|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство образования и науки 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\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

КАФЕДРА \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ***

***НА ТЕМУ:***

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

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

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

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

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

Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

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

Руководитель ВКР **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

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

(Подпись, дата) (И.О.Фамилия)

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

(Подпись, дата) (И.О.Фамилия)

Нормоконтролер **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

*20 г.*

**Министерство образования и науки 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)**

УТВЕРЖДАЮ

Заведующий кафедрой \_\_\_\_\_\_\_\_\_\_

(Индекс)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_

(И.О.Фамилия)

« \_\_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**

**на выполнение выпускной квалификационной работы бакалавра**

Студент группы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(фамилия, имя, отчество)

Тема квалификационной работы\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Источник тематики (НИР кафедры, заказ организаций и т.п.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Тема квалификационной работы утверждена распоряжением по факультету \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ № \_\_\_\_\_\_\_\_\_\_\_от « \_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

***Часть 1. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Часть 2.\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Часть 3. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_***

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***Оформление квалификационной работы:***

Расчетно-пояснительная записка на \_\_\_\_\_ листах формата А4.

Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата выдачи задания « \_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ г.

В соответствии с учебным планом выпускную квалификационную работу выполнить в полном объеме в срок до « \_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_ г.

**Руководитель квалификационной работы** \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(Подпись, дата) (И.О.Фамилия)

**Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Подпись, дата) (И.О.Фамилия)

Примечание:

1. Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

**Министерство образования и науки 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** **\_\_\_\_\_\_\_\_\_\_\_** УТВЕРЖДАЮ

**КАФЕДРА** **\_\_\_\_\_\_\_\_\_\_\_\_\_** Заведующий кафедрой \_\_\_\_\_\_\_\_\_\_\_\_\_

(Индекс)

**ГРУППА** **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(И.О.Фамилия)

« \_\_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_\_ г.

**КАЛЕНДАРНЫЙ ПЛАН**

**выполнения выпускной квалификационной работы** студента:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(фамилия, имя, отчество)

Тема квалификационной работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **№ п/п** | **Наименование этапов выпускной квалификационной работы** | **Сроки выполнения этапов** | | **Отметка о выполнении** | |
| **план** | **факт** | **Должность** | **ФИО, подпись** |
|  | Задание на выполнение работы. Формулирование проблемы, цели и задач работы | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | 1 часть \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | *\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | Утверждение окончательных формулировок решаемой проблемы, цели работы и перечня задач | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Заведующий кафедрой |  |
|  | 2 часть \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | 3 часть \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | 1-я редакция работы | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | Подготовка доклада и презентации | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  |  |  |
|  | Заключение руководителя | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Руководитель ВКР |  |
|  | Допуск работы к защите на ГЭК (нормоконтроль) | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  | Нормоконтролер |  |
|  | Внешняя рецензия | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  |  |  |
|  | Защита работы на ГЭК | *\_\_\_\_\_\_\_\_\_\_\_\_*  *Планируемая дата* |  |  |  |

*Студент \_\_\_\_\_\_\_\_\_\_\_*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ *Руководитель работы \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

(подпись, дата) (подпись, дата)

# АННОТАЦИЯ

Работа посвящена статической верификации программ, написанных на языке Рефал. В данной работе будет рассмотрен диалект Рефала, Рефал-5.

Проведено исследование существующих неформальных нотаций для разных диалектов языка Рефал. Изучена возможность верификации программ путем статического анализа. Разработан язык описания типов и расписан алгоритм верификации.

Проведено проектирование структуры представления программ на Рефале, а также структуры для представления описания верификатора. Выполнена программная реализация построения абстрактных синтаксических деревьев на основе деревьев синтаксического анализа с дальнейшей верификацией.

Пояснительная записка к выпускной квалификационной работе содержит 60 страниц текста формата А4, 2 рисунка, 26 листингов и список используемой литературы, включающий 6 библиографических источников.

Содержание

[Введение 4](#_Toc517888635)

[1. Постановка задачи 5](#_Toc517888636)

[2. Теоретическая часть 7](#_Toc517888637)

[2.1 Описание базисного Рефала 7](#_Toc517888638)

[2.1.1 Функции и предложения 11](#_Toc517888639)

[2.1.2 Правила синтаксического отождествления 13](#_Toc517888640)

[2.2 Верификатор 19](#_Toc517888641)

[2.3 Обзор существующих неформальных нотаций 22](#_Toc517888642)

[2.3.1 Рефал-6 22](#_Toc517888643)

[2.3.2 Простой Рефал 22](#_Toc517888644)

[2.4 Описание синтаксиса 23](#_Toc517888645)

[2.4.1 Пример 24](#_Toc517888646)

[2.5 Описание алгоритма верификации 26](#_Toc517888647)

[2.5.1 Псевдотипы 26](#_Toc517888648)

[2.6 Сопоставление с образцом 27](#_Toc517888649)

[2.7 Вложение типа в тип 28](#_Toc517888650)

[2.7.1 Пример 30](#_Toc517888651)

[3. Практическая часть 32](#_Toc517888652)

[3.1 Maven 33](#_Toc517888653)

[3.2 Реализация лексического анализатора 34](#_Toc517888654)

[3.3 Реализация синтаксического анализа 35](#_Toc517888655)

[3.4 Представление программ для последующей верификации 36](#_Toc517888656)

[3.5 Верификация 40](#_Toc517888657)

[4. Тестирование 41](#_Toc517888658)

[4.1 Тест 1: Построение абстрактного дерева 41](#_Toc517888659)

[4.2 Тест 2: Неописанная альтернатива для типа 42](#_Toc517888660)

[4.3 Тест 3: Неверный тип результата функции 43](#_Toc517888661)

[4.4 Тест 4: Неверное количество аргументов 44](#_Toc517888662)

[4.5 Тест 5: Сопоставление с е-переменной 46](#_Toc517888663)

[4.6 Тест 6: Сопоставление с растяжимым типом 47](#_Toc517888664)

[4.7 Тест 7: Вложение типа в тип 47](#_Toc517888665)

[ЗАКЛЮЧЕНИЕ 48](#_Toc517888666)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 49](#_Toc517888667)

[Приложение 1 50](#_Toc517888668)

[Приложение 2 52](#_Toc517888669)

[Приложение 3 54](#_Toc517888670)

[Приложение 4 57](#_Toc517888671)

Введение

РЕФАЛ (РЕкурсивных Функций АЛгоритмический язык) - это функциональный язык программирования, ориентированный на так называемые "символьные преобразования": обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой, решение проблем, связанных с искусственным интеллектом.

В рамках данной работы был разработан язык для верификации функций и типов базисного Рефала, для нахождения ошибок связанных с несовместимостью типов.

1. Постановка задачи

Базисный Рефал является диалектом языка Рефал, разработанным в Нью-Йоркском Сити Колледже. Возможности, включенные в Базисный Рефал, прошли проверку временем.

В отличие от ЛИСПа, основу Рефала составляет сопоставление с образцом. Благодаря этому типовая программа на Рефале вдвое-втрое короче по объему, чем аналогичная программа на языке ЛИСП, и гораздо более читаема.

С формальной точки зрения все Рефал-функции являются функциями одного аргумента. Однако очень часто этот единственный аргумент состоит из частей, которые по существу являются под аргументами (но обычно называются просто аргументами). Когда определяется функция, на самом деле представляющая собой функцию нескольких аргументов, производится объединение этих аргументов каким-либо образом, чтобы образовать единственный формальный аргумент. Это задает формат функций.

К сожалению, из-за своей динамической типизации в БАЗИСНЫЙ Рефал могут возникать ошибки с типом аргументов - это передача в функцию данных, не входящих в область определения функции — в таких случаях происходит аварийный останов программы. Такие ошибки можно обнаруживать путём статического анализа — если задавать (а) область определения каждой функции каким-то формальным образом, (б) уметь сравнивать фактический аргумент с областью определения. Фактические аргументы могут быть частично заданы литерально, частично могут зависеть от результата работы других функций. Следовательно, нужно знать области значений функций, чтобы уметь вычислять множества значений аргументов других функций.

Надо заметить, что уже существует неформальная нотация, используемая в Простом Рефале для описания интерфейсов между разными проходами компилятора. Это показывает, что потребность в верификации существует и этот факт также является одним из источников появления данной задачи.

Поэтому необходимо разработать формальную нотацию для описания типов и верификатор, проверяющий эти типы.

Заметим, что идеальный верификатор разработать нельзя, так как существует проблема останова.

Говорят, что эта проблема рекурсивно не разрешима в том смысле, что не существует алгоритма, который для каждой машины Тьюринга и каждой конфигурации определял бы, остановится ли машина когда-нибудь.

Если считать, что ошибки в программе приводят к зацикливанию алгоритма, то проблема доказательства отсутствия ошибок сводится к доказательству того, что алгоритм не зацикливается.

Из этого следует, что невозможно написать верификатор, который всегда сможет сказать будет работать алгоритм или нет. Наш верификатор может сказать либо алгоритм будет работать, либо есть места, где он может потенциально не сработать.

1. Теоретическая часть

Для реализации задуманного необходимо придумать язык верификатора, а также описать алгоритм, производящий непосредственно верификацию программы.

## 2.1 Описание базисного Рефала

Основной конструкцией языка Рефал, служащей для записи обрабатываемых символьных данных и правил их обработки, является так называемое *рефал-выражение*. Рефал-выражения, как и рефал-программы в целом, записываются при помощи знаков, которые принято делить на *собственные* и *объектные*.

