**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 Информатика и системы управления

КАФЕДРА Теоретическая информатика и компьютерные технологии

# *РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА*

# *К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ*

# *НА ТЕМУ:*

# *«Вычислитель и верификатор типов функций для языка РЕФАЛ-5»*

| | | |
|---|---|---|
| Студент ИУ9-82 | _____ | Г. М. Иванов |
| (Группа) | (Подпись, дата) | (И.О.Фамилия) |
| Руководитель ВКР | _____ | А.В.Коновалов |
| | (Подпись, дата) | (И.О.Фамилия) |
| Консультант | _____ | _____ |
| | (Подпись, дата) | (И.О.Фамилия) |
| Консультант | _____ | _____ |
| | (Подпись, дата) | (И.О.Фамилия) |
| Нормоконтролер | _____ | _____ |
| | (Подпись, дата) | (И.О.Фамилия) |

*2019 г.*

# Аннотация

Объем дипломной работы составляет 61 страница, на которых размещены 1 рисунок, 2 таблицы и 33 листинга. При написании диплома использовалось 13 источников литературы.

Исследуемой областью являлась верификация типов в языке с динамической типизацией РЕФАЛ-5.

В дипломную работу входит введение, три главы и заключение. Первая посвящена описанию предметной области, во второй формулируется и реализуется алгоритм, а оставшаяся третья посвящена тестированию верификатора.

В введении раскрывается проблематика работы и ее актуальность, определяется цель и постановки задачи.

В заключении описываются результаты проделанной работы и дальнейшие перспективы разработки.

СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

При разработке компьютерной программы часто возникают ошибки, которых не избежать. Размер промышленных программ может варьироваться от несколько тысяч до сотен миллионов строк кода, поэтому сложность разрабатываемой программы подходит к границе её понимания. А стоимость ошибки в данных программах очень высока, что повлечёт за собой полный провал создаваемой программы. Эти ошибки не являются чем-то особенными, так как их причины очевидны (неправильная спецификация, непредвиденные условия работы и другие). Есть различные методы проверки программы – тестирование, верификация. Наиболее распространенный метод тестирования – модульное тестирование (unit testing), которое существует, практически во всех языках программирования. Однако, сложно выявить тестированием редко появляющиеся ошибки в системах реального времени. Верификация – методы доказательства (или опровержения) удовлетворения программы некоторой заданной спецификации. Верификацию чаще всего проводят на моделях программы. Доказать, что программа удовлетворяет некоторым требованиям очень сложно, поэтому при верификации на моделях проверяется не сама программа, а её модель. Данная модель чаще всего проще программы, то есть отражает лишь значимые характеристики. Данная верификация может находить логические ошибки (работает с моделью программы), проводиться до написания программы. Но модель может оказываться иногда неадекватной, поэтому верификация зависит от адекватности модели. Сам язык спецификации может быть неполным для всех требований. [1]

РЕФАЛ (РЕкурсивных Функций АЛгоритмический язык) – это один из старейших функциональных языков программирования, который ориентирован на символьные преобразования: обработку символьных строк (например, алгебраические выкладки), перевод с одного языка (искусственного или естественного) на другой. РЕФАЛ был впервые реализован в 1968 году в России

Валентином Турчиным. Соединяет в себе математическую простоту с практической ориентацией на написание больших и сложных программ. [3]

В 1978 году была опубликована диссертация С. А. Романенко «Машинно-независимый компилятор с языка рекурсивных функций». [2] В 1970-1990 годах было реализовано несколько различных диалектов этого языка, которые используются в настоящее время (РЕФАЛ-2, РЕФАЛ-5, РЕФАЛ+). В МГТУ имени Н.Э. Баумана на кафедре ИУ9 также был разработан диалект языка (РЕФАЛ-5$\lambda$) [4], а компилятор некоторых диалектов языка (РЕФАЛ-5, Простой РЕФАЛ, РЕФАЛ-5$\lambda$) используется в качестве учебного полигона для курсовых и выпускных квалификационных работ.

