|  |  |
| --- | --- |
|  | **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\_\_

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОМУ ПРОЕКТУ***

***НА ТЕМУ:***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Компилятор языка Scala\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

***\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

Студент \_\_\_ИУ7-21М\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_А.С. Карпухин\_\_**

(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсового проекта **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_А.А. Ступников**\_\_**\_**

(Подпись, дата) (И.О.Фамилия)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*2021 г.*

Оглавление

[Введение 4](#_Toc74946057)

[1. Аналитический раздел 6](#_Toc74946058)

[1.1. Этапы компиляции 6](#_Toc74946059)

[1.2. Лексический анализ 6](#_Toc74946060)

[1.3. Синтаксический анализ 7](#_Toc74946061)

[1.3.1. Нисходящий разбор 7](#_Toc74946062)

[1.3.2. Восходящий разбор 7](#_Toc74946063)

[1.3.3. Средства построения анализаторов 8](#_Toc74946064)

[1.3.4. Выбор методов лексического и синтаксического анализа 8](#_Toc74946065)

[1.4. Формат грамматики целевого языка 9](#_Toc74946066)

[1.5. Семантический анализ 10](#_Toc74946067)

[1.5.1. Методы обхода дерева разбора 10](#_Toc74946068)

[1.5.1.1. Паттерн «Visitor» 10](#_Toc74946069)

[1.5.1.2. Паттерн «Listener» 10](#_Toc74946070)

[1.5.1.3. Выбор метода обхода 11](#_Toc74946071)

[1.5.2. Таблицы символов 11](#_Toc74946072)

[1.5.2.1. Линейные списки 12](#_Toc74946073)

[1.5.2.2. Хеш-таблицы 12](#_Toc74946074)

[1.5.2.3. Выбор реализации таблицы символов 13](#_Toc74946075)

[Вывод по разделу 13](#_Toc74946076)

[2. Конструкторский раздел 14](#_Toc74946077)

[2.1. Декомпозиция задачи 14](#_Toc74946078)

[2.2. Конструкции языка Scala 15](#_Toc74946079)

[2.3. Семантический анализ 16](#_Toc74946080)

[2.3.1. Построение таблицы символов 16](#_Toc74946081)

[2.3.2. Построение графа наследования 17](#_Toc74946082)

[2.3.3. Вывод типов выражений 18](#_Toc74946083)

[2.3.4. Построение графа вызовов 20](#_Toc74946084)

[Вывод по разделу 21](#_Toc74946085)

[3. Технологический раздел 22](#_Toc74946086)

[3.1. Формат входных и выходных данных 22](#_Toc74946087)

[3.2. Восстановление после ошибок 22](#_Toc74946088)

[3.3. Результаты работы ПО 22](#_Toc74946089)

[3.3.1. Дерево разбора 22](#_Toc74946090)

[3.3.2. Таблица символов 24](#_Toc74946091)

[3.3.3. Граф наследования 26](#_Toc74946092)

[3.3.4. Граф вызовов 26](#_Toc74946093)

[Заключение 28](#_Toc74946094)

[Список использованных источников 29](#_Toc74946095)

[Приложение А 30](#_Toc74946096)

**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 [1].

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 [2].

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.

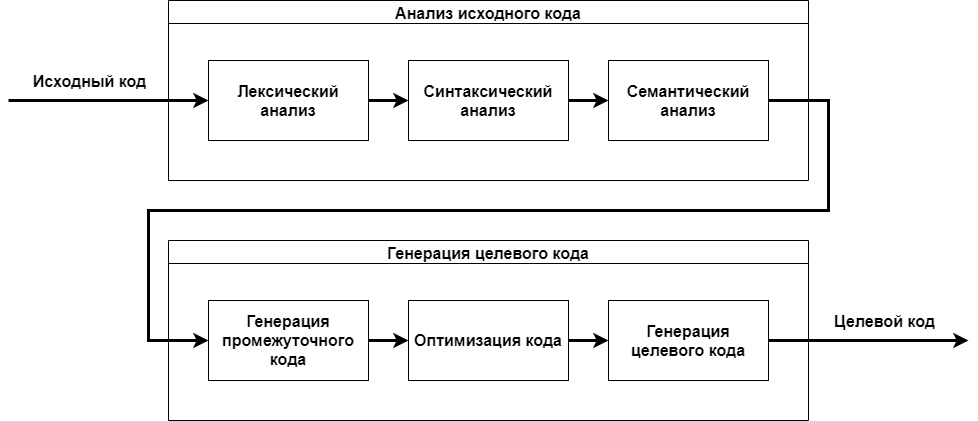


Рисунок 1 – Фазы работы компилятора

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 [3].

* 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 [4].

* + 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. Полная грамматика языка приведена в приложении А в листинге А.1.

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:

* хранение имен всех сущностей в структурированной форме в единой области памяти;
* проверка наличия объявления/определения в случае, когда объявление/определение должно предшествовать использованию;
* проверка типов;
* разрешение областей видимости определения/объявления символов.
  + - 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**

В данном разделе приведены диаграммы IDEF0 процесса анализа исходного кода программы, а также схемы алгоритмов основных этапов семантического анализа.

**2.1. Декомпозиция задачи**

Диаграмма верхнего уровня для процесса анализа исходного кода на языке Scala разрабатываемым ПО приведена на рисунке 2.1.

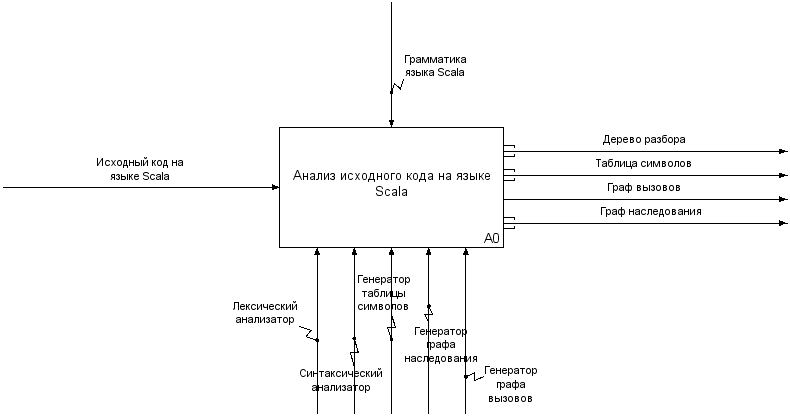


Рисунок 2.1 – Диаграмма процесса анализа исходного кода верхнего уровня

На рисунке 2.2. приведена детальная диаграмма анализа исходного кода с указанием основных этапов:

* лексический анализ;
* синтаксический анализ;
* семантический анализ:
  + генерация таблицы символов;
  + генерация графа наследования;
  + генерация графа вызовов.

