|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ \_\_\_\_\_\_\_\_\_\_\_\_Информатика и системы управления\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_Программное обеспечение ЭВМ и информационные технологии\_\_\_\_

**Расчётно-пояснительная записка**

**к курсовому проекту на тему**

**«?»**

**по курсу «Проектирование компиляторов»**

Студент \_\_\_ИУ7-21М\_\_\_\_ \_\_\_\_Карпухин А.С. \_\_\_\_

(Группа) (И.О.Фамилия)

Студент \_\_\_ИУ7-22М\_\_\_\_ \_\_\_\_Магазинов Н.А.\_\_\_\_

(Группа) (И.О.Фамилия)

Руководитель курсового проекта \_\_\_\_Ступников А.А.\_\_\_\_

(И.О.Фамилия)

*2021 г*

**Оглавление**

[**Введение** 4](#_Toc74526481)

[**1.** **Аналитический раздел** 6](#_Toc74526482)

[**1.1.** **Этапы компиляции** 6](#_Toc74526483)

[**1.2.** **Лексический анализ** 6](#_Toc74526484)

[**1.3.** **Синтаксический анализ** 7](#_Toc74526485)

[**1.3.1.** **Нисходящий разбор** 7](#_Toc74526486)

[**1.3.2.** **Восходящий разбор** 7](#_Toc74526487)

[**1.4.** **Средства построения анализаторов** 8](#_Toc74526488)

[**1.4.1.** **Выбор методов лексического и синтаксического анализа** 8](#_Toc74526489)

[**1.5.** **Формат грамматики целевого языка** 9](#_Toc74526490)

[**1.6.** **Семантический анализ** 10](#_Toc74526491)

[**1.6.1.** **Методы обхода дерева разбора** 10](#_Toc74526492)

[**1.6.1.1.** **Паттерн «Visitor»** 10](#_Toc74526493)

[**1.6.1.2.** **Паттерн «Listener»** 10](#_Toc74526494)

[**1.6.1.3.** **Выбор метода обхода** 11](#_Toc74526495)

[**1.6.2.** **Таблицы символов** 11](#_Toc74526496)

[**1.6.2.1.** **Линейные списки** 11](#_Toc74526497)

[**1.6.2.2.** **Хеш-таблицы** 12](#_Toc74526498)

[**1.6.2.3.** **Выбор реализации таблицы символов** 12](#_Toc74526499)

[**Вывод по разделу** 13](#_Toc74526500)

[**2.** **Конструкторский раздел** 14](#_Toc74526501)

[**2.1. Декомпозиция задачи** 14](#_Toc74526502)

[**3.** **Технологический раздел** 15](#_Toc74526503)

[**Заключение** 16](#_Toc74526504)

[**Список использованных источников** 17](#_Toc74526505)

**Введение**

Scala – мультипарадигменный язык программирования, сочетающий в себе основные черты функционального и объектно-ориентированного подходов. Спроектирован для написания краткого и типобезопасного компонентного программного обеспечения.

Scala является объектно-ориентированным языком в том смысле, что любое значение в программе есть объект, и функциональным языком в том смысле, что любая функция есть значение, то есть элемент данных, который может быть использован также как любая другая переменная.

Любая операция в Scala – это отправка сообщения (как диктует объектно-ориентированный подход в его изначальном виде), то есть вызов какого-либо метода объекта. По этой причине в языке отсутствуют встроенные операторы, а арифметические и логические операции реализованы в виде методов классов.

WebAssembly – формат бинарных инструкций для стековой виртуальной машины, спроектированный как портативная цель компиляции для высокоуровневых языков. Стековая виртуальная машина, исполняющая инструкции бинарного формата wasm, может быть запущена как в среде браузера, так и в серверной среде.

Цель курсовой работы – разработать программное обеспечение, реализующий компилятор языка Scala в язык виртуальной машины WebAssembly.

