализ: запущен  - Лексический анализ: прерван  - \* ERROR: Incorrect identifier: 123\_ident |

# 7 Проект синтаксического анализатора

Синтаксический анализ – процесс сопоставления последовательности лексем формального языка в соответствии с грамматикой языка. Он определяет, является ли последовательность лексем валидным предложением или программой в соответствии с определенными правилами синтаксиса языка.

Для реализации синтаксического анализа был выбран метод рекурсивного спуска (Recursive Descent Parser).

**7.1 Модель данных**

Синтаксический анализатор был реализован на языке программирования python в виде класса "Parser" в котором реализовано несколько функций, обеспечивающих работу анализатора.

Для описания структуры данных синтаксического анализатора была сделана диаграмма моделей данных. Диаграмма моделей данных представлена на рисунке 1.

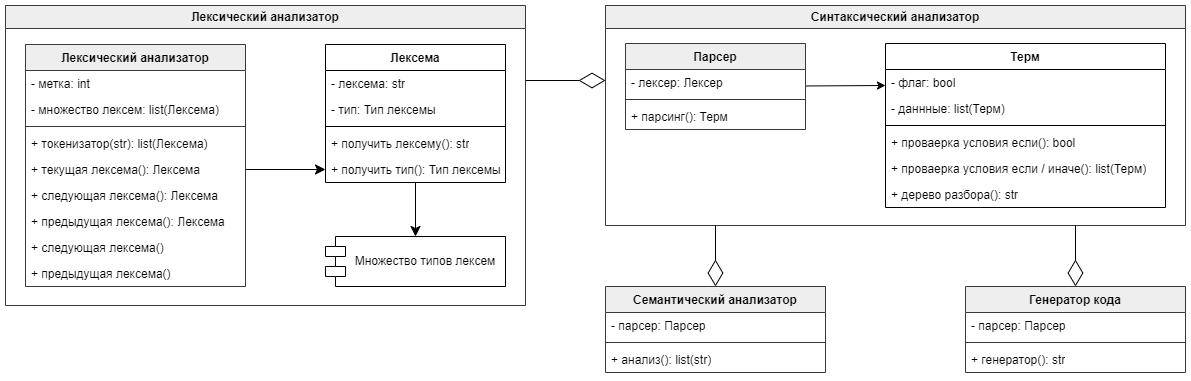


Рисунок 1 – Диаграмма моделей данных

## 7.2 Описание алгоритма метода рекурсивного спуска

Метод рекурсивного спуска — алгоритм нисходящего синтаксического анализа, реализуемый путём взаимного вызова процедур, где каждая процедура соответствует одному из правил грамматики.

Общий алгоритм метода рекурсивного спуска:

– Определение набора правил грамматики.

– Создание функций разбора.

– Рекурсивный вызов функций разбора.

– Обработка терминалов.

– Построение синтаксического дерева.

## 7.3 Пример дерева разбора

Пример дерева разбора представлен в таблице 5.

Таблица 5 – Пример дерева разбора

|  |  |
| --- | --- |
| **Входные данные** | **Дерево разбора** |
| *package main*  *var a int = 1*  *var b int = 2*  *func main() {*  *Print(a + b)*  *}* | *Пакетирование: main*  *Var: a*  *| Type: int*  *| int\_lit: 1*  *Var: b*  *| Type: int*  *| int\_lit: 2*  *Функция: main*  *| Параметры: (*  *| Входные: ()*  *| Выходные: ()*  *| )*  *| Блок: {*  *| identifier: Print (*  *| Token: { addop: + }*  *| identifier: a*  *| identifier: b*  *| )*  *| }* |

## 7.4 Типы ошибок

Были выделены следующие типы ошибок представленные в таблице 6.

Таблица 6 – Типы ошибок синтактического анализатора

|  |  |
| --- | --- |
| **Тип ошибки** | **Пример** |
| Identifier / Received | var var int = 0  =>  Синтаксический анализ:  - Инициализация парсера  - Синтаксический анализ: запущен  Error:Expected 'Identifier', Received 'var : var'  Error:Expected 'Semicolon', Received 'var : var' |

# 8 Проект семантического анализатора

Семантический анализ – программа включает в себя проверку правильности использования переменных и функций в коде. Одной из ошибок, связанных с семантическим анализом, является повторное объявление переменных и функций.

Для реализации этой задачи был выбран алгоритм анализа потока данных (Data Flow Analysis).