Собственныезнаки – это специальные символы языка, выполняющие в нём функцию, аналогичную роли ключевых слов в других языках программирования. К числу собственных знаков относятся круглые и угловые скобки **(** и **)**, **<** и **>**, знак равенства **=**, одинарная кавычка (апостроф) ', а также латинские буквы **S** и **s**, **T** и **t**, **E** и **e**.

Объектные знаки, или символы-литеры, служат для изображения символов (литер), встречающихся в обрабатываемых данных. В набор объектных знаков входят цифры, буквы, знаки арифметических операций и другие знаки кода ASCII. В отличие от собственных знаков, любой объектный знак при записи рефал-выражения должен быть заключён в одинарные кавычки (апострофы), называемые символьными скобками, например: 'B', '+', 'e', ')'. Таким образом, собственный знак ) при записи отличается от объектного ')', а собственный знак **Е** от объектного 'Е' и т.п. Запись ' ' обозначает литеру пробела, которая является такой же полноценной литерой, как и все остальные.

Если необходимо записать несколько подряд идущих объектных знаков, то эти знаки можно написать друг за другом и все их заключить в апострофы, например: 'x+y', 'define', '  ' (последняя цепочка состоит из двух символов пробела). Заметим, что такая запись цепочек символов есть сокращение последовательной записи символов-литер, при которой между символами-литерами должен стоять один или несколько пробелов: 'x' '+' 'y', 'd' 'e' 'f' 'i' 'n' 'e', ' ' ' '.

Выражения языка Рефал строятся из *атомарных* (т.е. неделимых) *элементов*, каковыми являются *символы-литеры, символы-числа* (записываются как целые десятичные числа)*, составные символы* (записываются как имена, начинаются с латинской буквы, состоят из букв, цифр, дефисов и прочерков)и *переменные*, а также скобок трёх видов:

* структурные **(** **)**;
* функциональные < >;
* символьные ' '.

Структурные скобки служат для структурирования обрабатываемых данных, а функциональные скобки – для записи обращений к функциям. Упрощённо говоря, рефал-выражение представляет собой некоторую последовательность атомарных элементов, сбалансированную по всем трём видам скобок, например:

**(**'for' **(**'i=1'**))** sa **(**'g' <f5 e1> **(**e2**))**

В этом примере 'for', 'i=1', 'g' – цепочки символов-литер, sa, е1, e2 – переменные, f5 – имя функции, а <f5 e1> обращение к этой функции.

Переменная языка Рефал включает:

*признак\_типа .имя*

Признак типа записывается буквой s, t или е и определяет тип значений, которые может принимать эта переменная:

* s (или S) – значением переменной может быть только символ-литера;
* t (или T) – значением переменной может быть так называемый *терм*– символ-литера или выражение в структурных скобках;
* e (или E) – значением переменной может быть произвольное рефал-выражение (в том числе и пустое).

*Имя* переменной может содержать любые буквы латинского алфавита, цифры, дефисы и прочерки. Фактически индекс является именем переменной и служит для её идентификации, поэтому, к примеру, e.1 и e.Abc – разные переменные.

Если переменные и символы-литеры служат атомарными элементами рефал-выражений, то более крупной структурной единицей выражения является *терм*.

Атомарный элемент представляет простейший случай терма, а в общем случае терм– это рефал-выражение, взятое в структурные или функциональные скобки и называемое соответственно *структурным* или *функциональным* *термом*. Заметим, что функциональный терм есть по сути вызов функции.

Рефал-выражение (или просто – выражение) представляет собой последовательность термов, его синтаксис можно описать следующими правилами в форме Бэкуса-Наура (далее – БНФ-правила):

*выражение* ::= *пусто* | *терм выражение*

*терм* ::= *атомарный\_элемент* | *структурный\_терм* |

*функциональный\_терм*

*атомарный\_элемент* ::= *переменная* | *символ-литера*

*структурный\_терм* ::= (*выражение*)

*функциональный\_терм* ::= <*имя\_функции выражение>*

*пусто* ::=

Отметим, что простейшим видом выражения является *пустое выражение*. Непустое выражение состоит из одного или нескольких термов, называемых *термами верхнего уровня*. Например, выражение 'А'('BЕ'('C')'D') состоит на верхнем уровне из двух термов: символа-литеры 'А' и структурного терма ('BЕ'('C')'D'), выражение внутри которого в свою очередь состоит из четырёх вложенных термов (трёх символов-литер 'B', 'E', 'D' и одного структурного терма ('C')).

В зависимости от того, какие атомарные элементы и какие скобки используются при образовании выражения, различают следующие виды выражений:

* *объектное выражение* – выражение, не содержащее переменных и функциональных скобок, например: 'type'**(**'int3 = 1..3'**)**;
* *выражение-образец* – выражение, не содержащее функциональных скобок, например: e1 '+' e2 (sa e3);
* *рабочее выражение* – выражение без переменных, например:   
  'a-b' <pr '+c+g'>(<fun 'dg'>);
* *выражение общего вида*, в котором могут встретиться все виды скобок и атомарных элементов (далее – *рефал-выражение* или просто *выражение*).

Важно, что рефал-переменные могут иметь значением только объектные выражения. Объектным выражением должен быть и аргумент вычисляемого обращения к функции.

### 2.1.1 Функции и предложения

Каждая функция языка Рефал имеет только один аргумент, однако такое ограничение числа аргументов рефал-функций не снижает выразительной и вычислительной мощности языка. Действительно, аргументом функции является объектное выражение – последовательность из нескольких термов, которые могут рассматриваться как последовательные аргументы функции.

Обращение к рефал-функции имеет вид функционального терма, причём имя функции стоит внутри функциональных скобок:

<*имя\_функции  выражение>*

Определение рефал-функции состоит из нескольких *предложений* вида

*выражение-образец*= *выражение ;*

окруженных фигурными скобками и записываемых последовательно друг за другом по строкам. Каждое предложение включает *левую часть* (выражение-образец) и *правую часть* (рефал-выражение общего вида) и описывает правило преобразования и замены: выражение-образец характеризует заменяемое выражение, выражение в правой части определяет результат его замены. Подчеркнём, что в левой части предложения не могут встречаться обращения к функциям.

При определении функции, в начале её первого предложения, перед его левой частью записывается имя функции, которое отделяется от последующего выражения-образца одним или несколькими пробелами. Остальные предложения функции не нуждаются в именовании, но их запись на строке должна начинаться хотя бы с одного пробела.

Приведём пример функции, описываемой двумя предложениями. В нашем примере функции первое предложение описывает нужное преобразование цепочки символов (замену в ней знака +), а второе соответствует случаю, когда знак + в цепочке символов отсутствует, и она остаётся прежней.

\* Функция ChangeFirstPlus заменяет в цепочке символов

\* знак сложения (если он есть) на знак вычитания,

\* например, 'A+B' будет преобразовано в 'A-B',

\* а 'XY+Z85' – в 'XY-Z85'

ChangeFirstPlus {

e.1 '+' e.2 = e.1 '-' e.2;

e.1 = e.1;

}

Порядок записи предложений в рефал-функции существенен, поскольку каждое предложение может быть применено только тогда, когда не применимы все предыдущие предложения.

Выражения-образцы в левой части предложений обычно содержат рефал-переменные. Эти переменные локальны, областью их действия является только то предложение функции, в котором они употребляются. Для нашего примера функции ChangeFirstPlus это означает, что значение переменной e1 при использовании первого предложения может быть отлично от значения этой же переменной при использовании второго предложения.

Переменные получают свои значения при проверке применимости предложения, и эти значения используются для выполнения описанной в этом предложении замены. Тем самым, в правой части предложения могут быть использованы только те переменные, которые были использованы в левой части.

В общем случае может быть несколько вхождений одной и той же переменной как в левую, так и в правую часть предложения, например:

s.1 e.2 s.1 e.3 = s.1 s.1 e.2 e.3

Все вхождения одной и той же переменной (т.е. переменной с конкретным индексом) в левую и правую часть предложения должны иметь одинаковый признак типа. Тем самым недопустимы предложения вида s.1 e.1 = e.1 или s.1 e.2 = e.1.

Приведём пример ещё одной рефал-функции:

\* Функция ChangeAllPlus заменяет в цепочке символов

\* все знаки сложения на знаки вычитания, например,

\* 'A+B+C' будет преобразовано в 'A-B-C'

ChangeAllPlus {

e.1 '+' e.2 = <ChangeAllPlus e.1 '-' e.2>;

e.1 = e.1;

}

В отличие от функции ChangeFirstPlus, функция ChangeAllPlus *рекурсивна*: в правой части первого предложения стоит обращение к ней самой, рекурсия реализует последовательную, циклическую замену знаков сложения. Этот процесс заканчивается, когда в обрабатываемой цепочке символов не останется знаков +, и применяется второе предложение функции.

Использование рекурсии является одной из основных особенностей программирования на языке Рефал.

### 2.1.2 Правила синтаксического отождествления

Синтаксическое отождествление выражения-образца и объектного выражения является наиболее сложной операцией, выполняемой рефал-машиной. Отождествление означает, что объектное выражение является частным случаем выражения-образца, как в рассмотренных выше и приводимых ниже примерах.

Выражение-образец 'for ' e.1 '=' e.2 'to' e.3 'do' e.4 отождествимо со строкой литер 'for i=1 to 12 do n:=n+1' при значениях переменных: e.1 ↔ 'i', e.2 ↔ '1 ' (значение е2 – два символа, знак единицы и знак пробела), e.3 ↔ ' 12 '(значение е.3 – четыре символа: пробел, 1, 2, пробел), e.4 ↔ ' n:=n+1'.