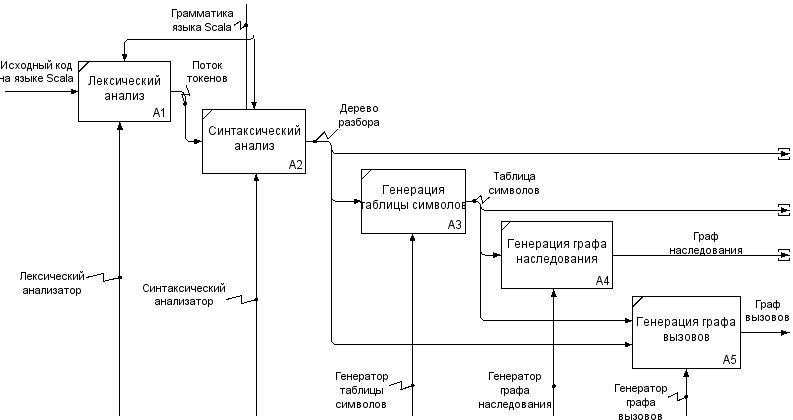


Рисунок 2.2 – Поэтапная декомпозиция процесса анализа исходного кода

**2.2. Конструкции языка Scala**

В разрабатываемом программном обеспечении для реализации были выбраны следующие основные конструкции языка Scala:

* классы (без case и implicit);
* объекты;
* модификаторы доступа;
* наследование (без использования механизмов trait);
* перегрузки функций;
* вложенные объявления классов, объектов и функций;
* инфиксная нотация вызовов функций (для арифметических и логических операций).

Реализуемая иерархия стандартных классов приведена на рисунке 2.3. Структура упрощена относительно оригинальной версии языка, а также добавлен тип String, наследуемый от абстрактного класса AnyVal, так как в исходном варианте для строк используется класс java.lang.String, а реализуемое ПО не предполагает использования виртуальной машины Java для исполнения скомпилированного кода.

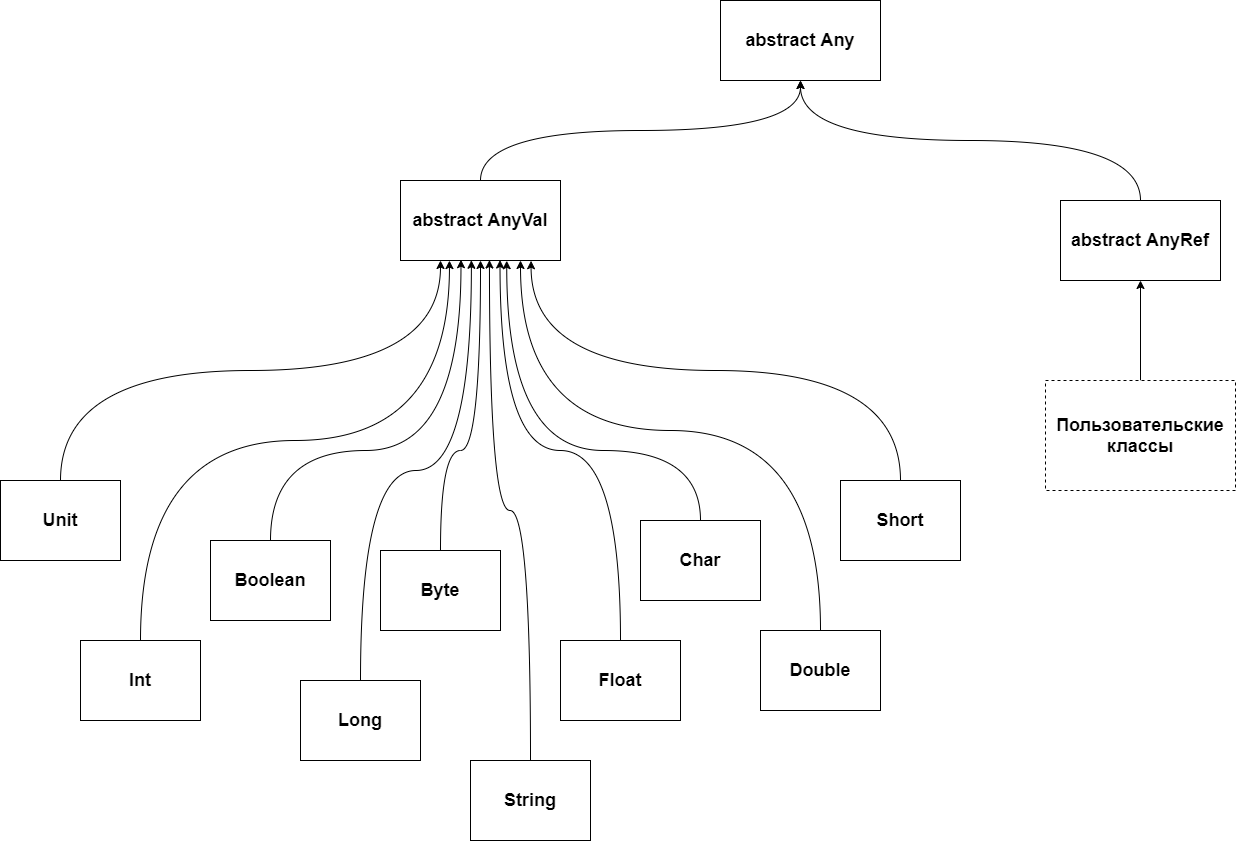


Рисунок 2.3 – Реализуемая иерархия стандартных классов

**2.3. Семантический анализ**

**2.3.1. Построение таблицы символов**

Построение таблицы всех символов, объявляемых или определяемых в программе, выполняется на основе дерева разбора, полученного в результате синтаксического анализа.

Алгоритм заполнения таблицы основан на рекурсивном обходе дерева разбора с использованием паттерна Visitor. Схема алгоритма приведена на рисунке 2.4.