Основа РЕФАЛа состоит в сопоставлении с образцом (pattern matching), подстановке и рекурсивном вызове, поэтому легко реализовать символьную обработку. Можно заметить то, что программа на РЕФАЛе во много раз короче по объему, чем программа, написанная на языке LISP, и гораздо более читаема и понятна программисту. РЕФАЛ концептуально прост. В РЕФАЛе сопоставление с образцом происходит в прямом направлении, что делает простым для написания различных алгоритмов и программ. Основной структурой данных в РЕФАЛе – это объектное выражение, которое в отличие от списков читается в разных направлениях. В базисном Рефале выражаются через вспомогательные функции конструкции вида: if, while, for, switch.

Функция в РЕФАЛе принимает один аргумент и возвращает одно значение, и аргумент, и значение являются объектным выражением. А если передать в функцию множество (более одного) аргументов, то тогда из них сформируется одно объектное выражение, которое будет содержать в себе все аргументы, а уже в определении функции будет разбираться и сопоставляться с левыми частями предложений. Поэтому можно сделать вывод о том, что РЕФАЛ – динамический язык, и что в РЕФАЛе могут возникать ошибки с типом аргументов – это передача в функцию аргументов, не входящих в область определения функции, а в таких случаях происходит аварийная остановка программы.

Таким образом, вытекает цель моей работы. **Целью** ВКР является разработка верификатора типов в РЕФАЛ-5, который будет проверять корректность вызовов функций в программе, т.е. аргумент в любом вызове функции должен быть совместим с форматом аргумента. В ходе ВКР должны быть выполнены следующие **задачи**:

- Обзор предметной области;
- Разработка алгоритма вывода типов для полного РЕФАЛ-5;
- Реализация алгоритма вывода типов для полного РЕФАЛ-5;
- Тестирования работоспособности приложения.

Благодаря верификатору производится проверка корректности программ, а также повышение местности и ко-местности функций, что полезно при векторном [8] и векторно-списковом [9] представлении данных. Форматы встроенных функций будут вшиты в верификатор. Нашей основной задачей будет построить аппроксимации областей определения и областей значений функций в виде жёстких выражений. А потом проверить соответствие вызовов функций их областям определения.

Форматом мы будем называть способ упаковки нескольких значений в одно объектное выражение, а формат функции будет состоять из формата входных параметров и выходных параметров.

Пусть данная программа, написанная на РЕФАЛ-5:

Листинг 1. Пример программы на РЕФАЛ-5

```
F {
 (e.X) e.Y = /* пусто */;
  e.X = <G e.x>;
}

G {
  s.1 e.2 = /* пусто */;
  t.3 = /* пусто */;
}
```

Для функции *G* очевиден будет формат функции: *G t e = /* пусто */*, на основе входных параметров. А вот для функции *F* аналогичный вывод мы сделать не можем, так как во втором предложении вызывается функция *G*, которая должна сопоставляться с входным её форматом. В результате чего формат функции *F* такой: *F t e = /* пусто */*.

Актуальность данной ВКР:

- Статически проверять корректность программ на РЕФАЛе, что актуально при написании больших программ. Например, суперкомпилятор SCP4 содержит 28 тыс. строк кода, а MSCP-A — 12 тыс. строк.
- Автоматически повышать местность программ, что актуально при использовании массивов для представления объектных выражений.
- Автоматически повышать местность программ, что актуально для некоторых высокоуровневых оптимизаций.
- Автоматически повышать местность программ, что полезно при суперкомпиляции.

Заинтересованность в этой теме возникает по причине того, что РЕФАЛ является отечественным языком программирования, из-за этого можно сделать вывод о том, что данная ВКР является некоторым вкладом в развитие информационных технологий Российской Федерации.