**8.1 Модель данных**

Семантический анализатор был реализован на языке программирования python в виде класса "Semantic" в котором реализовано несколько функций, обеспечивающих работу анализатора.

Для описания структуры данных семантического анализатора была сделана диаграмма моделей данных. Диаграмма моделей данных представлена на рисунке 1 (Глава 7.1).

## 8.2 Описание алгоритма анализ потока данных

Общий алгоритм Data Flow Analysis:

– Инициализация.

– Построение графа потока управления.

– Определение уравнений потока данных.

– Решение уравнений потока данных.

– Анализ результатов.

## 8.3 Типы ошибок

Были выделены следующие типы ошибок представленные в таблице 7.

Таблица 7 – Типы ошибок семантического анализатора

|  |  |
| --- | --- |
| **Тип ошибки** | **Пример кода** |
| Redeclaring a variable | var a int = 0  var a = 0 |
| Expression contains multiple types | var a int = 0 + 0.0 |
| Unknown variable | var a int = temp |
| Type mismatch | var a int = 0.0 |
| Redeclaring a function | func plus(a, b int) int { return a + b }  func plus(a, b float64) float64{ return a + b } |
| The function returns unknown type | func plus(a, b int) ints { return a + b } |
| Redeclaring a variable in function parameters | func plus(a, b int, a float64) int  { return a + b } |
| Not identifier | (i)++ |
| Continue is not in loop | func main() { continue } |
| Break is not in loop or switch | func main() { break } |
| None return value | func main() { return 0 } |
| Not return value | func main() int { return } |
| Expression contains multiple types in return | func main() int { return 0 + 0.0 } |
| Unknown variable in return | func main() int { return temp } |
| The first element is the counter declaration | for i <= 0; i <= 0; i++ { … } |
| The third element is the counter increment | for i := 0; i <= 0; i <= 0 { … } |

# 9 Проект генератора кода

Генератор кода – процесс компиляции, предназначенный для автоматической генерации кода на основе синтаксически корректного исходного кода (преобразованного в синтаксическое дерево).

Для реализации кодогенерации был выбран метод шаблонной кодогенерации (Template Code Generator).

**9.1 Модель данных**

Генератор кода был реализован на языке программирования python в виде класса "Generator" в котором реализовано несколько функций, обеспечивающих работу анализатора.

Для описания структуры данных генератора кода была сделана диаграмма моделей данных. Диаграмма моделей данных представлена на рисунке 1 (Глава 7.1).

## 9.2 Метод шаблонной кодогенерации

Общий алгоритм метода шаблонной кодогенерации:

– Определение и создание набора шаблонов.

– Получение синтаксического дерева.

– Обработка синтаксического дерева на основе имеющихся шаблонов.

– Генерация кода.

## 9.3 Пример кодогенерации

Пример дерева разбора представлен в таблице 8.

Таблица 8 – Пример дерева разбора

|  |  |
| --- | --- |
| **Входные данные** | **Выходные данные** |
| *package main*  *var a int = 1*  *var b int = 2*  *func main() {*  *Print(a + b)*  *}* | *a = 1*  *b = 2*  *def main()*  *print(a + b)* |

**9.4 Оптимизация кода**

Оптимизации кода при трансляции с одного языка на другой представляет собой инструмент, который осуществляет оптимизацию исходного кода программы перед его генерацией на другой язык программирования.

В качестве метода оптимизации кода был использован метод удаления неиспользуемых констант и переменных, а также комментариев.

Пример оптимизации кода представлен в таблице 9.

Таблица 9 – Пример оптимизации кода

|  |  |
| --- | --- |
| **Входные данные** | **Выходные данные** |
| *package main*  *// Const*  *const a int = 1*  *const a int = 2*  *const a int = 3*  *// Var*  *var b int = 1*  *var b int = 2*  *var b int = 3* | *a = 3*  *b = 3* |