Для учета областей видимости определения символов используется стек, вершина которого является текущей областью видимости. Таким образом, все определения, найденные во время нахождения данной области на вершине стека, считаются вложенными в нее. При этом для определений символов классов, объектов и функций создаются и заносятся в стек вложенные области видимости, после чего выполняется обход поддерева тела функции, класса или объекта с занесением всех найденных объявлений и определений символов во вложенную область видимости.

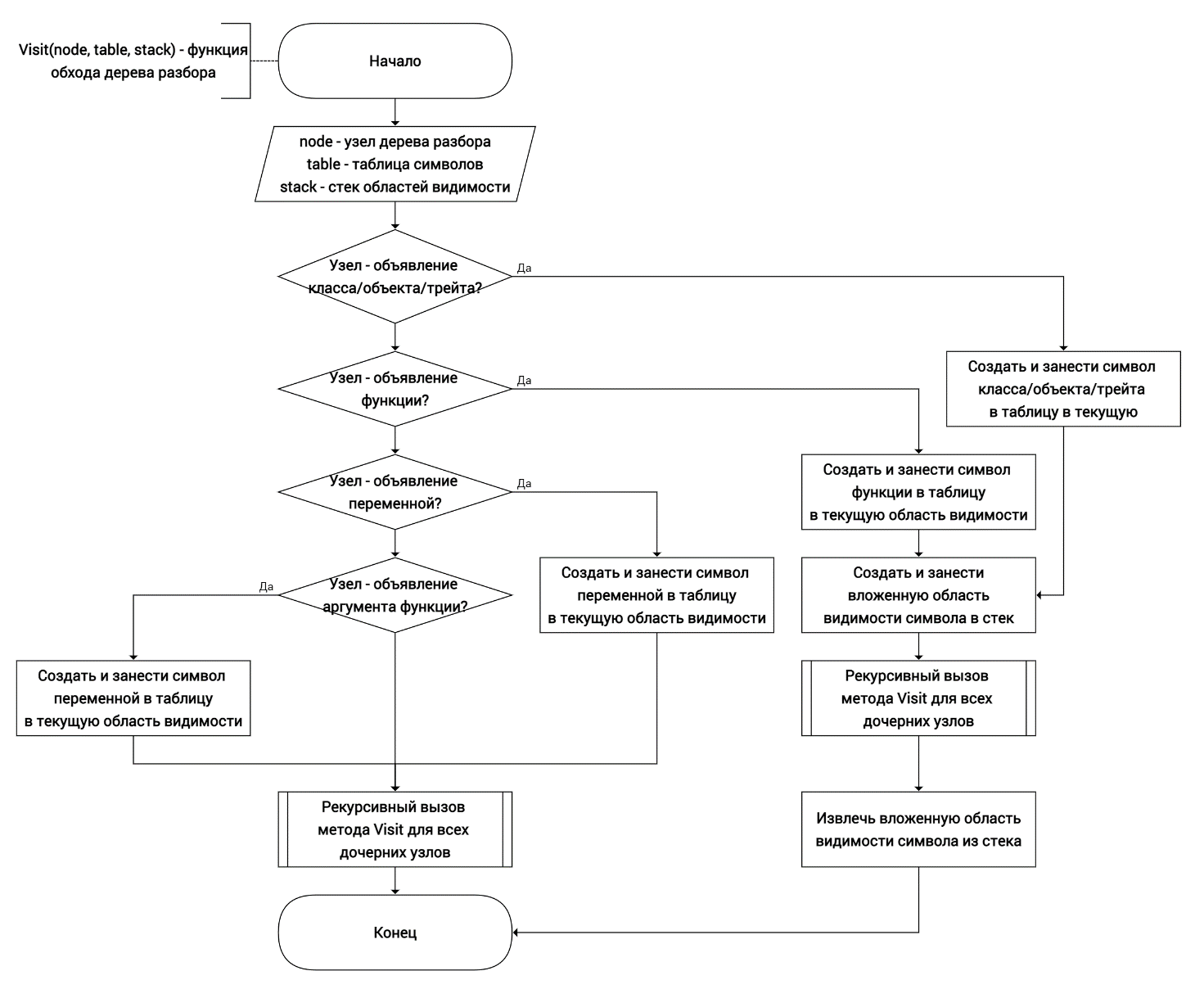


Рисунок 2.4 – Схема рекурсивного алгоритма построения таблицы символов

**2.3.2. Построение графа наследования**

Граф иерархии наследования классов строится на основе заполненной на предыдущем этапе таблицы символов. Для этого каждый дескриптор класса или объекта хранит в поле Parent ссылку на дескриптор родителя. При этом имя родительского класса в коде программы может быть псевдонимом типа, и тогда ссылка на родителя будет указывать не на дескриптор родительского класса, а на дескриптор псевдонима типа. В таком случае необходимо получить из дескриптора псевдонима ссылку на дескриптор класса путем обхода всей цепочки псевдонимов от данного к фактически подразумеваемому имени класса.