Для достижения поставленной цели необходимо решить следующие задачи:

* реализовать лексический анализатор;
* реализовать синтаксический анализатор;
* реализовать семантический анализ, по результатам которого получить:
  + таблицы локальных и глобальных символов (классов, объектов, типов, функций, переменных);
  + граф иерархии наследования;
  + граф вызовов.
* Реализовать генератор промежуточного представления;
* Реализовать генератор целевого кода платформы WebAssembly.

1. **Аналитический раздел**
   1. **Этапы компиляции**

Компилятор – программное обеспечение, транслирующее код на исходном языке в эквивалентный код на целевом языке. Основные этапы компиляции приведены ниже на рисунке 1.

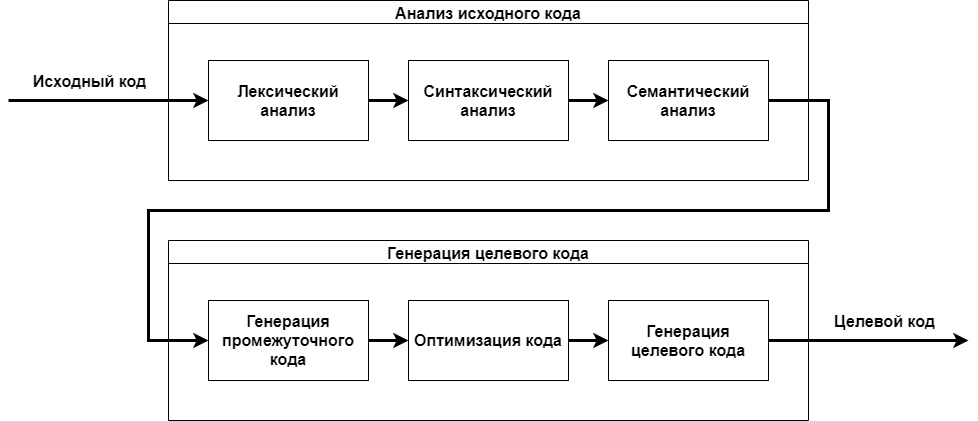


Рисунок 1 – Фазы работы компилятора

В следующих разделах подробно рассмотрены этапы анализа исходного кода.

* 1. **Лексический анализ**

Лексический анализ образует первый этап процесса компиляции. На этом этапе символы, составляющие исходную программу, считываются и группируются в отдельные лексические элементы – лексемы. Лексический анализатор заменяет в программе идентификаторы и константы лексемами, что делает представление программы удобнее для дальнейшей обработки. Так же лексический анализатор устраняет ненужные пробелы и комментарии.

Лексический анализ может быть представлен и как самостоятельная фаза трансляции, и как составная часть фазы синтаксического анализа. В первом случае лексический анализатор реализуется в виде отдельного модуля, который принимает последовательность символов, составляющих текст компилируемой программы, и выдаёт список обнаруженных лексем. Во втором случае лексический анализатор фактически является подпрограммой, вызываемой синтаксическим анализатором для получения очередной лексемы.

В процессе лексического анализа так же выявляются лексические ошибки – ошибки, связанные с наличием недопустимых символов, некорректной записью идентификаторов, строк и т.д.

Результатом работы лексического анализа является цепочка лексем.

* 1. **Синтаксический анализ**

Второй фазой процесса компиляции является фаза синтаксического анализа. Входными данными синтаксического анализатора является результат работы лексического анализатора – цепочка лексем. Синтаксический анализ – разбор, в котором исследуется цепочка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в синтаксисе языка. Результатом работы синтаксического анализатора является дерево разбора, которое представляет синтаксическую структуру исходной программы.

* + 1. **Нисходящий разбор**

В нисходящем анализе дерево вывода цепочки строится от корня к листьям, т.е. нисходящие анализаторы строят вывод, начиная от аксиомы грамматики и заканчивая цепочкой терминальных символов. В общем виде нисходящий анализ представлен в анализе методом рекурсивного спуска, который может использовать откаты, т.е. производить повторный просмотр считанных символов.