Выражение-образец s.1 e.2 ')' e.3 отождествимо со строкой 'A\*(B+C)-D' при значениях переменных: s.1 ↔ 'A', e.2 ↔ '\*(B+C', e.3 ↔ '-D' (скобка в выражении-образце не является структурной, в нём указана литера круглой скобки).

Выражение-образец e.1 '+' t.2 e.3 синтаксически отождествимо со строкой литер 'X‑5\*Y+8/Z' при таких значениях переменных: e.1 ↔ 'X‑5\*Y', t.2 ↔ '8', e.3 ↔ '/Z'.

Сформулируем общие правила выполнения синтаксического отождествления. Синтаксическое отождествление выражения-образца и объектного выражения выполняется последовательно по входящим в них атомарным элементам и всем парам структурных скобок. При этом должны выполняться следующие условия:

1. Символ-литера синтаксически отождествим только с таким же символом-литерой, например, 'g' отождествим с 'g'.
2. Переменная отождествима только с выражением, вид и структура которого соответствует типу переменной. В частности, e-переменная отождествима с любым рефал выражением, в том числе пустым; t-переменная отождествима с термом; s-переменная отождествима с любым символом-литерой.
3. Все вхождения в выражение-образец одной и той же переменной должны получить одно и то же значение. Поэтому, к примеру, выражение-образец s.1 e.2 s.1 отождествимо со строкой 'abca' (s.1 ↔ 'a', e.2 ↔ 'bc'), но не со строкой 'abcd'.
4. При отождествлении левых структурных скобок выражений должны отождествляться и соответствующие (парные к ним) правые структурные скобки этих выражений. Это означает, что при отождествлении баланс структурных скобок не может быть нарушен.

Реализованный в рефал-интерпретаторе алгоритм, выполняющий синтаксическое отождествление выражений и в случае успеха определяющий значения употреблённых в выражении-образце переменных, называется *алгоритмом проектирования* выражения-образца *на* объектное выражение. Для написания эффективных рефал-программ необходимо понимать важные особенности этого алгоритма.

В общем случае выражение-образец содержит так называемые *жёсткие элементы* – элементы, которые однозначно проектируются на соответствующие элементы объектного выражения. Символы-литеры, структурные скобки, s- и t-переменные – это жёсткие элементы выражения, и если у такого элемента спроектирован один конец, то однозначно проектируется и второй, либо делается вывод о невозможности дальнейшего отождествления. Для символов-литер, структурных скобок и s-переменных это очевидно. Для t‑переменных возможная проекция – это либо символ (тоже очевидный случай), либо выражение в структурных скобках (но в этом случае однозначно находится парная скобка). На всех шагах проектирования действует общий принцип: если спроектирован один конец жёсткого элемента, то делается попытка спроектировать и его второй конец, т.е. весь элемент в целом.

Например, выражение-образец 'A' (e.1 t.2) s.3 будет отождествляться с объектным выражением 'A'(('2В'))'7' следующим образом (см. Рис. 1):

1. символ-литера 'A' отождествится с символом-литерой 'A';
2. первая открывающаяся структурная скобка из объектного выражения отождествится с открывающейся скобкой из выражения-образца;
3. парная ей последняя закрывающаяся скобка из объектного выражения отождествится с закрывающейся структурной скобкой из выражения-образца;
4. переменная t.2 выражения-образца отождествится со структурным термом ('2В');
5. переменная e.1 отождествится с пустым выражением;
6. переменная s.3 отождествится с символом-литерой '7'.

'A'

(

е.1

t.2

)

s.3

'A'

(

(

'2'

'В'

)

)

'7'

Рисунок 1. Пример отождествления

1

2

5

4

3

6

Рассмотрим дополнительные примеры.

Выражение-образец s.1(e.2 ',' e.3) синтаксически отождествим с объектным выражением 'c'('ga,d'), при значениях переменных s.1 ↔ 'c', e.2 ↔ 'ga', e.3 ↔ 'd'. Образец s.x t.y отождествим с тем же объектным выражением при s.x ↔ 'c', t.y ↔ ('ga,d').

Выражения ')' и '(' являются обычными символами-литерами и могут быть значениями s-переменных, в отличие от структурных скобок. Баланс символов-литер скобок при отождествлении не проверяется. Поэтому единственным вариантом отождествления образца (e.1(s.2'(')) с объектным выражением ('a)'('b(')) являются значения переменных e.1 ↔ 'a)' и s.2 ↔ 'b'.

Можно заметить, что в ряде случаев при использовании в выражении-образце нескольких e‑переменных синтаксическое отождествление возможно для нескольких вариантов приписывания этим переменным значений. К примеру, при отождествлении образца e1'+'e2 и выражения 'a+b+c' возможны два варианта: e.1 ↔ 'a', e.2 ↔ 'b+c' и e.1 ↔ 'a+b', e.2 ↔ 'c'. Возникновение таких неоднозначностей связано с тем, что е-переменные не ограничиваются по длине, их значениями могут быть произвольные последовательности термов.

Ясно, что алгоритм проектирования должен устранять неоднозначности такого типа в случае их возникновения. Возможны два способа, которые называются соответственно отождествление слева направо и отождествление справа налево, или левое и правое согласование*.*

При левом согласовании рефал-машина выбирает тот вариант отождествления (вариант приписывания переменным значений), при котором первая слева е-переменная выражения-образца принимает самое короткое возможное для неё значение. Если это не устраняет неоднозначности, то такой же отбор производится по второй слева е-переменной, затем третьей слева и т.д.

При правом согласовании рефал-машина выбирает тот вариант отождествления, при котором первая справа е-переменная принимает самое короткое возможное значение. Если же это не снимает неоднозначности, то такой же отбор производится по второй справа е-переменной, затем третьей справа и т.д.

В нашем примере первый вариант отождествления соответствует левому согласованию, а второй – правому. По умолчанию рефал-машина реализует левое согласование.

Заметим, что при отождествлении образца е.1 е.2 с тем же выражением 'a+b+c' при левом согласовании переменные получат значения: e.1 ↔ *пусто*, e.2 ↔ 'a+b+c'; а при правом: e.1 ↔ 'a+b+c', e.2 ↔ *пусто*.

Рассмотрим вновь функцию ChangeAllPlus, определённую в разделе 1.3, и приведём пример её работы.

Пусть в поле зрения рефал-машины помещено выражение <ChangeAllPlus 'A+B+C'>. На первом шаге работы рефал-машины к этому функциональному терму применимо первое предложение функции ChangeAllPlus, поскольку выражение 'A+B+C' отождествимо с образцом e.1 '+' e.2 при e.1 ↔ 'A', e.2 ↔ 'B+C' (по умолчанию используется левое согласование). В результате в поле зрения рефал-машины появится выражение  
<ChangeAllPlus 'А-B+C'>   
Поскольку это выражение содержит только один функциональный терм, он будет ведущим, и к входящему в него выражению 'А-B+C' на втором шаге работы рефал-машины будет вновь применено первое предложение функции ChangeAllPlus, но уже при значениях переменных e.1 ↔ 'А‑B', e.2 ↔ 'C'. Поле зрения после выполнения этого шага примет вид: <ChangeAllPlus 'А-В-C'>

На третьем шаге работы рефал-машины единственный (и ведущий) функциональный терм заменится на выражение 'А-В-C', так как применимым будет только второе предложение функции при е.1 ↔ 'А‑В‑C' (в аргументе функционального обращения нет символа +).

На четвёртом шаге произойдёт останов рефал-машины, поскольку в поле зрения нет функциональных термов, а оставшееся в нём выражение 'A-В-С' является результатом вычисления функционального вызова <ChangeAllPlus 'A+B+C'>

2.2 Верификатор

Как любой другой продукт производства, программа для вычислительной машины перед использованием должна быть проверена. Одним из путей проверки или тестирования программ является выполнение этой программы по одному разу с каждой из возможных комбинаций входных данных. Легко видеть, что в большинстве случаев это совершенно непрактично. Обычная вычислительная машина может представлять около миллиарда различных целых чисел, поэтому, если мы хотим исчерпывающим образом проверить работу программы, которая читает всего одно целое число, нам придется выполнить ее миллиард раз и проверить миллиард наборов выходных результатов. Ясно, что нам надо как-нибудь уменьшить число необходимых проверок.

Одним из путей такого уменьшения является выполнение случайных тестов. Хотя случайное тестирование может обнаружить в программе крупные ошибки, например, программа постоянно выдает неправильный результат, очень маловероятно. Чтобы случайное тестирование обнаружило какие-либо нерегулярности в поведении программы. Программа, использующая, например, значения функции tg(x), может выдавать ненадежные результаты при х ≈ π/2, но случайное тестирование этого, наверное, не обнаружит.

Предположим, у нас есть программа, содержащая оператор, который присваивает z значение max (х, у), т. е. наибольшего из чисел х и у.

Программист, разрабатывающий тест для этой программы, заметит, что существуют два случая, х > у и х ≤ у, и позаботится о том, чтобы тест включал в себя данные, проверяющие оба этих случая. Вообще говоря, согласно этому методу, во время тестирования нужно выполнить операторы на всех ветвях программы.

Существует множество больших и сложных программ, которые, хотя и прошли весьма тщательное тестирование, иногда сбиваются. Например, компиляторы работают сотни раз в день, но вдруг могут отказаться, компилировать правильную программу. Многие операционные системы непрерывно работают целыми днями, но могут вдруг неожиданно сбиваться и останавливаться. В этих случаях некоторые редкие комбинации событий привели процесс к состоянию, не предусмотренному программистом.