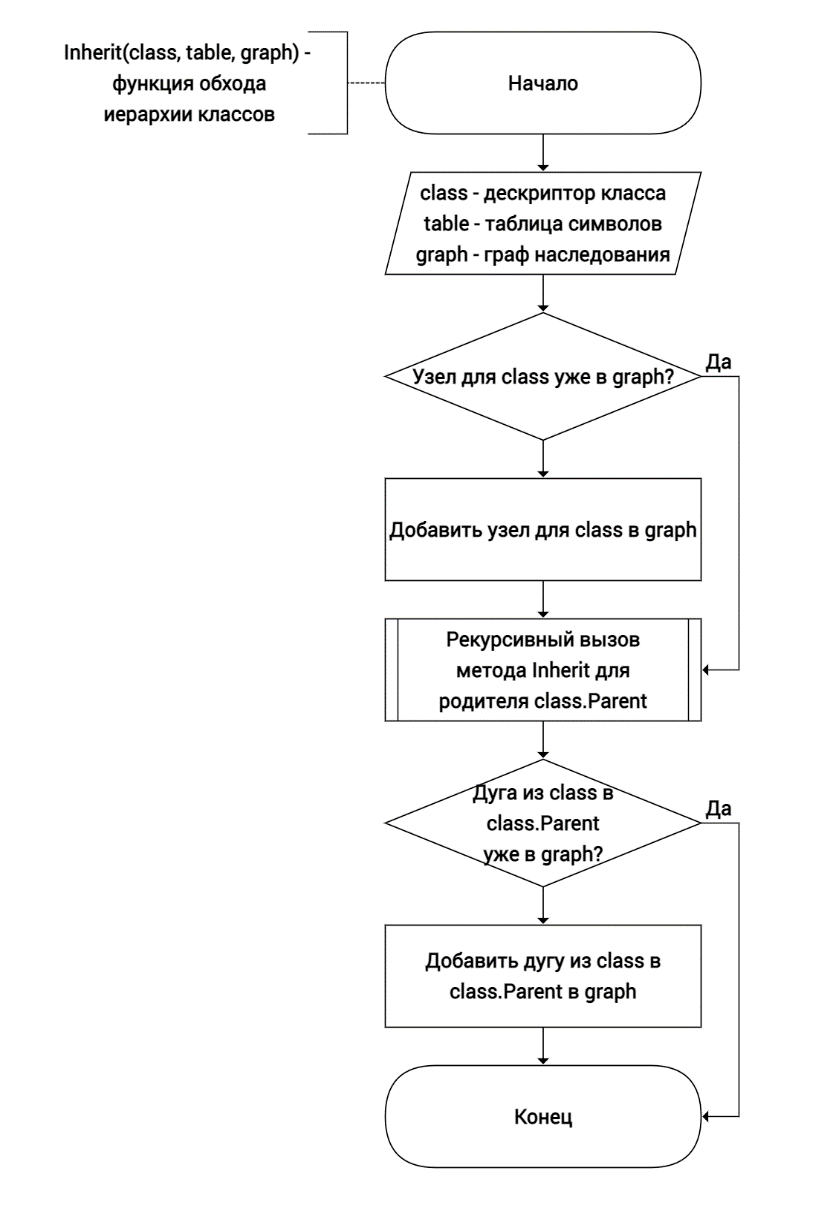


Рисунок 2.5 – Схема рекурсивного алгоритма построения графа наследования

**2.3.3. Вывод типов выражений**

В Scala существуют следующие типы выражений:

* префиксные;
* инфиксные;
* постфиксные;
* блочные;
* смешанные.

Были сделаны следующие допущения относительно синтаксиса выражений:

* постфиксная нотация считается нежелательной и должна быть явно импортирована для ее использования, поэтому в разрабатываемом ПО поддержка постфиксных выражений не реализована;
* методы apply объектов не могут быть вызваны без явного указания имени метода;
* методы, не принимающие ни одного аргумента, не могут быть вызваны без указания пустых круглых скобок;
* не допускается опускание оператора **return** в конце функции, возвращающей значение, тип которого отличен он **Unit**.

Для вывода типа результата каждого из четырех видов выражений, кроме постфиксных, используется отдельный класс:

* **PrefixExprTypeDeductor** – вывод типа результата префиксного выражения;
* **InfixExprTypeDeductor** – вывод типа результата инфиксного выражения;
* **BlockExprTypeDeductor** – вывод типа результата блочного выражения;
* **ExprTypeDeductor** – вывод типа результата произвольного выражения.

Последний использует все остальные для вывода типов результатов подвыражений в частных случаях. Общая схема алгоритма вывода типа для выражения смешанного типа с использованием механизма **ExprTypeDeductor** приведена на рисунке 2.6.

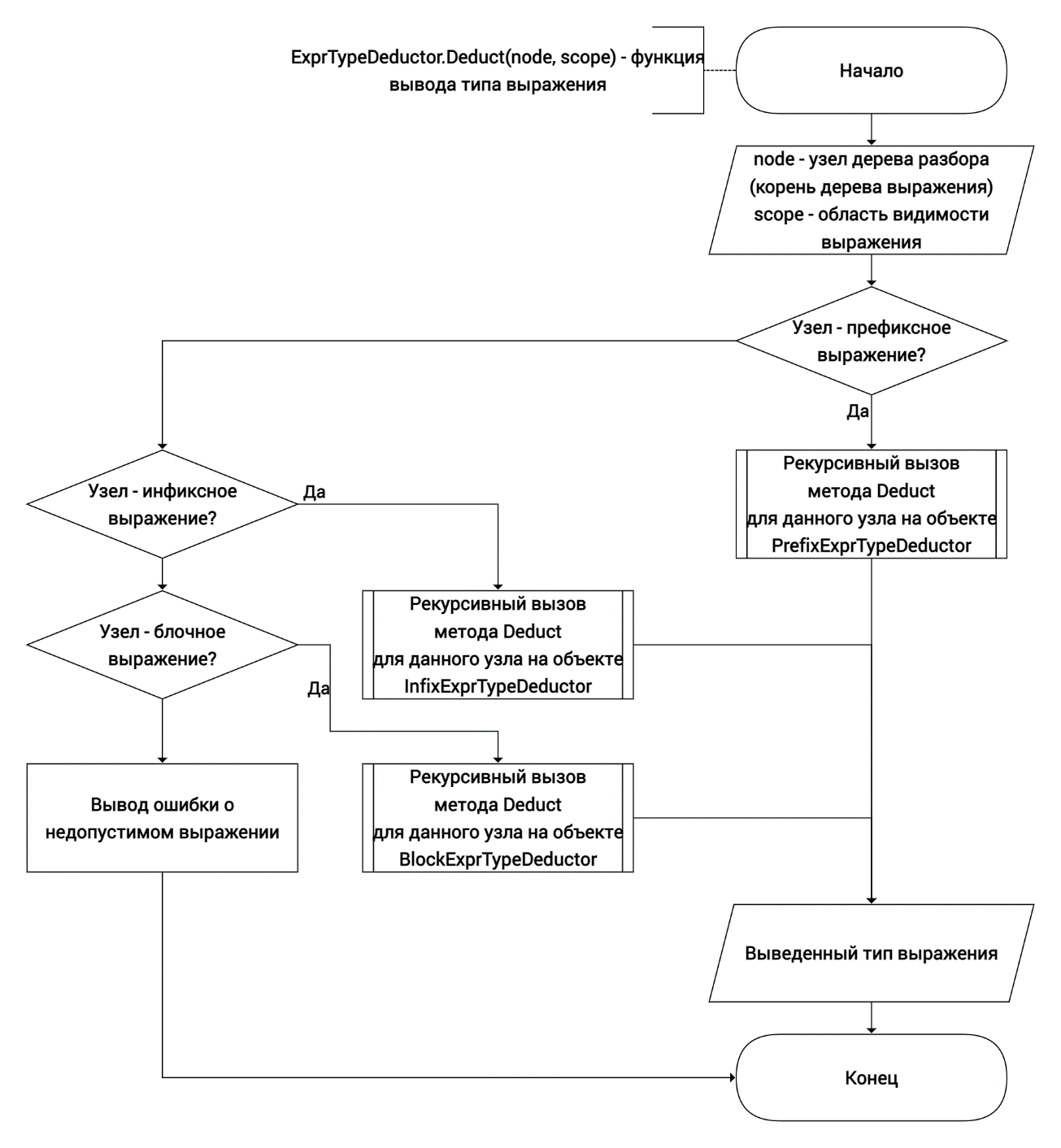


Рисунок 2.6 – Схема алгоритма вывода типа результата смешанного выражения

**2.3.4. Построение графа вызовов**

Граф вызовов строится для конкретной на основе дерева разбора определения этой функции и предварительно заполненной таблицы символов.