Входной язык верификатора будет является классический РЕФАЛ-5. Почему не был выбран РЕФАЛ-5$\lambda$? В РЕФАЛ-5$\lambda$ есть вложенные функции, которые серьёзно усложнят анализ. Почему не РЕФАЛ-6, РЕФАЛ-7? В них есть некоторые неуспехи, которые тоже усложнят анализ. Почему не РЕФАЛ+? В нём уже есть явная статическая типизация – типы всех функций должны быть явно описаны пользователем. Далее будет понятно, что мы хотим неявную статическую типизацию для РЕФАЛа-5.

# 1. Обзор предметной области

## 1.1. Основные понятия

Каждый диалект языка РЕФАЛ отличается друг от друга деталями синтаксиса и также «дополнительными возможностями», расширяющий первоначальный вариант. Данный вариант называют еще также Базисным Рефалом. В нем определен набор базовых понятий, использующихся во многих других диалектах языка.

Язык Рефал работает с символьными данными. Символ (атом) в РЕФАЛе – минимальная структура данных. Символы бывают простые и составные. Простой символ – это обычный символ. При записи программы простой символ обособляется апострофами, за исключением самого апострофа. Составной символ – это либо идентификатор, либо макроцифра. Идентификатор – это последовательность букв, цифр и знаков -, _. Длина идентификатора не ограничена. Макроцифра – это последовательность цифр, число. Число записывается в обычной десятичной системе счисления. «Длинная» арифметика реализована в РЕФАЛ, а это означает, что большие числа трактуются как последовательность цифр, если число превышает $2^{32}$. 1 2 означает число $1 \cdot 2^{32} + 2$. Последовательность символов – это несколько символов, идущих подряд. При записи последовательности простых символов в программе они разделяются пробелами.

Свободная переменная – переменная символа (s-переменная), терма (t-переменная), выражения (e-переменная). Свободные переменные записываются по формату: тип переменной, символ «точка», индекс переменной (любая непустая последовательность из латинских букв, цифр и знаков - (дефис) и _ (прочерк), как и в случае с именами, последние два символа эквивалентны, индексы чувствительны к регистру). Переменная символа (s-переменная) может принимать значение только одного символа любого. Переменная терма (t-переменная) может принимать значение любого терма, т. е. либо одного символа, либо одного объектного выражения, заключенного в структурные скобки.

Переменная выражения (e-переменная) может принимать значение любого объектного выражения (в том числе и пустым).

Две переменные называются повторными в выражении, если их типы и индексы совпадают. Например, s.Var и s.Var являются повторными переменными, но s.Var и t.Var, s.Var и s.var – неповторные (разные индексы).

Объектное выражение – это выражение, состоящее из атомов, а также из левых и правых структурных скобок «(» и «)». Скобки должны образовывать правильную скобочную структуру. Выражение может быть и пустым.

Образцовое выражение – это выражение, состоящее из переменных, символов (атомов), но без вызовов функций, которое задаёт некоторое множество объектных выражений (элементы множества получаются путём подстановки на место переменных допустимых объектных выражений). Результатное выражение – это выражение, состоящее из переменных, символов (атомов), и вызовов функций (конкретизация). Конкретизация (вызов функции) обозначается при помощи двух угловых скобок, заголовка (имя вызываемой функции) и аргументов функции.

Задание семантики РЕФАЛ-программ описывается через определение абстрактной виртуальной машины, называемой «РЕФАЛ-машина». «РЕФАЛ-машина» имеет два бесконечных хранилища информации: поле программы и поле зрения. РЕФАЛ-программа изначально загружается в поле программы «РЕФАЛ-машины» и не изменяется в ходе выполнения. В поле зрения хранятся выражения без свободных переменных. «РЕФАЛ-машина» выполняет РЕФАЛ-программы пошагово, которая демонстрирует вычисление РЕФАЛ-функции, что удобно для трассировки. В результате выполнения процедуры конкретизации, функция возвращает результат (объектное выражение, которое не содержит свободных переменных и термы конкретизации), который заменяет данный терм конкретизации в поле зрения РЕФАЛ-машины. Вложенные термы конкретизации выполняются последовательно, начиная с самого внутреннего терма (аппликативный порядок), а термы конкретизации, находящиеся на одном уровне вложенности, выполняются параллельно.