Подобные сложные программы никогда не могут быть оттестированы полностью. Например, компилятор должен обрабатывать любую строку символов конечной длины и, если строка соответствует определению языка, должен сформировать эквивалентную рабочую программу. Операционные системы тестировать еще труднее, поскольку они управляют множеством параллельных процессов. Когда сбивается операционная система, часто бывает невозможно даже воспроизвести те условия, которые привели к сбою.

Некоторые специалисты предлагают, вместо того чтобы тестировать программы, доказывать, что они правильно выполняют свои функции. Так как доказанный оператор надежен не более, чем предположения, лежащие в основании доказательства, перед тем, как доказывать правильность программы, написанной на данном языке, нам надо определить смысл каждой языковой конструкции. Это осуществимо только в том случае, если в нашем распоряжении имеется идеальный процессор, поэтому обычно мы не можем гарантировать правильную работу доказанной программы на реальной вычислительной машине. Однако, можно утверждать, что программа правильна, даже если вычислительная машина не сможет правильно ее выполнить.

Подробное изложение теории доказательства правильности программ или их верификации, как обычно называют этот процесс, не входит в задачу этой работы. В самом деле, концепции, используемые при верификации, представляют собой не более чем формализованные

у : = 1/х

где x и y — вещественные переменные некоторой программы.

Написав этот оператор, мы должны иметь доводы в пользу того, что х в данный момент не равен нулю, поскольку иначе нельзя будет вычислить выражение и, кроме того, мы ожидаем, что после выполнения оператора у будет иметь значение 1/х. В развитие этих соображений мы и опишем методику верификации программы.

С каждым оператором программы мы свяжем два булевских выражения. Первое из них будет называться пред-условием оператора, оно выражает наше знание о состоянии программы перед началом выполнения оператора. Второе выражение назовем пост-условием оператора, оно будет выражать наше знание о состоянии программы после выполнения оператора.

Предположим, в некоторой точке программы, где есть уверенность что х > 0, мы пишем оператор

у : = 1/х

Далее мы будем знать, что и после выполнения этого оператора условие х > 0 останется справедливым, так как присваивание не изменит значение х, мы будем также знать, что у = 1/х. Поэтому:

у : = 1/х

{(X > 0) & (Y = 1/Х)}

Буквами X и Y мы обозначили текущие значения переменных х и у.

2.3 Обзор существующих неформальных нотаций

Как уже говорилось ранее уже существует несколько неформальных нотаций, и чтобы разработать нотацию для верификатора сначала рассмотрим существующие примеры.

2.3.1 Рефал-6

Рефал-6 является диалектом и расширением языка Refal-5. Отличительной чертой являются откатные функции и блоки, разделяемые термы, большинство данных интерпретируются как абстрактные контейнеры, динамическая загрузка, полный доступ к программному коду. В исходных текстах Рефала-6 (написанных на Рефале-6) встречается своя нотация описания типов.

|  |
| --- |
| \* <Expl sD s.nel t.Vars t.Map (eL) eT (eR) t.Holes>  \*        => e.Ops s.nel1 t.Vars1 t.Map1  \*  sD = L | R : which side to inspect;  \*  e.Ops - Accumulator for result program  \*  s.Nel - the first free nel;  \*  t.Vars = ( (s.Type s.Ind)s.Nel ... ) ; |

*Листинг 1*

Как видим для описания функций используется конструкция вида <Имя функции входные данные> => выходные данные, а для описания типов используется конструкция вида Тип = вариант1 | вариант2 … Сами варианты, как и типы задаются как в Рефале.

2.3.2 Простой Рефал

Особенности языка, отличающие его от других диалектов Рефала:

Функции — подмножество Базисного Рефала, т.е. расширенных конструкций типа условий, блоков, действий и т.д. не имеют; вложенные безымянные функции; поддержка инкапсуляции на уровне данных — именованные скобки, т. н. абстрактные типы данных; идентификаторы (аналог compound-символов) не могут создаваться во время выполнения.

Как говорилось ранее, в Простом Рефале неформальная нотация используется для описания интерфейсов между отдельными проходами компиляции.

|  |
| --- |
| <ParseProgram t.ErrorList e.Tokens>  == t.ErrorList e.AST  e.AST ::= t.ProgramElement\*  t.ProgramElement ::=  (#Function t.SrcPos s.ScopeClass (e.Name) e.Body)  | (#Enum t.SrcPos s.ScopeClass e.Name)  | (#Swap t.SrcPos s.ScopeClass e.Name)  | (#Declaration t.SrcPos s.ScopeClass e.Name)  | (#Ident t.SrcPos e.Name)  | (#NativeBlock t.SrcPos e.Code)  s.ScopeClass ::= #GN-Entry | #GN-Local  t.SrcPos ::= s.LineNumber |

*Листинг 2*

Как видим, здесь нотация похожа на ту, что мы рассмотрели в первом примере. Разница заключается лишь в обозначениях, а также в наличии тегов типов в описании.

2.4 Описание синтаксиса

Опираясь на рассмотренные выше примеры напишем описание языка в форме Бэкуса-Наура.

Терминальные символы: круглые скобки, треугольные скобки, знак двойного равенства, вертикальная черта, точка с запятой, квадроточие и равенство, звездочка, знак плюс, Имя – последовательность латинских букв, переменная – тип переменной как в Рефале, точка, последовательность латинских букв начиная с заглавной буквы, метапеременная – тип переменной как в Рефале, точка, последовательность латинских букв начиная с заглавной буквы.

Все остальные символы не терминальные и будут писаться маленькими буквами.

спецификация ::= описание типа | описание функции

описание типа ::= ПЕРЕМЕННАЯ “::=” простые типы “;”

простые типы ::= простой тип | простой тип ВЕРТИКАЛЬНАЯ ЧЕРТА простые типы

простой тип ::= фиксированный тип | фиксированный тип терм“\*” фиксированный тип

фиксированный тип ::= терм | терм фиксированный тип

терм ::= ПЕРЕМЕННАЯ | МЕТАПЕРЕМЕННАЯ | (простой тип) | (простой тип)\*

описание функции ::= <ИМЯ аргумент> == простой тип ;

аргумент ::= простой тип

ВЕРТИКАЛЬНАЯ ЧЕРТА означает альтернативу. Термы схожи с термами из Рефал. Если имеется переменная, то для нее должно существовать описание в верификаторе, а для метапеременной нет. Растяжимый тип означает, что данный элемент может повторяться: произвольное количество раз (символ \*).

По правилам нашего верификатора все типы должны быть описаны, кроме базовых. Базовых типов шесть: s.NUMBER, s.COMPOUND, s.CHAR, s.ANY, t.ANY, e.ANY. Они описывают базовые примитивы, которые есть в Рефале.

2.4.1 Пример

|  |
| --- |
| t.Token ::= (s.TokenType t.Coords e.info);  s.TokenType ::= Number | Ident | LBracket | RBracket | Plus | MINUS;  t.Coords ::= (t.PosBegin t.PosEnd);  t.PosBegin ::= t.Pos;  t.PosEnd ::= t.Pos;  t.Pos ::= (s.Line s.Col);  s.Col ::= s.NUMBER;  s.Line ::= s.NUMBER;  <PrintTokens t.Token\*> == ;  <StrOfCoords t.Coords> == s.CHAR\*;  <StrOfPos t.Pos> == s.CHAR\*; |

*Листинг 3*

Данная программа работает с токенами, каждый из которых состоит из трех полей: тип токена, координаты токена и информация о токене. Тип токена может быть числом, переменной, правой или левой скобкой, знаком плюс или минус. Координаты состоят из двух полей, каждое из которых имеет тип позиция. Позиция – это упорядоченная пара чисел. Информацией может быть что угодно.

PrintTokens принимает на вход токены и ничего не возвращает.

StrOfCoords переводит координату в строку.

StrOfPos t.Pos переводит позицию в строку.

2.5 Описание алгоритма верификации

1. Для каждой функции в программе
   1. Для каждого предложения функции
      1. Сопоставляем тип аргумента с образцом
      2. Переменные связываются с некоторым образцом
      3. Верифицируем вызовы в правой части
      4. Верифицируем тип результатной части
      5. Сопоставляем тип результатной части с типом функции

Верифицирование вызова в правой части:

Вычисляем тип фактического аргумента параметра функции из типов переменных. Сопоставляем тип фактического параметра с типом формального параметра. В результате получают значение метапеременных типов и подставляют в формальный результат функции типы метапеременных и получаем фактический результат.

### 2.5.1 Псевдотипы

Псевдотипы описывают «тип» результатного выражения. Нам для удобства алгоритмов сопоставления типа с образцом и вложения типа в тип удобно оперировать фиксированными и растяжимыми типами. Т.е. либо с фиксированным числом термов известного типа, либо с «гармошкой» с единственным итератором посередине. Но результатное выражение не может быть описано как тип в таком определении. Например, при сопоставлении с образцом переменные e.X и e.Y получили типы-«гармошки», условно A B {C} D E ← e.X, P {Q} R S ← e.Y. Тогда выражение вида e.X '+' e.Y будет описываться как A B {C} D E '+' P {Q} R S. «Двойная гармошка». Можно придумать алгоритм сведения гармошки к некоему «объемлющему» типу, т.е. потерять информацию. Но можно определить вложение псевдотипа (в таком «ненормализованном» виде) в нормальный тип.