Вызовы накапливаются в процессе проверки типов выражений смешанного вида, коими являются определения функций. В процессе вывода типа результата каждого выражения все встреченные вызовы функций заносятся в дерево вызовов в качестве дочерних элементов текущего узла. Затем выполняется рекурсивный разбор определений каждой из найденных функций по тому же принципу. При этом функция может не иметь определения, и тогда рекурсивный вызов процедуры разбора для нее произведен не будет.

**Вывод по разделу**

В данном разделе была приведен общая схема анализа программного кода на языке Scala с использованием разрабатываемого ПО, а также подробная декомпозиция с указанием основных этапов анализа.

Помимо этого, были описаны алгоритмы реализации основных задач семантического анализа, а именно:

* построение глобальных и локальных таблиц символов;
* построение иерархии наследования;
* проверка и вывод типов выражений;
* построение графов вызовов функций.

1. **Технологический раздел**

В данном разделе описываются особенности реализации ПО, а также приводятся результаты работы для различных входных данных.

**3.1. Формат входных и выходных данных**

На вход разработанному ПО подается файл с кодом на языке Scala. В результате работы программа формирует следующие результаты:

* дерево разбора в формате DOT;
* таблица символов в формате DOT;
* граф наследования в формате DOT;
* граф вызовов в формате DOT для выбранной функции.

**3.2. Восстановление после ошибок**

В реализованном ПО была использована стратегия восстановления в режиме паники. Согласно ей, анализатор при обнаружении ошибки пропускает входные символы до некоторого синхронизирующего токена, в качестве которого рассматривается начало следующего выражения. Таким образом, если в ходе разбора некоторого определения возникла ошибка, данное определение отбрасывается, а соответствующий символ, если таковой имеется, не заносится в таблицу. При этом разбор остальных выражений продолжается с целью выявить как можно больше ошибок в исходном коде.

**3.3. Результаты работы ПО**

**3.3.1. Дерево разбора**

В листинге 3.1. приведен код, на основе которого ПО строит дерево разбора и сериализует его в формат DOT. Результат визуализации полученного дерева разбора приведен на рисунке 3.1.

Листинг 3.1. Исходный код на Scala для генерации дерева разбора

|  |
| --- |
| class A {  private var a: Int = 90  def f(b: Double): Double = ???  } |

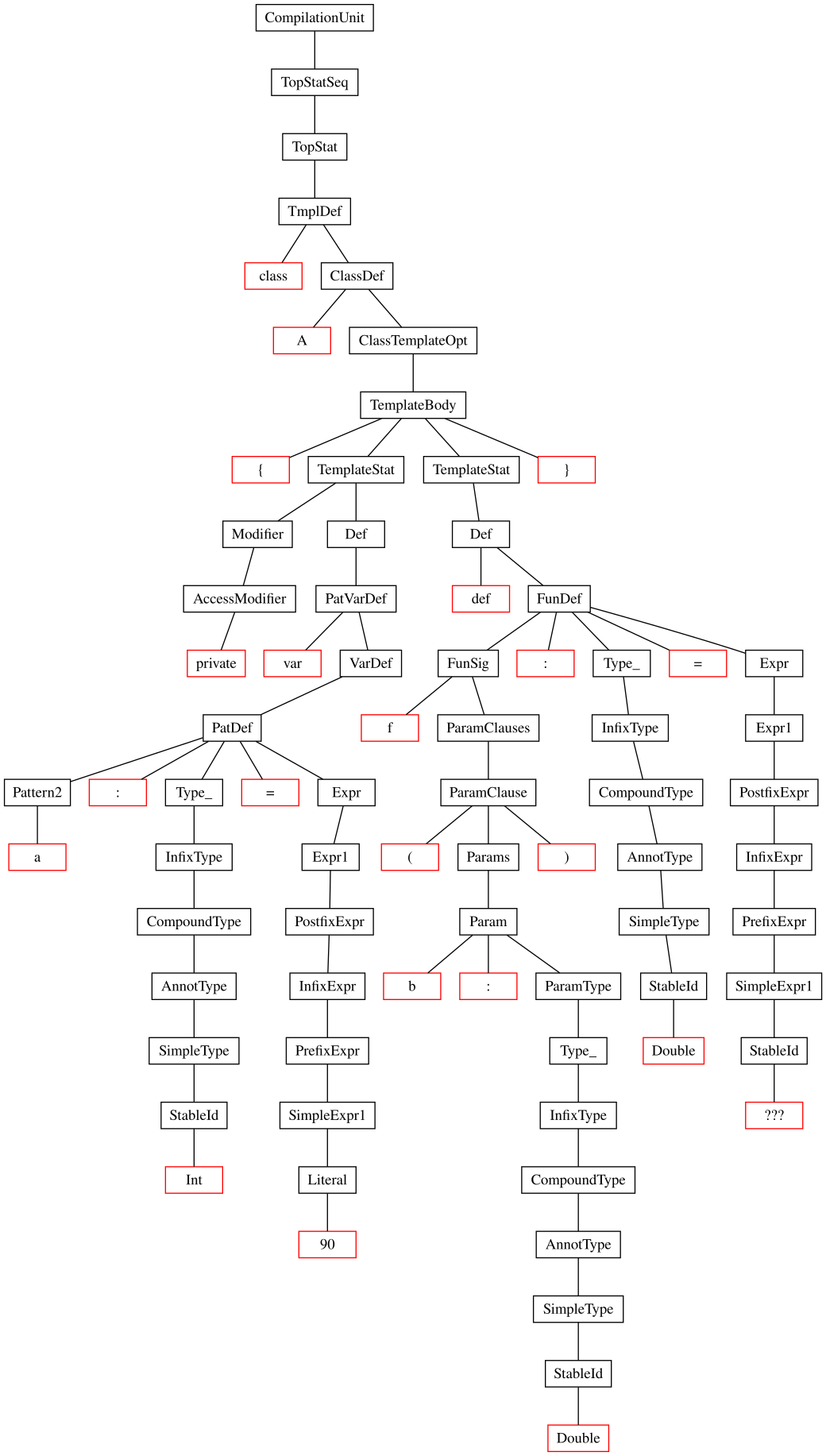


Рисунок 3.1 – Дерево разбора исходного кода Scala

**3.3.2. Таблица символов**

Таблица символов строится в виде древовидной структуры, в которой узлы представляют собой таблицы локальных символов для текущей области видимости. Корневым элементом является таблица глобальных символов, которая помимо пользовательских классов и объектов включает в себя также стандартные типы языка Scala.