Программа на РЕФАЛе состоит из последовательности программных элементов — определений функций и ссылок на функции, определённые в других программах (через ключевые слова «$EXTRN», «$EXTERN», «$EXTERNAL»). Переводы строк эквивалентны обычным пробельным символам (пробел или табуляция). Пробельные символы можно вставлять между двумя любыми символами (атомами). Пробелы обязательны лишь только в том случае, когда два атома, записываемые подряд, могут интерпретироваться как один атом (например, два числа, записанные слитно, будут интерпретироваться как одно число, иначе же это одно число, но уже с другим «намного» большим значением). Пробелы недопустимы внутри идентификаторов. Пробелы внутри цепочек литер интерпретируются как образы литер со значением «пробел». Глобальная регулярная функция представляет собой именованный блок из набора предложений, который может начинаться с ключевого слова «$ENTRY». Сама РЕФАЛ-функция представляет собой упорядоченный набор предложений, состоящих из образца и результата, разделённых символом «=». Все предложения заключены в фигурные скобки, а каждое предложение заканчивается на символ «;». РЕФАЛ-программа также может комментирующий строки. Однострочный комментарий начинается с символа «*», а многострочный комментарий – с символов «/*» и заканчивается символами «*/».

В РЕФАЛе существует единственный способ анализа объектного выражения — сопоставление с образцом. Опишем понятие процедуры сопоставления с образцом. При вызове функции, сопоставляется входное выражение с образцом первого предложения. Если сопоставление удачно, то результатом вызова функции будет правая часть первого предложения. Если сопоставление неудачно, то происходит переход ко второму предложению. Эти шаги повторяются до тех пор, пока есть предложения в функции. Если ни одно предложение не сопоставилось с аргументом вызываемой функции, то программа аварийно завершается с ошибкой сопоставления (RECOGNITION_IMPOSSIBLE). Иногда чтобы избежать этой ошибки,

добавляют предложение, с образцом которого можно сопоставить любое выражение (образец предложения эквивалентен e-переменной). Сопоставление с образцом необходимо в том случае, если необходимо извлечь из аргумента (объектного выражения) его составные части (отдельные значения). Операция сопоставления обозначается так: $E : P$, где $E$ – аргумент, $P$ – образец операции сопоставления, а символ «:» - знак сопоставления. Также можно говорить то, что выражение $E$ распознается как частный случай $P$.

Приведем простой пример программы на РЕФАЛ-5 (функция, подсчитывающая количество букв A во входном выражении).

Листинг 2. Пример программы на РЕФАЛ-5.

```
F {
    s.Count e.Items A =
        <F <Add s.Count 1> e.Items>;
    s.Count e.Items s.Other =
        <F s.Count e.Items>;
    s.Count /* пусто */ = s.Count;
}
```

Для разделения двух подвыражений, требуется только одна пара структурных скобок, поэтому имеется «некая» свобода в выборе форматов:

$$< F\ (e.1)\ e.2 >$$

$$< F\ e.1\ (e.2) >$$

$$< F\ (e.1)\ (e.2) >$$

### 1.2. Разработка языка описания типов

Продолжим рассматривать форматы функций на основе РЕФАЛ+. В этом диалекте языка РЕФАЛ уже есть явная статическая типизация – типы всех функций должны быть явно описаны пользователем. Рассмотрим её более подробно.

Если говорить формально, то все функции в языке РЕФАЛ+ одного аргумента. Но чаще всего мы заранее знаем, формат входного аргумента и формат результата. Например, функция *Add* получает в виде аргумента объектное выражение, которое состоит из двух символов (атомов), и всегда возвращает результат также в виде одного объектного выражения, которое состоит из одного символа.