На самом деле типы близки к т.н. жёстким выражениям (или L-выражениям) Рефала — выражениям, в которых на каждом скобочном уровне не может располагаться более одной e-переменной и e-переменные не могут повторяться. А псевдотипы — выражениям общего вида, на которые никакие ограничения не накладываются. Можно решить уравнение вида E : L, т.е. имея выражение общего вида и L-выражение, подобрать значения переменных для обоих половинок такое, что уравнение обращается в верное равенство. Алгоритм описан Турчиным, он достаточно простой. А вот решить уравнение E1 : E2, т.е. приравнять два выражения общего вида — это уже целый раздел математики, «уравнения в словах» так называемые.

С типами и псевдотипами нечто похожее. Для проверки вложения типа в псевдотип достаточно простого алгоритма с индукцией по конструкторам. Для проверки равенства или вложения двух псевдотипов нужно привлекать мат.аппарат древесных автоматов (это как конечные автоматы, только работают не со строкой, а с деревом).

2.6 Сопоставление с образцом

Сопоставление с образцом — метод анализа и обработки структур данных, основанный на выполнении определённых инструкций в зависимости от совпадения исследуемого значения с тем или иным образцом, в качестве которого может использоваться константа, предикат, тип данных или иная поддерживаемая языком конструкция.

Сопоставление с образцом происходит следующим образом:

ТЕ – типовое выражение

ТТ – терм типа

РЕ – образцовое выражение

РТ – образец уровня терм

Если у нас сравнение ТТ ТЕ ~ t.Var РЕ, то t.Var -> ТТ; TE ~ PE. Заметим, что если ранее данная переменная уже связывалась с неким типом, то новый тип должен совпадать со старым, иначе ошибка.

Если у нас сравнение ТЕ ТТ ~ РЕ t.Var, то t.Var -> ТТ; TE ~ PE.

Если у нас сравнение (ТЕ1) ТЕ2 ~ (РЕ1) РЕ2, то TE1 ~ PE1; TE2 ~ PE2.

Атом1 ТЕ ~ Атом2 РЕ => ТЕ ~ РЕ и Атом1 == Атом2, иначе ошибка.

Если ТТ\* ~ t.Var PE, то t.Var -> TT; TT\* ~ PE.

Если ТE ~ e.Var, то e.Var -> TE.

Если ТТ\* ~ e.Var1 PE e.Var2, то e.Var1, e.Var2 -> TT\*; TT\* ~ PE.

## Вложение типа в тип

В данном разделе мы рассмотрим алгоритм вложения типа в тип. Общей концепцией для правел приведенных ниже, является следующая концепция: Если мы раскрываем псевдотип, то все его альтернативы должны вкладываться в тип. А если мы раскрываем тип, то нам нужно, чтобы псевдотип вкладывался хотя бы в одну из альтернатив типа.

ВЛОЖЕНИЕ(ПСЕВДОТИП, ТИП) возвращает булевское значение

ВЛОЖЕНИЕ(Tt T, Tt’ T’) = ВЛОЖЕНИЕ(Tt, Tt’) && ВЛОЖЕНИЕ(T, T’)

ВЛОЖЕНИЕ(Tt \*T, T’) = ВЛОЖЕНИЕ(T, T’) || ВЛОЖЕНИЕ(Tt Tt\*T, T’)

ВЛОЖЕНИЕ(T Tt\*, T’) = ВЛОЖЕНИЕ(T, T’) ||ВЛОЖЕНИЕ(T Tt\* Tt, T’)

ВЛОЖЕНИЕ(e.Type T, T’) =

Пусть e.Type ::= T1 | T2 | T3 | … | TN тогда

ВЛОЖЕНИЕ(T1 T, T’) &&

ВЛОЖЕНИЕ(T2 T, T’) &&

…

ВЛОЖЕНИЕ(TN T, T’)

ВЛОЖЕНИЕ(T, e.Type) =

Пусть e.Type ::= T1 | T2 | T3 | … | TN тогда

ВЛОЖЕНИЕ(T, T1) ||

ВЛОЖЕНИЕ(T, T2) ||

…

ВЛОЖЕНИЕ(T, TN)

ВЛОЖЕНИЕ(T1 T2,Tt\*)=ВЛОЖЕНИЕ(T1, Tt\*) && ВЛОЖЕНИЕ(T2, Tt\*)

ВЛОЖЕНИЕ(Tt, Tt’\*) = ВЛОЖЕНИЕ(Tt, Tt’)

ВЛОЖЕНИЕ(Tt\*, Tt’\*) = ВЛОЖЕНИЕ(Tt, Tt’)

ВЛОЖЕНИЕ((T), (T’)) = ВЛОЖЕНИЕ(T, T’)

ВЛОЖЕНИЕ(var, Tt) =

Пусть var ::= Tt1 | Tt2 | Tt3 | … | TtN тогда

ВЛОЖЕНИЕ(Tt1, Tt) &&

ВЛОЖЕНИЕ(Tt2, Tt) &&

…

ВЛОЖЕНИЕ(TtN, Tt)

ВЛОЖЕНИЕ(Tt, var) =

Пусть var ::= Tt1 | Tt2 | Tt3 | … | TtN тогда

ВЛОЖЕНИЕ(Tt, Tt1) ||

ВЛОЖЕНИЕ(Tt, Tt2) ||

…

ВЛОЖЕНИЕ(Tt, TtN)

### Пример

Рассмотрим пример из двоичного дерева и 2-3-дерева

t.Tree1 ::= Leaf | (t.Tree1 s.ANY t.Tree1);

t.Tree2::=Leaf | (t.Tree2 t.ANY t.Tree2) |(t.Tree2 t.ANY t.Tree2 t.ANY t.Tree2);

Мы хотим проверить, вкладывается ли двоичное дерево в 2-3-дерево

t.Tree1 ≤ t.Tree2

[Leaf ≤ t.Tree2] && [(t.Tree1 t.ANY t.Tree1) ≤ t.Tree2]

Как говорилось ранее, раскрывая левую часть мы проверяем, чтобы каждая из альтернатив удовлетворяла правой части. Далее мы раскроем правую часть. В этом случае для нас важно, чтобы хотя-бы одна из альтернатив удовлетворяла условию вложения.

[Leaf ≤ Leaf] || [Leaf ≤ (t.Tree2 t.ANY t.Tree2)] || [Leaf ≤ (t.Tree2 t.ANY t.Tree2 t.ANY t.Tree2)]] && [[(t.Tree1 s.ANY t.Tree1) ≤ Leaf] || [(t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2)] || [(t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2 t.ANY t.Tree2)]]

Leaf ≤ Leaf – истинно, а значит весь блок ИЛИ истинен. (t.Tree1 s.ANY t.Tree1) ≤ Leaf – ложно и поэтому надо проверить на истинность оставшиеся выражения.

TRUE && [FALSE || [(t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2)] || [(t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2 t.ANY t.Tree2)]]]

(t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2) и (t.Tree1 s.ANY t.Tree1) ≤ (t.Tree2 t.ANY t.Tree2 t.ANY t.Tree2) тоже самое, что и t.Tree1 ≤ t.Tree2. Получаем рекурсию. В этом случае возвращаем TRUE.

3. Практическая часть

Как говорилось ранее для реализации верификатора был выбран язык программирования java.

Программы на Java транслируются в байт-код Java, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java:

применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,

широкое использование платформенно-ориентированного кода (native-код) в стандартных библиотеках,

аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами фирмы ARM).

Идеи, заложенные в концепцию и различные реализации среды виртуальной машины Java, вдохновили множество энтузиастов на расширение перечня языков, которые могли бы быть использованы для создания программ, исполняемых на виртуальной машине.

## 3.1 Maven

Apache Maven — фреймворк для автоматизации сборки проектов на основе описания их структуры в файлах на языке POM, являющемся подмножеством XML. Проект Maven издаётся сообществом Apache Software Foundation, где формально является частью Jakarta Project.

Название программы, maven, — является словом из языка идиш, смысл которого можно примерно выразить как «собиратель знания».

Maven обеспечивает декларативную, а не императивную (в отличие от средства автоматизации сборки Apache Ant) сборку проекта. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, Maven выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Вся структура проекта описывается в файле pom.xml (POM – Project Object Model), который должен находиться в корневой папке проекта. Ключевым понятием Maven является артефакт — это, по сути, любая библиотека, хранящаяся в репозитории. Это может быть какая-то зависимость или плагин.

Зависимости — это те библиотеки, которые непосредственно используются в вашем проекте для компиляции кода или его тестирования.

Плагины же используются самим Maven'ом при сборке проекта или для каких-то других целей (деплоймент, создание файлов проекта для Eclipse и др.).

3.2 Реализация лексического анализатора

Для написания лексического анализатора использовался генератор лексических анализаторов JFlex.

JFlex - это генератор лексического анализатора для Java, написанный на Java. Это также переработка инструмента JLex (Berk 1996), который был разработан Эллиотом Берком в Принстонском университете. Как утверждает Верн Пакссон в своем инструменте C / C ++ flex (Paxson 1995): они не разделяют никакого кода.

Генератор лексического анализатора принимает в качестве входных данных спецификацию с набором регулярных выражений и соответствующих действий. Он генерирует программу (лексер), которая считывает ввод, сопоставляет ввод с регулярными выражениями в специальном файле и запускает соответствующее действие, если регулярное выражение соответствует. Лексеры обычно являются первым интерфейсом в компиляторах, сопоставляя ключевые слова, комментарии, операторы и т. Д. И генерируя поток входных токенов для парсеров. Они также могут использоваться для многих других целей.