* + 1. **Восходящий разбор**

Восходящий анализатор предназначен для построения дерева разбора, начиная с листьев и двигаясь вверх к корню дерева разбора. Можно представить себе этот процесс как "свертку" исходной строки к аксиоме грамматики. Каждый шаг свертки заключается в сопоставлении некоторой подстроки и правой части какого-то правила грамматики, и замене этой подстроки на нетерминал, являющийся левой частью правила. Если на каждом шаге подстрока выбирается правильно, то в результате получается правый вывод строки.

* + 1. **Средства построения анализаторов**

Существует множество различных стандартных средств для построения синтаксических анализаторов: Lex и Yacc, Coco/R, ANTLR и др.

Генератор Yacc (Yet Another Compiler Compiler) - стандартный генератор синтаксических анализаторов в Unix-системах. Yacc генерирует парсер на основе аналитической грамматики, описанной в нотации BNF (форма Бэкуса-Наура) или контекстно-свободной грамматики. Как правило, Yacc используется в связке с Lex –генератором лексических анализаторов.

Генератор lex является стандартным генератором лексических анализаторов в операционных системах Unix.

ANTLR – это генератор синтаксических анализаторов для чтения, обработки или трансляции как структурированных текстовых, так и бинарных файлов. На основе заданной грамматики языка ANTLR генерирует код нисходящего синтаксического анализатора, который может строить абстрактное синтаксического дерево и производить его обход.

* + 1. **Выбор методов лексического и синтаксического анализа**

В данной работе для реализации целевого программного обеспечения ввиду больших объемов грамматики выбранного языка было принято решение использовать генератор лексических и синтаксических анализаторов ANTLR. Результат генерации включает в себя четыре файла на выбранном целевом языке:

* Файл с кодом лексического анализатора;
* Файл с кодом нисходящего синтаксического анализатора;
* Файл с кодом базовой реализации паттерна «Visitor»;
* Файл с кодом базовой реализации паттерна «Listener».

Упомянутые реализации паттернов поведения используются для обхода полученного в результате синтаксического анализа дерева разбора исходного кода компилируемой программы.

* 1. **Формат грамматики целевого языка**

В качестве компилируемого языка был выбран языка Scala. Грамматика языка описана в подходящем для ANTLR формате g4. Структура описания грамматики имеет вид, представленный в листинге 1.1.

Листинг 1.1 – Общий вид грамматики

grammar Name;

rule1

...

ruleN

Правила грамматики начинаются с ":" и заканчиваются ";". Несколько правил разделяются знаком "|". Пример правил языка Scala приведен в листинге 1.2.

Листинг 1.2 – Пример грамматики языка Scala

|  |
| --- |
| expr  : (bindings | 'implicit'? Id | '\_') '=>' expr  | expr1  ;  expr1  : 'if' '(' expr ')' NL\* expr ('else' expr)?  | 'while' '(' expr ')' NL\* expr  | 'try' expr ('catch' expr)? ('finally' expr)?  | 'do' expr 'while' '(' expr ')'  | 'for' ('(' enumerators ')' | '{' enumerators '}') 'yield'? expr  | 'throw' expr  | 'return' expr?  | ((simpleExpr | simpleExpr1 '\_'?) '.')? Id '=' expr  | simpleExpr1 argumentExprs '=' expr  | postfixExpr ascription?  | postfixExpr 'match' '{' caseClauses '}'  ; |

* 1. **Семантический анализ**

В процессе семантического анализа производится проверка на наличие семантических ошибок, а также накапливается информация о типах в программе для следующей стадии – кодогенерации. В качестве входных данных этапа рассматривается дерево разбора, полученное в результате синтаксического анализа.