# 10 Тестирование

## 10.1 Лексический анализатор

|  |  |  |
| --- | --- | --- |
| **Ситуация** | **Входные данные** | **Выходные данные** |
| Пустой файл |  | [] |
| Объявление пакета | package main | [Token: { package: package }, Token: { identifier: main }, Token: { semicolon: \n }] |
| Объявление константы | const a int = 1 | [Token: { const: const }, Token: { identifier: a }, Token: { identifier: int }, Token: { equals: = }, Token: { int\_lit: 1 }, Token: { semicolon: \n }] |
| Объявление переменной | var a int = 1 | [Token: { var: var }, Token: { identifier: a }, Token: { identifier: int }, Token: { equals: = }, Token: { int\_lit: 1 }, Token: { semicolon: \n }] |
| Объявление функции | func main()  {} | [Token: { func: func }, Token: { identifier: main }, Token: { lbr: ( }, Token: { rbr: ) }, Token: { semicolon: \n }, Token: { lcbr: { }, Token: { rcbr: } }, Token: { semicolon: \n }] |
| Ключевое слово | continue | [Token: { continue: continue }] |
| Зарезервированное слово | Print | [Token: { identifier: Print }] |
| Число | 0  0.0 | [Token: { int\_lit: 0 }, Token: { semicolon: \n }, Token: { float\_lit: 0.0 }, Token: { semicolon: \n }] |
| Строка | "Hello world!" | [Token: { string\_lit: "Hello world!" }] |
| Оператор | + - | [Token: { addop: + }, Token: { addop: - }] |
| Некорректный идентификатор | 1a | ERROR: Incorrect identifier: 1a |

## 10.2 Синтаксический анализатор

|  |  |  |
| --- | --- | --- |
| **Ситуация** | **Входные данные** | **Выходные данные** |
| Пустой файл |  | Error:Expected 'PackageClause', Received 'EOF : None'  PARSING - OK |
| Объявление пакета | package main | Пакетирование: main |
| Объявление константы | package main  const a int = 1 + 2 + 3 | Пакетирование: main  Const: a  | Type: int  | int\_lit: 1  | Token: { addop: + }  | int\_lit: 2  | Token: { addop: + }  | int\_lit: 3 |
| Объявление переменной | package main  var a int = 1 + 2 + 3 | Пакетирование: main  Var: a  | Type: int  | int\_lit: 1  | Token: { addop: + }  | int\_lit: 2  | Token: { addop: + }  | int\_lit: 3 |
| Объявление функции | package main  func main() {} | Пакетирование: main  Функция: main  | Параметры: (  | Входные: ( )  | Выходные: ( )  | )  | Блок: {  | } |
| Функция с параметрами | package main  func main(a, b int) int {  c := a + b  return c  } | Пакетирование: main  Функция: main  | Параметры: (  | Входные: (  | | Type: int  | | | identifier: a, identifier: b  | )  | Выходные: (  | | Type: int  | )  | )  | Блок: {  | identifier: c :=  | Token: { addop: + }  | identifier: a  | identifier: b  | Return  | | identifier: c  | } |
| Блок IF | package main  func main() {  if true {  } else {  }  } | Пакетирование: main  Функция: main  | Параметры: (  | Входные: ( )  | Выходные: ( )  | )  | Блок: {  | Ветвление: if:  | | Параметры: (  | | identifier: true  | | )  | | Блок {  | | }  | Ветвление: else  | | Блок {  | | }  | } |
| Блок SWITCH | package main  func main() {  switch i {  case 1: break  default: break  }  } | Пакетирование: main  Функция: main  | Параметры: (  | Входные: ( )  | Выходные: ( )  | )  | Блок: {  | Ветвление: switch: (  | | identifier: i  | | )  | | Блок {  | | Case: (  | | | int\_lit: 1  | | | )  | | | Блок: {  | | | Break  | | | }  | | Default:  | | | Блок: {  | | | Break  | | | }  | | }  | } |
| Блок FOR | package main  func main() {  for i:=0; i<10; i++ {  Print(i)  }  } | Пакетирование: main  Функция: main  | Параметры: (  | Входные: ( )  | Выходные: ( )  | )  | Блок: {  | Цикл: for  | | Параметры: (  | | identifier: i :=  | | . int\_lit: 0  | | Token: { relop: < }  | | identifier: i  | | int\_lit: 10  | | identifier: i++  | | )  | | Блок: {  | | identifier: Print (  | | identifier: i  | | )  | | }  | } |
| Блок WHILE | package main  func main() {  for i<10 {  Print(i)  }  } | Пакетирование: main  Функция: main  | Параметры: (  | Входные: ( )  | Выходные: ( )  | )  | Блок: {  | Цикл: while  | | Параметры: (  | | Token: { relop: < }  | | identifier: i  | | int\_lit: 10  | | )  | | Блок: {  | | identifier: Print (  | | identifier: i  | | )  | | }  | } |
| Identifier / Received | func main(a, b int) int {  c := a + b  return c  } | Error:Expected 'PackageClause', Received 'func : func'  PARSING - FALSE |