Цели проектирования

Основными целями дизайна JFlex являются:

Поддержка Unicode

Быстрое создание сканеров

Быстрое создание сканера

Удобный синтаксис спецификации

Независимость платформы

Совместимость JLex

В приложении 1 приведено описание для верификатора, а в приложении 2 для базисного Рефал. Описание разделено на три части.

1. В первой описаны настройки генератора, а также необходимый встраиваемый код. В частности, там есть строка «%cup» означающая, что мы будем использовать генератор парсеров CUP.
2. Во второй описаны макросы через регулярные выражения для обозначения необходимых элементов.
3. В третий части описаны непосредственно лексические правила.

3.3 Реализация синтаксического анализа

Для реализации синтаксического анализатора использовался генератор CUP.

CUP - это система для генерации парсеров LALR из простых спецификаций. Он выполняет ту же роль, что и широко используемая программа YACC, и на самом деле предлагает большинство функций YACC. Однако CUP написан на Java, использует спецификации, включая встроенный Java-код, и создает парсеры, которые реализованы на Java.

Рабочие знания YACC также очень помогают понять, как работают спецификации CUP.

Использование CUP предполагает создание простой спецификации, основанной на грамматике, для которой необходим парсер, а также построение сканера, способного разбивать символы на значащие токены (например, ключевые слова, цифры и специальные символы).

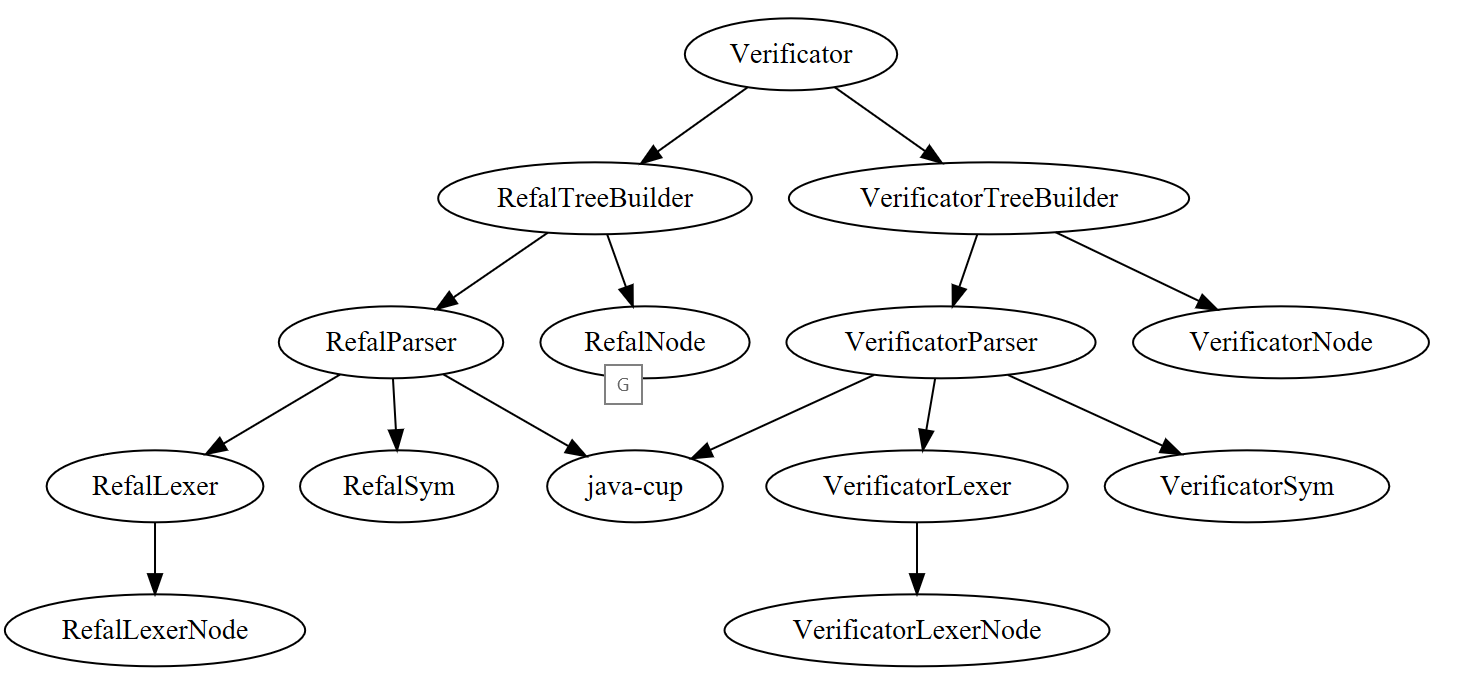
Данный генератор по нотации в форме Бэкуса-Наура создает синтаксический анализатор и использует лексический анализатор для того чтобы, осуществлять проверку кода программы.

В приложении 3 приведено описание для верификатора, а в приложении 4 для Базисный Рефал. В описании есть следующие элементы:

* Встраиваемый код для обработки ошибок
* Перечисление терминалов и не терминалов, а также их типов
* Уровни приоритетов для терминалов
* Правила грамматики

3.4 Представление программ для последующей верификации

Для представления программ используется дерево. В листьях дерева находятся терминалы. С помощью CUP мы строим синтаксическое дерево. Далее мы обходим его и строим абстрактное синтаксическое дерево для Рефала и верификатора.



*Рисунок 2. Иерархия классов*

В описании узла находится либо значение (если это, например, переменная), либо тип (если это, например, круглые скобки). Все терминалы являются экземплярами класса LexerToken и имеют поле name. Далее мы строем отдельные деревья верификатора и Рефала. Эти деревья представляются в виде иерархии классов, которые представлены ниже.

|  |
| --- |
| class Spec {  Type[] types;  Func[] funcs;  }  enum Mode { S, T, E }  class Type {  Mode mode;  string Name;  SimpleType[] constructors;  }  abstract class SimpleType {}  class FixedType : SimpleType {  TermType[] terms;  }  class StretchType : SimpleType {  TermType[] left;  TermType stretch; // серединка, которая итерируется  TermType[] right;  }  // для типа, описанного одной переменной.  class TypeRef : SimpleType {  Mode mode; // всегда Mode.E;  string name;  Type ref; // заполняется при семантическом анализе  }  abstract class TermType {}  abstract class Symbol : TermType {}  class Number : Symbol { long value; }  class Compound : Symbol { string value; }  class Character : Symbol { char value; }  class VarTermType : TermType {  Mode mode; // Mode.T или Mode.S  string name;  Type ref; // заполняется при семантическом анализе  }  Func {  string Name;  SimpleType argument;  SimpleType[] result;  } |

*Листинг 4: Описание классов верификатора в виде псевдокода*

И псевдокод для дерева рефала.

|  |
| --- |
| class AST {   public Function[] functions; }  class Function {   public string name;   public Position pos; }  class Position {   public string file;   public int line, col; }  class Extern : Function {}  class Definition : Function {   public bool is\_entry;   public Sentence[] sentences; }  class Sentence {   public Expression pattern;   public Condition[] conditions;  // пустой список — нет условий   public Expression result;   public Sentence[] block;  // null — если нет блока. }  class Condition {   public Expression result, pattern; }  class Expression {   public Term[] terms; }  class Term {}  class Char : Term {   public char value; }  class Macrodigit : Term {   public long value; }  class CompoundSymbol : Term {   public string value; }  // круглые скобки class StructuralBrackets : Term {   public Term[] content; }  // угловые скобки, только в результатных выражениях class CallBrackets : Term {   public string func\_name;   public Position pos;   public Term[] content; }  class Variable : Term {   public char type; // 's', 't', 'e'   public string index;   public Position pos; } |

*Листинг 4: Описание классов базисного РефалА в виде псевдокода*

Информация об узле храниться в виде LexerToken. Для построения дерева, во время синтаксического анализа мы последовательно обрабатываем токены класса java\_cup.runtime.Symbol (Листинг 6).

|  |
| --- |
| public class Symbol {  public int sym;  public int parse\_state;  boolean used\_by\_parser = false;  public int left, right;  public Object value;  } |

*Листинг 6*

3.5 Верификация

В начале мы строим деревья для входных данных. При построении дерева верификатора, мы проверяем, чтобы все типы должны быть описаны кроме встроенных: s.NUMBER, s.COMPOUND, s.CHAR, s.ANY, t.ANY, e.ANY.

Для каждой функции мы определяем соответствующую функцию из файла описания. Если данной функции нет, выдаем ошибку.

Потом мы по отдельности обрабатываем каждое предложение, в котором мы в начале проверяем левую часть, сопоставляя с образцом, а потом правую часть, помня о типах переменных, полученных из правой части.

1. Тестирование

4.1 Тест 1: Построение абстрактного дерева

Для тестирования была использована программа из листинга 7.

|  |
| --- |
| Sum {  (S t.X) t.Y = <Sum t.X (S t.Y)>;  NULL t.Y = t.Y;  }  Mul1 {  NULL t.Y = NULL;  (S t.X) t.Y = <Sum <Mul1 t.X t.Y> t.Y>;  }  $ENTRY Fact {  NULL = (S NULL);  (S t.Num) = <Mul1 <Fact t.Num> (S t.Num)>;  } |

*Листинг 7*

Для данной программы парсер построил следующее дерево.

|  |
| --- |
| [Sum{  (S T.X ) T.Y = <Sum T.X (S T.Y ) > ;  NULL T.Y = T.Y ;  }  , Mul1{  NULL T.Y = NULL ;  (S T.X ) T.Y = <Sum <Mul1 T.X T.Y > T.Y > ;  }  , $ENTRY Fact{  NULL = (S NULL ) ;  (S T.Num ) = <Mul1 <Fact T.Num > (S T.Num ) > ;  }] |

*Листинг 8*

Для данной программы был составлен верифицирующая документация из листинга 9.

|  |
| --- |
| t.Yum ::= s.ANY;  t.Num ::= NULL | (S t.Num);  <Sum t.Num t.Num> == t.Num;  <Mul1 t.Num t.Num> == t.Num;  <Fact t.Num> == t.Num; |

*Листинг 9*

На выходе мы получили два отдельных связанных дерева: одно для функция и одно для переменных(Листинг 10).

|  |
| --- |
| [T.Yum, constructors=[[S.ANY]]},  T.Num, constructors=[[NULL], [([S, T.Num])]]}]  [<Sum [T.Num, T.Num]> == [T.Num],  <Mul1 [T.Num, T.Num]> == [T.Num], <Fact [T.Num]> == [T.Num]] |

*Листинг 10*

Как видим исходный код восстановлен без ошибок, а значит построение правильное.