Для каждого класса в области видимости его определения также определяется функция с именем класса, представляющая собой конструктор данного класса.

Для каждой функции, в том числе для конструкторов классов, аргументы во вложенной области видимости дублируются в виде локальных переменных. При этом локальные переменные, являющиеся прообразами аргументов функции, получают модификатор **val**, то есть являются неизменяемыми.

В листинге 3.2 содержится код, для которого на рисунке 3.2 приведена визуализация таблицы всех символов, определенных в данной программе. При этом глобальная область видимости также включает в себя классы стандартных типов и их конструкторы.

Листинг 3.2. Исходный код на Scala для генерации таблицы символов

|  |
| --- |
| abstract class A {  var a = "string"  private val b: Int = 90  def f(): Unit  private def f(b: Double): Double = ???  protected def g(x: Any): Unit = {  def inner(): Int = 100  val y: Double = 1.1  }  }  abstract class B extends A {  def f(): Unit = ???  val x: Any  val l = g(x)  } |

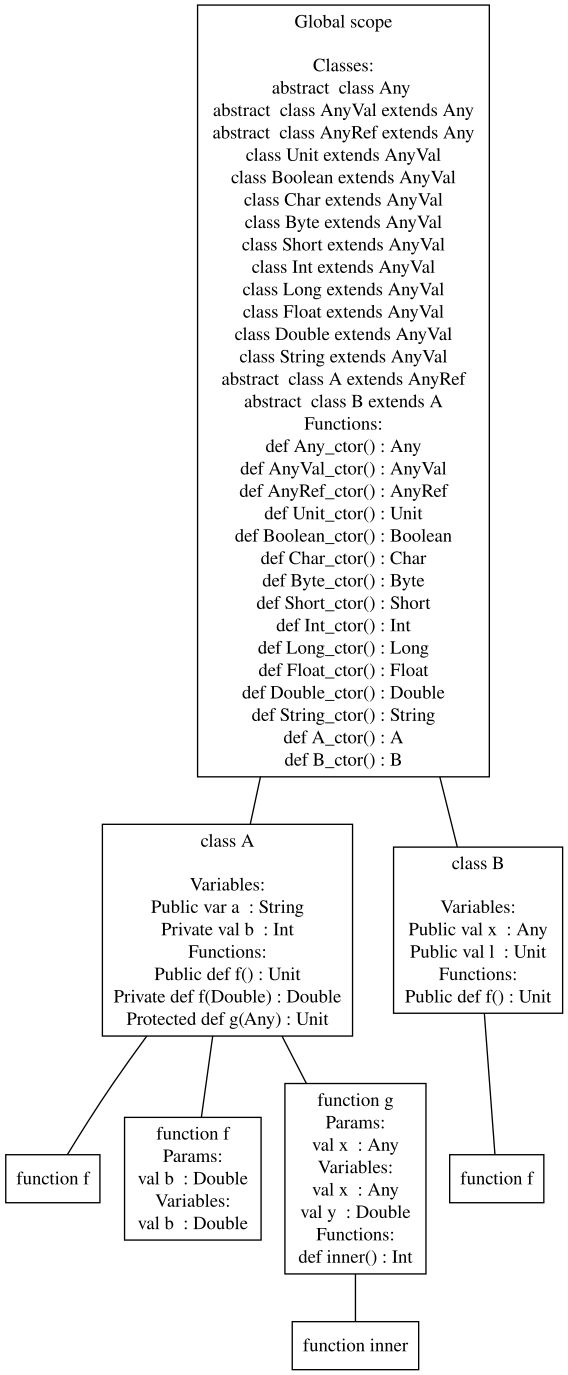


Рисунок 3.2 – Таблица символов для исходного кода Scala

**3.3.3. Граф наследования**

Граф наследования отображает иерархию наследования классов в программе и строится на основе заполненной таблицы символов. Диаграмма наследования для кода из листинга 3.3 приведена на рисунке 3.3.

Листинг 3.3. Исходный код на Scala для построения графа наследования

|  |
| --- |
| class A  class B extends A  class C extends A  class D extends B  class E extends C  class Class extends AnyRef |

Как видно из изображения, все пользовательские классы прямо или косвенно наследуют **AnyRef**, который, в свою очередь, наследует **Any**. При этом даже если для пользовательского класса явно не указан родитель, им по умолчанию будет **AnyRef**.

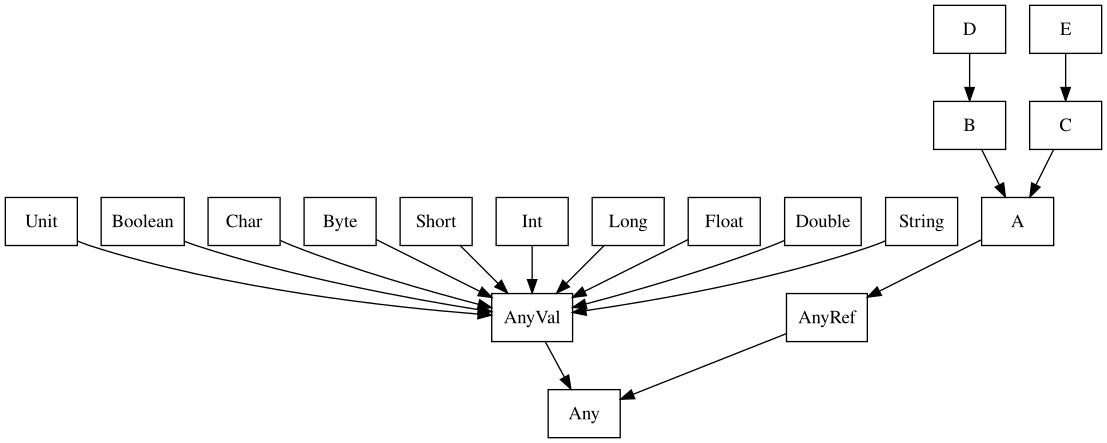


Рисунок 3.3 – Диаграмма наследования для кода из листинга 3.3

**3.3.4. Граф вызовов**

Граф вызовов строится на основе механизма проверки типов с использованием дерева разбора и заполненной таблицы символов программы. Он включает в себя помимо пользовательских методов также методы стандартных классов Scala.