Ограничения, которые накладываются на формат аргументов и результатов функций, описываются при помощи объявлений спецификации функций. Объявление спецификации функции *Add* имеет вид:

$$\$func\ Add\ sX\ sY = sZ;$$

В общем случае объявление функции *F* имеет вид:

$$\$func\ F\ F_{in} = F_{out};$$

где $F_{in}$ – входной формат функции, а $F_{out}$ – ее выходной формат. Форматы функции могут содержать атомы (символы), структурные скобки и свободные переменные. Индексы переменных в этом случае никакой информации не несут, чаще всего используются просто в виде комментариев и могут опускаться.

Формат аргумента и результата функции должны быть «жёсткими» выражениями, иначе задача будет являться неразрешимой. Жёсткое выражение – это образцовое выражение без открытых и повторных переменных, т. е. на одном уровне скобок может находиться не более чем одна e-переменная. Жёсткое выражение P является уточнением жёсткого выражения Q – P получается из Q путём подстановки на место переменных допустимых жёстких выражений. Q обобщает P – отношение, обратное уточнению. Аппроксимация множества объектных выражений жёстким выражением – такое жёсткое выражение P, которое включает в себя все выражения из множества, но при этом не существует такого выражения R, уточняющего P, что R также будет включать в себя все выражения из множества.

Все образцы и результаты функции в РЕФАЛ+ должны иметь структуру, которая предписана объявлением спецификации. Объявление спецификации функции должно быть предопределено до ее первого использования в каком-либо результатном выражении в программе. Если функция определена в самой программе, ее объявление появляется в тексте программы явно.

При компиляции программы производится проверка во всех вызовах функций, а именно структура аргумента вызываемой функции соответствует ее входному формату. Например, результатное выражение

$$< Add\ 2 < Add\ sX\ sY >>$$

построено правильно. Проверяя корректность внешнего вызова, необходимо использовать информацию о формате результата функции *Add*. А именно, заменяем внутренний терм конкретизации $< Add\ sX\ sY >$ на формат результата функции *Add*, после чего получается выражение $< Add\ 2\ sZ >$. Заметим, что аргумент вызова функции соответствует входному формату функции Add. Но если же мы напишем в программе результатное выражение

$$< Add\ 2 < Add\ sX\ sY > 3 >$$

то оно будет считаться ошибочным, поскольку аргумент внешнего вызова функции *Add* состоит из трех символов, а не из двух, как описано в спецификации.

Таким образом, информация о входных и выходных форматах функций помогает находить многие ошибки в программе еще на стадии компиляции. Это упрощает написание и тестирование программы.

Изучив статическую типизацию РЕФАЛ+ [10], базисом для собственного языка типов в динамическом языке РЕФАЛ-5 будем иметь ввиду статическую типизацию в РЕФАЛ+, а именно для определения типа функции для верификатора спецификация функции будет иметь вид:

$$func\_name\ format\_in = format\_out;$$

### 1.3. Лексический и синтаксический анализы

Для начала определим основные определения для лексического анализа.

- Лексический анализ – это фаза компиляции, объединяющая последовательно идущие во входном потоке кодовые точки в группы, называемые лексемами исходного языка.
- Лексический домен – это заданное некоторым формальным способом множество строк. Лексический домен может быть задан с помощью порождающей грамматики. На практике лексический домен часто является регулярным языком.
- Лексема – это фрагмент текста исходной программы, принадлежащий некоторому лексическому домену.

- Токен – это описатель лексемы, представляющий собой кортеж вида $\langle domain, coords, attr \rangle$, где *domain* – лексический домен, которому принадлежит лексема, *coords* – координаты лексемы в тексте программы, а *attr* – атрибут токена.