4.2 Тест 2: Неописанная альтернатива для типа

|  |
| --- |
| Sum {  (S t.X) t.Y = <Sum t.X (S t.Y)>;  NULL t.Y = t.Y;  }  Mul1 {  NULL t.Y = NULL;  (S t.X) t.Y = <Sum <Mul1 t.X t.Y> t.Y>;  }  $ENTRY Fact {  NULL = (S NULL);  (S t.Num) = <Mul1 <Fact t.Num> (S t.Num)>;  } |

*Листинг 11*

Далее идет текст для верификатора.

|  |
| --- |
| t.Yum ::= s.ANY;  t.Num ::= NULL | (t.Num);  <Sum t.Num t.Num> == t.Num;  <Mul1 t.Num t.Num> == t.Num;  <Fact t.Num> == t.Num; |

*Листинг 12*

Для данной программы получим следующие ошибки.

|  |
| --- |
| Can't match term with termType: (S T.X )  Error, not match with description in Sum{  (S T.X ) T.Y = <Sum T.X (S T.Y ) > ;  NULL T.Y = T.Y ;  }  Can't match term with termType: (S T.X )  Error, not match with description in Mul1{  NULL T.Y = NULL ;  (S T.X ) T.Y = <Sum <Mul1 T.X T.Y > T.Y > ;  }  Can't match term with termType: (S NULL )  Error, not match with description in $ENTRY Fact{  NULL = (S NULL ) ;  (S T.Num ) = <Mul1 <Fact T.Num > (S T.Num ) > ;  } |

*Листинг 13*

В данном случае верификатор не может сопоставить с типом (S NULL ), так как он нигде не описан.

4.3 Тест 3: Неверный тип результата функции

|  |
| --- |
| Sum {  (S t.X) t.Y = <Sum t.X (S t.Y)>;  NULL t.Y = t.Y;  }  Mul1 {  NULL t.Y = NULL;  (S t.X) t.Y = <Sum <Mul1 t.X t.Y> t.Y>;  }  $ENTRY Fact {  NULL = (S NULL);  (S t.Num) = <Mul1 <Fact t.Num> (S t.Num)>;  } |

*Листинг 14*

Далее идет текст для верификатора.

|  |
| --- |
| t.Yum ::= NULL | s.ANY;  t.Num ::= NULL | (S t.Num);  <Sum t.Num t.Num> == t.Num;  <Mul1 t.Num t.Num> == t.Yum;  <Fact t.Num> == t.Num; |

*Листинг 15*

Для данной программы получим следующие ошибки.

|  |
| --- |
| Wrong function result: found [[T.Yum]] but expected [T.Num]  Error, not match with description in Mul1{  NULL T.Y = NULL ;  (S T.X ) T.Y = <Sum <Mul1 T.X T.Y > T.Y > ;  }  Wrong function result: found [[T.Yum]] but expected [T.Num]  Error, not match with description in $ENTRY Fact{  NULL = (S NULL ) ;  (S T.Num ) = <Mul1 <Fact T.Num > (S T.Num ) > ;  } |

*Листинг 16*

4.4 Тест 4: Неверное количество аргументов

В данном случае мы рассматриваем как неверное количество аргументов в левой части, ток и неверное количество аргументов при вызове функции в правой.

|  |
| --- |
| Sum {  (S t.X) t.Y = <Sum t.X (S t.Y)>;  NULL = t.Y;  }  Mul1 {  NULL t.Y = NULL;  (S t.X) t.Y = <Sum <Mul1 t.X> t.Y>;  }  $ENTRY Fact {  NULL = (S NULL);  (S t.Num) = <Mul1 <Fact t.Num> (S t.Num)>;  } |

*Листинг 17*

Далее идет текст для верификатора.

|  |
| --- |
| t.Yum ::= s.ANY;  t.Num ::= NULL ;  <Sum t.Num t.Num> == t.Num;  <Mul1 t.Num t.Num> == t.Num;  <Fact t.Num> == t.Num; |

*Листинг 18*

Для данной программы получим следующие ошибки.

|  |
| --- |
| Invalid number of arguments  Error, not match with description in Sum{  (S T.X ) T.Y = <Sum T.X (S T.Y ) > ;  NULL = T.Y ;  }  Invalid number of arguments  Can't match term with simpleType: <Mul1 T.X > expected [T.Num, T.Num] but found null  Can't match term with simpleType: <Sum <Mul1 T.X > T.Y > expected [T.Num] but found null  Error, not match with description in Mul1{  NULL T.Y = NULL ;  (S T.X ) T.Y = <Sum <Mul1 T.X > T.Y > ;  } |

*Листинг 19*

Как видим, верификатор нашел все ошибки.

4.5 Тест 5: Сопоставление с е-переменной

Рассмотрим функцию с e-переменной.

|  |
| --- |
| F {  t.1 e.2 = e.2;  } |

*Листинг 20*

Далее идет текст для верификатора.

|  |
| --- |
| <F 1 2 3 4> == 2 3 4; |

*Листинг 21*

Данная программа верифицируется без ошибок. Если же мы заменим результат выражения в описании для верификатора с «2 3 4» на «3 4», то получим следующую ошибку.

|  |
| --- |
| Cant match E.2 and [3, 4]  Error, not match with description in F{  T.1 E.2 = E.2 ;  } |

*Листинг 22*

4.6 Тест 6: Сопоставление с растяжимым типом

Рассмотрим функцию с повторяющимися символами

|  |
| --- |
| F { = B B B B; } |

*Листинг 23*

Верификатор опишем через растяжимый тип.

|  |
| --- |
| <F> == B\*; |

*Листинг 24*

Данная программа верифицируется без ошибок, так как идет повторение символа «B».

4.7 Тест 7: Вложение типа в тип

Рассмотрим простейшую функцию.

|  |
| --- |
| F { s.X = s.X; } |

*Листинг 25*

Далее идет текст для верификатора.

|  |
| --- |
| s.AB ::= A | B; <F B> == s.AB; |

*Листинг 26*

Данная программа верифицируется без ошибок. Как видновложение

ЗАКЛЮЧЕНИЕ

Динамическая типизация как в Базисный Рефал дает большое пространство для реализации алгоритмов. Однако такой подход дает большое пространство и для возможных ошибок. В рамках данного диплома был разработан язык для верификации программ на Рефал и верификатор для проверки кода. Сам язык верификатора позволяет описать программу любой сложности, однако верификатор на данном этапе позволяет проверять лишь некоторые программы. Но даже это уже дает возможность проверять программы на наличие ошибок и соответствие заранее продуманным схемам. Для дальнейшего усовершенствования нужно улучшить верификатор для работы с условиями и блоками.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. [Электронный ресурс] http://www.jflex.de/
2. [Электронный ресурс] http://www2.cs.tum.edu/projects/cup/
3. [Электронный ресурс] http://refal.ru/refer\_r5.html
4. [Электронный ресурс] http://refal.botik.ru/book/html/
5. [Электронный ресурс] http://refal.ru/rf5\_frm.htm
6. [Электронный ресурс] https://github.com/bmstu-iu9/simple-refal/blob/master/src/compiler/interfaces.md

Приложение 1

|  |
| --- |
| import java\_cup.runtime.\*;  %%  %class VerificatorLexer %unicode %line %column %cup %{  private Symbol symbol(int type) {  return new Symbol(type, yyline, yycolumn);  }  private Symbol symbol(int type, LexerToken value) {  return new Symbol(type, yyline, yycolumn, value);  }  private LexerToken createToken(String val) {  LexerToken tk = new LexerToken(val);  return tk;  } %} *// Макро* CRLF= \n|\r|\r\n WHITE\_SPACE=[\ \t\f] FIRST\_NAME\_CHAR=[A-Z] NAME\_CHAR=[a-zA-Z\_\-0-9] VARIABLE\_TYPE = "s"|"t"|"e"  %state YYINITIAL  %% */\*лексика\*/* <YYINITIAL> {  "(" { return symbol(sym.LPAREN); }  ")" { return symbol(sym.RPAREN); }  "<" { return symbol(sym.LCHEVRON); }  ">" { return symbol(sym.RCHEVRON); }  "::=" { return symbol(sym.VAREQUAL); }  "==" { return symbol(sym.EQUAL); }  ";" { return symbol(sym.SEMICOLON); }  "\*" {return symbol(sym.MANY);}  "+" {return symbol(sym.ATLEAST);}  "|" {return symbol(sym.OR);}   {VARIABLE\_TYPE}"."{FIRST\_NAME\_CHAR}{NAME\_CHAR}\* { return symbol(sym.VARIABLE, createToken(yytext())); }  {VARIABLE\_TYPE}"."{NAME\_CHAR}+ { return symbol(sym.METAVARIABLE, createToken(yytext())); }  {NAME\_CHAR}+ { return symbol(sym.NAME, createToken(yytext())); }   ({CRLF}|{WHITE\_SPACE})+ {}    . { yybegin(YYINITIAL); } } |