Граф вызовов для функции **r()** класса **B**, определенных в листинге 3.4, приведен на рисунке 3.4.

Листинг 3.4 - Исходный код на Scala для построения графа вызовов

|  |
| --- |
| class A {  def f(): Unit = ???  private def f(b: Double): Double = ???  protected def f(b: Int): Double = {  val y = f(b)  val c = y \*\* f(y)  return y ++ b  }  }  abstract class B extends A {  def g(j: Int): Double = f(j)  def r(): Unit = {  val t = g(100)  }  } |

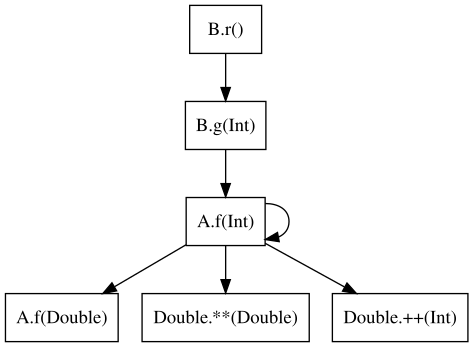


Рисунок 3.4 – Граф вызовов для функции **r()** класса **B** из листинга 3.4

**Заключение**

В ходе выполнения курсовой работы были рассмотрены основные этапы процесса компиляции и проведен обзор основных алгоритмов их реализации.

В результате выполнения работы был реализован компилятор языка Scala, выполняющий лексический, синтаксический и семантический анализ исходного кода на целевом языке, а также производящий на выходе:

* дерево разбора;
* таблицы глобальных и локальных символов;
* граф иерархии наследования;
* графы вызовов выбранных функций.

Для реализации этапов лексического и синтаксического анализа были использованы анализаторы, созданные при помощи генератора ANTLR4.

Все алгоритмы этапа семантического анализа, использующие обход дерева разбора, были реализованы с использованием паттерна Visitor.

Алгоритм формирования таблиц символов был выполнен на основе хеш-таблиц и стека областей видимости символов.

Построение графов наследования и вызовов функций реализовано на основе обхода дерева разбора и анализа информации в таблицах символов.

В качестве стратегии восстановления после ошибок использован подход на основе режима паники с целью выявить максимально возможное число ошибок в исходном коде.

**Список использованных источников**

1. Scala Documentation. – Режим доступа: <https://docs.scala-lang.org/>, свободный (Дата обращения: 20.05.2021).
2. Specifications – WebAssembly. - Режим доступа: [https://webassembly.github.io/spec/core/intro/introduction.html#id1](https://webassembly.github.io/spec/core/intro/introduction.html" \l "id1), свободный (Дата обращения: 20.05.2021).
3. Ахо А.В., Лам М. С., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. –М.: ООО Изд. Дом Вильямс, 2008. –1184 с.
4. Aхо А., Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Том 1. Синтаксический анализ. –М.: Мир, 1978. –613 с.