* + 1. **Методы обхода дерева разбора**

По умолчанию ANTLR реализует два способа обхода дерева разбора, генерируемого в результате синтаксического анализа. Оба метода реализованы с использованием стандартных поведенческих шаблонов проектирования – «Visitor» («Посетитель») и «Listener» («Слушатель»).

* + - 1. **Паттерн «Visitor»**

Паттерн «Посетитель» предполагает определение для набора объектов, принадлежащих различным классам, методов обработки каждого уникального типа в наборе.

В качестве реализации создается базовый класс Visitor с методами Visit() для каждого подкласса родительского Element, описывающего произвольный объект в наборе. В иерархию Element, в свою очередь, добавляется метод Accept(visitor), вызывающий соответствующую реализацию метода Visit() для данного объекта. Для каждой операции, выполняемой на объектах иерархии Element, создается производный от Visitor класс.

* + - 1. **Паттерн «Listener»**

Реализация паттерна «Слушатель» в ANTLR далека от классической, использующей механизм подписчиков и издателей. В данном случае реализация основывается на базовом классе Listener, в котором для каждого класса объекта в наборе создается два метода – Enter и Exit, вызываемые при входе в узел и выходе из узла дерева разбора соответственно. По результатам обхода поддерева текущего узла в методе Exit выполняются некоторые действия.

* + - 1. **Выбор метода обхода**

Для обхода дерева разбора исходного кода был выбран метод, основанный на реализации паттерна «Visitor» ввиду его простоты и эффективности при разборе древовидных структур. В данном случае эта реализация также удобна тем, каждый метод класса Visitor обязан возвращать некоторый результат. Это может быть удобно при выводе типов выражений подобно атрибутивным грамматикам.

* + 1. **Таблицы символов**

Таблица символов, как правило, применяется для хранения информации о различных конструкциях исходного языка. Содержимое таблицы просматривается каждый раз, когда в исходном тексте программы встречается некоторое имя. Каждый вновь объявленный в коде исходного языка именованный символ заносится в таблицу. Структура данных заполняется во время фазы анализа, и ее содержимое может быть использовано в дальнейшем при генерации целевого кода.

Механизм таблицы символов должен обеспечивать эффективный поиск и добавление в таблицу символов.

Таблица символов может служить следующим целям в зависимости от используемого языка:

* Хранить имена всех сущностей в структурированной форме в одном месте;
* Чтобы проверить, была ли объявлена ​​переменная;
* Чтобы реализовать проверку типов, путем проверки присваиваний и выражений в исходном коде семантически правильно;
* Чтобы определить область имени (разрешение области).
  + - 1. **Линейные списки**

Простейшая форма представления структуры данных таблицы символов – линейный список записей. В данном случае используется один или несколько массивов для хранения имен символов и связанной с ними информации. Добавление символов осуществляется в том же порядке, в котором они встречаются в исходном коде. Таким образом, поиск от конца списка к его началу гарантирует, что для заданного имени будет найден соответствующий символ с таким именем, объявленный последним.

В случае, когда таблица содержит *n* записей, операция добавления нового символа без проверки его наличия выполняется за фиксированное время. Если все имена в таблице должны быть уникальны, то для соблюдения данного условия требуется осуществить поиск, время которого пропорционально *n*.

В общем случае время, необходимое для внесения *n* имен и выполнения *e* запросов к таблице символов, не превышает *cn(n + e),* где константа *c* представляет собой машинно-зависимую величину, определяемую временем выполнения нескольких машинных операций.

* + - 1. **Хеш-таблицы**

Схема, основанная на открытом хешировании, то есть без ограничения на количество элементов в таблице, позволяет выполнить *e* запросов на *n* имен за время *n(n + e)/m,* где константа *m* может быть определена вручную сколь угодно большой вплоть до *n*. При этом память, требуемая для размещения структуры данных, растет с увеличением *m*.