Приложение 2

|  |
| --- |
| import java\_cup.runtime.\*;  %%  %class RefalFiveLexer %unicode %line %column %cup %{  private Symbol symbol(int type) {  return new Symbol(type, yyline, yycolumn);  }  private Symbol symbol(int type, LexerToken value) {  return new Symbol(type, yyline, yycolumn, value);  }  private LexerToken createToken(String val) {  LexerToken tk = new LexerToken(val);  return tk;  }  %} *// Макро* CRLF= \n|\r|\r\n WHITE\_SPACE=[\ \t\f] FIRST\_NAME\_CHAR=[A-Z] NAME\_CHAR=[a-zA-Z\_\-0-9] VARIABLE\_TYPE = "s"|"t"|"e"  MULTILINE\_COMMENT=(("/\*"|"/\*\*")[^"\*"]{COMMENT\_TAIL})|"/\*" COMMENT\_TAIL=([^"\*"]\*("\*"+[^"\*""/"])?)\*("\*"+"/")? END\_OF\_LINE\_COMMENT="/""/"[^\r\n]\*  STRING\_LITERAL=\'([^\\\'\r\n]|{ESCAPE\_SEQUENCE})\*\' ESCAPE\_SEQUENCE=\\[^\r\n]  DIGIT=[0-9] DECIMAL\_INTEGER\_LITERAL={DIGIT}+ INTEGER\_LITERAL={DECIMAL\_INTEGER\_LITERAL} EXTERNAL="$EXTERNAL"|"$EXTERN"|"$EXTRN"  %state YYINITIAL  %%  */\*лексика\*/* <YYINITIAL> {  {EXTERNAL} { return symbol(sym.EXTERN); }  "$ENTRY" { return symbol(sym.ENTRY); }   "," { return symbol(sym.COMMA); }  ";" { return symbol(sym.SEMICOLON); }  ":" { return symbol(sym.COLON); }  "{" { return symbol(sym.LBRACE); }  "}" { return symbol(sym.RBRACE); }  "(" { return symbol(sym.LPAREN); }  ")" { return symbol(sym.RPAREN); }  "<" { return symbol(sym.LCHEVRON); }  ">" { return symbol(sym.RCHEVRON); }  "=" { return symbol(sym.EQUAL); }   {VARIABLE\_TYPE}"."{NAME\_CHAR}+ { return symbol(sym.VARIABLE, createToken(yytext())); }  {FIRST\_NAME\_CHAR}{NAME\_CHAR}\* { return symbol(sym.NAME, createToken(yytext())); }   {STRING\_LITERAL} { return symbol(sym.QUOTEDSTRING, createToken(yytext())); }  {INTEGER\_LITERAL} { return symbol(sym.INTEGER\_LITERAL, createToken(yytext())); }   ({CRLF}|{WHITE\_SPACE}|{END\_OF\_LINE\_COMMENT}|{MULTILINE\_COMMENT})+ {}    . { yybegin(YYINITIAL); } } |

Приложение 3

|  |
| --- |
| package verificator;  import java\_cup.runtime.\*; import java\_cup.runtime.Symbol;  parser code {: public void report\_error(String message, Object info) {  StringBuilder m = new StringBuilder("Error ");   if (info instanceof java\_cup.runtime.Symbol)  m.append( "("+info.toString()+")" );   m.append(" : "+message);   System.out.println(m);  }   public void report\_fatal\_error(String message, Object info) {  report\_error(message, info);  throw new RuntimeException("Fatal Syntax Error");  } :}   */\* Terminals (tokens returned by the scanner). \*/* terminal LexerToken LPAREN, RPAREN, NAME, LCHEVRON, RCHEVRON, SEMICOLON, OR; terminal LexerToken VAREQUAL, EQUAL, VARIABLE, METAVARIABLE, MANY, ATLEAST;  */\* Non terminals \*/* non terminal Node spec, type\_def, func, simple\_types, args\_type, simple\_type, stretch\_type, term\_type, res, f\_simple\_type, n\_simple\_type, end\_type; */\* The grammar rules \*/* precedence left RPAREN, LPAREN, METAVARIABLE, VARIABLE, NAME;  start with spec;  spec ::= type\_def spec  | func spec  |;  type\_def ::= VARIABLE:st VAREQUAL simple\_types SEMICOLON {:RESULT = new Node(st.name, st.start);:};  simple\_types ::= f\_simple\_type  | f\_simple\_type n\_simple\_type;  f\_simple\_type ::= simple\_type;  n\_simple\_type ::= OR simple\_type  | OR simple\_type n\_simple\_type;  simple\_type ::= term\_type  | stretch\_type;  stretch\_type ::= term\_type MANY  | term\_type ATLEAST;  term\_type ::= end\_type  | end\_type term\_type  | LPAREN simple\_type RPAREN {:RESULT = new Node("PARENS", -1);:}  | LPAREN simple\_type RPAREN MANY {:RESULT = new Node("PARENS", -1);:}  | LPAREN simple\_type RPAREN ATLEAST {:RESULT = new Node("PARENS", -1);:};  end\_type ::= VARIABLE:st {:RESULT = new Node(st.name, st.start);:}  | METAVARIABLE:st {:RESULT = new Node(st.name, st.start);:}  | NAME:st {:RESULT = new Node(st.name, st.start);:};   func ::= LCHEVRON NAME args\_type RCHEVRON res SEMICOLON {:RESULT = new Node("FUNC", -1);:};  args\_type ::= simple\_type;  res ::= EQUAL simple\_type  | EQUAL simple\_type res; |

Приложение 4

|  |
| --- |
| package refal;  import java\_cup.runtime.\*;  parser code {: public void report\_error(String message, Object info) {  StringBuilder m = new StringBuilder("Error ");   if (info instanceof java\_cup.runtime.Symbol)  m.append( "("+ info.toString()+")" );   m.append(" : "+message);   System.out.println(m);  }   public void report\_fatal\_error(String message, Object info) {  report\_error(message, info);  throw new RuntimeException("Fatal Syntax Error");  } :}   */\* Terminals (tokens returned by the scanner). \*/* terminal LexerToken SEMICOLON, COLON; terminal LexerToken EXTERN, ENTRY, LBRACE, RBRACE, EQUAL; terminal LexerToken LPAREN, RPAREN, NAME, LCHEVRON, RCHEVRON, QUOTEDSTRING, INTEGER\_LITERAL; terminal LexerToken VARIABLE, COMMA;  */\* Non terminals \*/* non terminal Node expression, term, program, f\_definition, external\_decl, f\_name\_list, block, sentence; non terminal Node left\_side, conditions, arg, result, right\_side; non terminal Node paren\_result\_term, chevron\_result\_term, block\_ending, lefts;  precedence left EQUAL, VARIABLE, NAME, QUOTEDSTRING, INTEGER\_LITERAL, LPAREN; precedence right COLON, EXTERN; precedence right COMMA; */\* The grammar rules \*/* start with program;  term ::= VARIABLE:st {:RESULT = new Node(st.name);:}  | QUOTEDSTRING:st {:RESULT = new Node(st.name);:}  | INTEGER\_LITERAL:st {:RESULT = new Node(st.name);:}  | NAME:st {:RESULT = new Node(st.name);:}  | LPAREN expression RPAREN {:RESULT = new Node("PARENS");:}  | LPAREN RPAREN{:RESULT = new Node("PARENS");:};  paren\_result\_term ::= LPAREN result RPAREN  | LPAREN RPAREN;  chevron\_result\_term ::= LCHEVRON result RCHEVRON;  block\_ending ::= arg COLON LBRACE block RBRACE;  right\_side ::= VARIABLE:st {:RESULT = new Node(st.name);:}  | QUOTEDSTRING:st {:RESULT = new Node(st.name);:}  | INTEGER\_LITERAL:st {:RESULT = new Node(st.name);:}  | NAME:st {:RESULT = new Node(st.name);:}  | paren\_result\_term {:RESULT = new Node("PARENS");:}  | chevron\_result\_term{:RESULT = new Node("CHEVRONS");:};  arg ::= right\_side;  expression ::= term  | term expression;  result ::= right\_side  | right\_side result;  external\_decl ::= EXTERN f\_name\_list;  f\_name\_list ::= NAME:st {:RESULT = new Node(st.name);:}  | NAME:st COMMA f\_name\_list{:RESULT = new Node(st.name);:};  conditions ::= COMMA arg COLON expression  | COMMA arg COLON expression conditions;  left\_side ::= term  | term left\_side;  lefts ::= left\_side conditions  | conditions;  sentence ::= lefts COMMA block\_ending  | lefts EQUAL result  | left\_side COMMA block\_ending  | left\_side EQUAL result  | lefts EQUAL  | left\_side EQUAL  | EQUAL result  | EQUAL;  block ::= sentence  | sentence SEMICOLON  | sentence SEMICOLON block;  f\_definition ::= NAME:st LBRACE block RBRACE{:RESULT = new Node(st.name);:}  | ENTRY NAME:st LBRACE block RBRACE{:RESULT = new Node(st.name);:};  program ::= f\_definition  | f\_definition program  | f\_definition SEMICOLON program  | external\_decl SEMICOLON program  | program external\_decl SEMICOLON; |