## 10.3 Семантический анализатор

|  |  |  |
| --- | --- | --- |
| **Ситуация** | **Входные данные** | **Выходные данные** |
| Переопределение констант и переменных | const a int = 0; a := 10  const a = “Hello”  var b int = 0; b := 10  var b = “Hello” | E::Redeclaring a variable  ::main  E::Redeclaring a variable  ::a  E::Redeclaring a variable  ::main  E::Redeclaring a variable  ::b |
| Переопределение констант и переменных в функции | func main(a, b int) {  const a int = 0;  var b int = 0;  } | W::Redeclaring a variable  ::a  W::Redeclaring a variable  ::b |
| Переопределение функции | func plus(a, b int) int {  return a + b  }  func plus(a, b float64) float64 {  return a + b  } | E::Redeclaring a function  ::plus |
| Неизвестная переменная | A = 1 + 2 + 3 + 4 + 5 | E::Unknown variable  ::main |
| Операции с разными типами | var a int = 0 + “str”  var a int = 0.0 | E::Expression contains multiple types  ::a  E::Type mismatch: int / float  ::a |
| Некорректная функция | func plus(a, b int) ints {  return a + b  }  func plus(a, a int) int {  return a + b  } | E::The function returns unknown type  ::plus  E::Redeclaring a variable in function parameters  ::a |
| Continue / Break | func main() {  continue  break  } | E::Continue is not in loop  ::main  E::Break is not in loop or switch  ::main |
| Return | func main() {return 0 }  func main() int { return }  func main() int { return 0 + 0.0 }  func main() int { return temp } | E::None return value  ::main  E::Not return value  ::main  E::Expression contains multiple types in return  ::main  E::Unknown variable in return  ::main |
| Некорректный цикл | for i <= 0; i <= 0; i++ {  Print(i)  }  for i := 0; i <= 0; i <= 0 {  Print(i)  } | E::The first element is the counter declaration  ::main for  E::The third element is the counter increment  ::main for |

## 10.4 Генератор кода

|  |  |  |
| --- | --- | --- |
| **Ситуация** | **Входные данные** | **Выходные данные** |
| Пустой файл |  |  |
| Объявление пакета | package main |  |
| Объявление константы | package main  const a int = 1 + 2 + 3 | a = 1+2+3 |
| Объявление переменной | package main  var a int = 1 + 2 + 3 | a = 1+2+3 |
| Объявление функции | package main  func main(a, b int) int {  c := a + b  return c  } | def main(a, b):  c = a+b  return c |
| Блок IF | package main  func main() {  i := 0  if i == 0 {  Print(i)  } else {  Print(100)  }  } | def main():  i = 0  if i==0:  print(i)  else:  print(100) |
| Блок SWITCH | package main  func main() {  i := 0  switch i {  case 1: break  default: break  }  } | def main():  i = 0  match i:  case 1:  break  case \_:  break |
| Блок FOR | package main  func main() {  for i:=0; i<10; i++ {  Print(i)  }  } | def main():  for i in range(0, 10, 1):  print(i) |
| Блок WHILE | package main  func main() {  i := 0  for i<10 {  Print(i)  }  } | def main():  i = 0  while i<10:  print(i) |

## 10.4 Оптимизация кода

|  |  |  |
| --- | --- | --- |
| **Ситуация** | **Входные данные** | **Выходные данные** |
| Константы, переменные, комментарии | package main  // Const  const a int = 1  const a int = 2  const a int = 3  // Var  var b int = 1  var b int = 2  var b int = 3 | E::Redeclaring a variable  ::a  E::Redeclaring a variable  ::a  E::Redeclaring a variable  ::b  E::Redeclaring a variable  ::b  a = 3  b = 3 |
| Функции | func plus(a, b int) int {  return a + b  }  func plus(a, b float64) float64 {  return a + b  } | E::Redeclaring a function  ::plus  def plus(a, b):  return a+b |

# Заключение

В ходе работы над транслятором Golang–Python была проделана следующая работа:

* Определена грамматика входного языка.
* Определены контекстные условия входного (Golang) и выходного (Python) языка.
* Составлена таблица соответствия языков.
* Создан лексический анализатор.
* Создан синтаксический анализатор.
* Создан семантический анализатор.
* Создан генератор кода выходного языка.
* Создан оптимизатор кода.
* Проведено тестирование.