Структура данных включает в себя две части – хеш-таблицу и блоки. Хеш таблица представляет фиксированный массив из *m* указателей на записи таблицы. Записи организованы в виде *m* связных списков, именуемых блоками. Каждая таблицы запись встречается только в одном из списков. Для определения наличия в таблице записи с именем *s* вычисляется хеш-функция *h(s),* результат которой определяет номер блока в хеш-таблице.

* + - 1. **Выбор реализации таблицы символов**

Для реализации таблицы символов были выбраны хеш-таблицы, позволяющие обеспечить более высокое быстродействие за счет снижения времени выполнения операций добавления символа в таблице и поиска по имени.

При этом следует учитывать, что данных подход требует больше памяти для размещения структуры данных таблицы символов, что может быть несущественно в небольших программах, однако может вызывать проблемы в крупных объемах кода.

**Вывод по разделу**

В данном разделе были приведены общие сведения об основных этапах компиляции. Для каждого из этапов были рассмотрены возможные методы его реализации.

Для генерации лексического и синтаксического анализаторов был выбран набор инструментов ANTLR, порождающий нисходящий синтаксический анализатор.

Для обхода дерева разбора, генерируемого в результате синтаксического анализа, был выбран подход на основе паттерна Visitor, базовая реализация которого также присутствует в генерируемом ANTLR коде.

В качестве структуры данных при реализации таблицы символов для анализируемого программного кода были выбраны хеш-таблицы, встроенная реализация для которых присутствует практически во всех стандартных библиотеках наиболее популярных языков программирования.

1. **Конструкторский раздел**

**2.1. Декомпозиция задачи**

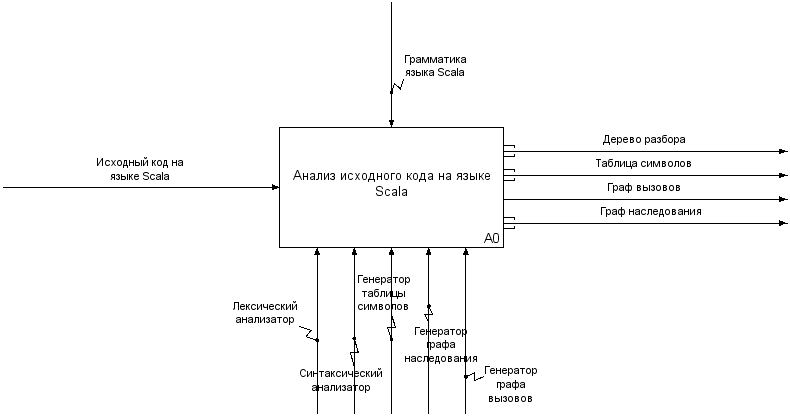


Рисунок 2.1 – Диаграмма процесса анализа исходного кода верхнего уровня

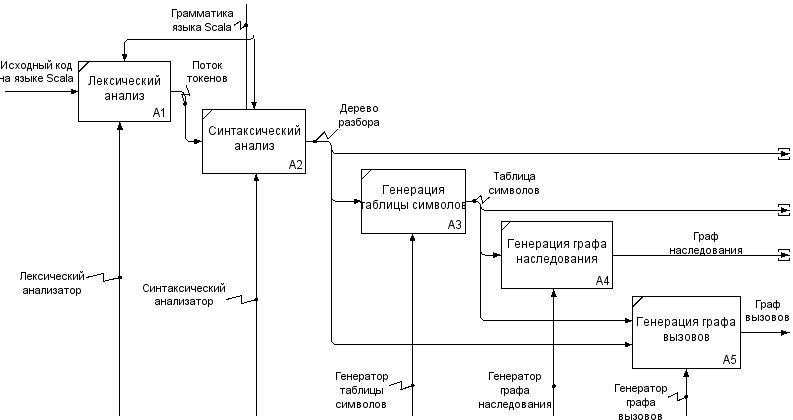


Рисунок 2.2 – Поэтапная декомпозиция процесса анализа исходного кода

1. **Технологический раздел**

**Заключение**

**Список использованных источников**