**Приложение А**

Листинг А.1. Грамматика языка Scala

|  |
| --- |
| /\*  [The "BSD licence"]  Copyright (c) 2014 Leonardo Lucena  Copyright (c) 2018 Andrey Stolyarov  All rights reserved.  Redistribution and use in source and binary forms, with or without  modification, are permitted provided that the following conditions  are met:  1. Redistributions of source code must retain the above copyright  notice, this list of conditions and the following disclaimer.  2. Redistributions in binary form must reproduce the above copyright  notice, this list of conditions and the following disclaimer in the  documentation and/or other materials provided with the distribution.  3. The name of the author may not be used to endorse or promote products  derived from this software without specific prior written permission.  THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR  IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES  OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,  INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT  NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF  THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.  \*/  /\*  Derived from https://github.com/scala/scala/blob/2.12.x/spec/13-syntax-summary.md  \*/  grammar Scala;  literal  : '-'? IntegerLiteral  | '-'? FloatingPointLiteral  | BooleanLiteral  | CharacterLiteral  | StringLiteral  | SymbolLiteral  | 'null';  qualId : Id ('.' Id)\* ;  Ids : Id (',' Id)\* ;  stableId  : Id  | stableId '.' Id  | (Id '.')? ('this' | 'super' classQualifier? '.' Id);  classQualifier : '[' Id ']' ;  type\_  : functionArgTypes '=>' type\_  | infixType existentialClause? ;  functionArgTypes  : infixType  | '(' (paramType (',' paramType)\*)? ')' ;  existentialClause  : 'forSome' '{' existentialDcl+ '}' ;  existentialDcl  : 'type' typeDcl  | 'val' valDcl ;  infixType : compoundType (Id compoundType)\* ;  compoundType  : annotType ('with' annotType)\* refinement?  | refinement ;  annotType : simpleType annotation\* ;  simpleType  : simpleType typeArgs  | simpleType '#' Id  | stableId ('.' 'type')?  | '(' types ')' ;  typeArgs : '[' types ']' ;  types : type\_ (',' type\_)\* ;  refinement : NL? '{' refineStat+ '}' ;  refineStat  : dcl  | 'type' typeDef ;  typePat : type\_ ;  ascription  : ':' infixType  | ':' annotation +  | ':' '\_' '\*' ;  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 '}' ;  prefixDef : '-' | '+' | '~' | '!' ;  postfixExpr : infixExpr Id? (prefixDef simpleExpr1)\* NL? ;  infixExpr  : prefixExpr  | infixExpr Id NL? infixExpr ;  prefixExpr : prefixDef? (simpleExpr | simpleExpr1 '\_'?) ;  simpleExpr  : 'new' (classTemplate | templateBody)  | blockExpr ;  // Dublicate lines to prevent left-recursive code.  // can't use (simpleExpr|simpleExpr1) '.' Id  simpleExpr1  : literal  | stableId  | '\_'  | '(' exprs? ')'  | simpleExpr '.' Id  | simpleExpr1 '\_'? '.' Id  | simpleExpr typeArgs  | simpleExpr1 '\_'? typeArgs  | simpleExpr1 argumentExprs  ;  exprs : expr (',' expr)\* ;  argumentExprs  : '(' args ')'  | '{' args '}'  | NL? blockExpr ;  args  : exprs?  | (exprs ',')? postfixExpr (':' | '\_' | '\*') ? ;  blockExpr  : '{' caseClauses '}'  | '{' block '}' ;  block : blockStat+ resultExpr? ;  blockStat  : import\_  | annotation\* ('implicit' | 'lazy')? def  | annotation\* localModifier\* tmplDef  | expr1  ;  resultExpr  : expr1  | (bindings | ('implicit'? Id | '\_') ':' compoundType) '=>' block ;  enumerators : generator+ ;  generator : pattern1 '<-' expr (guard | pattern1 '=' expr)\* ;  caseClauses : caseClause + ;  caseClause : 'case' pattern guard? '=>' block ;  guard : 'if' postfixExpr ;  pattern : pattern1 ('|' pattern1)\* ;  pattern1  : (BoundVarid| '\_' | Id) ':' typePat  | pattern2 ;  pattern2  : Id ('@' pattern3)?  | pattern3 ;  pattern3  : simplePattern  | simplePattern (Id NL? simplePattern)\* ;  simplePattern  : '\_'  | Varid  | literal  | stableId ('(' patterns? ')')?  | stableId '(' (patterns ',')? (Id '@')? '\_' '\*' ')'  | '(' patterns? ')' ;  patterns  : pattern (',' patterns)?  | '\_' '\*' ;  typeParamClause : '[' variantTypeParam (',' variantTypeParam)\* ']' ;  funTypeParamClause : '[' typeParam (',' typeParam)\* ']' ;  variantTypeParam : annotation\* ('+' | '-')? typeParam ;  typeParam  : (Id | '\_') typeParamClause? ('>:' type\_)? ('<:' type\_)? ('<%' type\_)\* (':' type\_)\* ;  paramClauses : paramClause\* (NL? '(' 'implicit' params ')')? ;  paramClause : NL? '(' params? ')' ;  params : param (',' param)\* ;  param : annotation\* Id (':' paramType)? ('=' expr)? ;  paramType  : type\_  | '=>' type\_  | type\_ '\*' ;  classParamClauses : classParamClause\* (NL? '(' 'implicit' classParams ')')? ;  classParamClause : NL? '(' classParams? ')' ;  classParams : classParam (',' classParam)\* ;  classParam  : annotation\* modifier\* ('val' | 'var')? Id ':' paramType ('=' expr)? ;  bindings : '(' binding (',' binding)\* ')' ;  binding : (Id | '\_') (':' type\_)? ;  modifier  : localModifier  | accessModifier  | 'override' ;  localModifier  : 'abstract'  | 'final'  | 'sealed'  | 'implicit'  | 'lazy' ;  accessModifier : ('private' | 'protected') accessQualifier? ;  accessQualifier : '[' (Id | 'this') ']' ;  annotation : '@' simpleType argumentExprs\* ;  constrAnnotation : '@' simpleType argumentExprs ;  templateBody : NL? '{' selfType? templateStat+ '}' ;  templateStat  : import\_  | (annotation NL?)\* modifier\* def  | (annotation NL?)\* modifier\* dcl  | expr ;  selfType  : Id (':' type\_)? '=>'  | 'this' ':' type\_ '=>' ;  import\_ : 'import' importExpr (',' importExpr)\* ;  importExpr : stableId ('.' (Id | '\_' | importSelectors))? ;  importSelectors : '{' (importSelector ',')\* (importSelector | '\_') '}' ;  importSelector : Id ('=>' (Id | '\_'))? ;  dcl  : 'val' valDcl  | 'var' varDcl  | 'def' funDcl  | 'type' NL\* typeDcl ;  valDcl : ids ':' type\_ ;  varDcl : ids ':' type\_ ;  funDcl : funSig (':' type\_)? ;  funSig : Id funTypeParamClause? paramClauses ;  typeDcl : Id typeParamClause? ('>:' type\_)? ('<:' type\_)? ;  patVarDef  : 'val' patDef  | 'var' varDef ;  def  : patVarDef  | 'def' funDef  | 'type' NL\* typeDef  | tmplDef ;  patDef : pattern2 (',' pattern2)\* (':' type\_)? '=' expr ;  varDef  : patDef  | ids ':' type\_ '=' '\_' ;  funDef  : funSig (':' type\_)? '=' expr  | funSig NL? '{' block '}'  | 'this' paramClause paramClauses ('=' constrExpr | NL? constrBlock) ;  typeDef : Id typeParamClause? '=' type\_ ;  tmplDef  : 'case'? 'class' classDef  | 'case'? 'object' objectDef  | 'trait' traitDef ;  classDef : Id typeParamClause? constrAnnotation\* accessModifier? classParamClauses classTemplateOpt ;  traitDef : Id typeParamClause? traitTemplateOpt ;  objectDef : Id classTemplateOpt ;  classTemplateOpt  : 'extends' classTemplate  | ('extends'? templateBody)? ;  traitTemplateOpt  : 'extends' traitTemplate  | ('extends'? templateBody)? ;  classTemplate : earlyDefs? classParents templateBody? ;  traitTemplate : earlyDefs? traitParents templateBody? ;  classParents : constr ('with' annotType)\* ;  traitParents : annotType ('with' annotType)\* ;  constr : annotType argumentExprs\* ;  earlyDefs : '{' earlyDef+ '}' 'with' ;  earlyDef : (annotation NL?)\* modifier\* patVarDef ;  constrExpr  : selfInvocation  | constrBlock ;  constrBlock : '{' selfInvocation (blockStat)\* '}' ;  selfInvocation : 'this' argumentExprs + ;  topStatSeq : topStat+ ;  topStat  : (annotation NL?)\* modifier\* tmplDef  | import\_  | packaging  | packageObject ;  Packaging : 'package' qualId NL? '{' topStatSeq '}' ;  packageObject : 'package' 'object' objectDef ;  compilationUnit : ('package' qualId)\* topStatSeq ; |