Опишем основные лексические домены языка РЕФАЛ-5 в виде регулярных выражений, исходя из вышеупомянутых определений базисного РЕФАЛа. В Листинге №3 указаны:

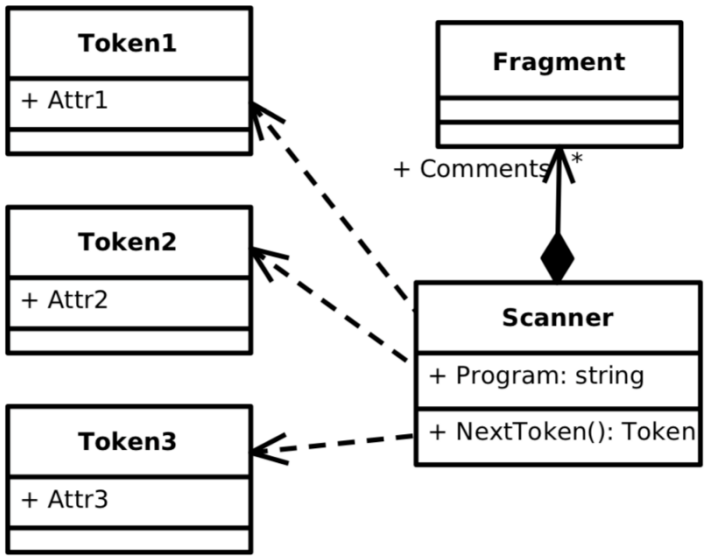
- EXTERN_KEYWORD – это ключевые слова в языке РЕФАЛ-5;
- IDENT – это идентификатор;
- MACRODIGIT – это числа;
- VARIABLE – это свободные переменные;
- COMPOSITE_SYMBOLS – это составные символы;
- CHARS – последовательность литер;
- MARK_SIGN – знаки пунктуации;
- LEFT_CALL_BRACKET – терм конкретизации.

Листинг 3. Основные лексические домены языка РЕФАЛ-5.

```
EXTERN_KEYWORD=\$EXTERN|\$EXTRN|\$EXTERNAL|\$ENTRY
IDENT=[A-Za-z][-A-Za-z_0-9]*
MACROGIDIT=[0-9]*
VARIABLE=[set]\.[-A-Za-z_0-9]+
COMPOSITE\_SYMBOLS=IDENT|([^\\"]|\\([\\nrt"'()<>]|x[0-9a-fA-F]{2}))*
CHARS=([^\\']|\\([\\nrt"'()<>]|x[0-9a-fA-F]{2}))*
MARK_SIGN=[()<>=;:,{}]
LEFT_CALL_BRACKET=<[+-*/\%?]
```

Сам лексический анализатор напишем по UML-диаграмме, представленной на рисунке 1.

Рисунок 1. UML-диаграмма лексического анализатора.




Приведем пример реализации лексического анализатора на Python.

Листинг 4. Лексический анализатор.

```
def next_token(self):
    tok = UnknownToken(self.cur.letter(), Fragment(self.cur, self.cur))
    while tok.tag == DomainTag.Unknown:
        while self.cur.is_white_space():
            self.cur = next(self.cur, self.cur)
        ...
        if self.cur.letter() == "*":
            self.read_comment()
            continue
        if self.cur.is_eof():
            tok = EopToken(Fragment(self.cur, self.cur))
        else:
            if self.cur.letter() == "$":
                read_tok = self.read_keyword()
            ...
            else:
                read_tok = UnknownToken(self.cur.letter(),
                                        Fragment(self.cur, self.cur))
            if read_tok.tag == DomainTag.Unknown:
                sys.stderr.write("Token[" + str(read_tok) + "]:" +
                                 "unrecognized token\n")
                self.cur = next(self.cur, self.cur)
            else:
                self.cur = read_tok.coords.following
                self.cur = next_or_current(self.cur)
                tok = read_tok
    return tok
```

После выделения лексем происходит синтаксический анализ. Синтаксический анализ – это фаза компиляции, группирующая лексемы, порождаемые на фазе лексического анализа, в синтаксические структуры. Задача синтаксического анализа – проверить является ли последовательность лексем предложением в грамматике G. В случае положительного ответа на этот вопрос

следует построить абстрактное дерево вывода последовательности лексем в грамматике G.

Определим BNF грамматику РЕФАЛ-5.

Листинг 5. Грамматика языка РЕФАЛ-5.

```
Program ::= Global*;
Global ::= Externs | Function | ';';
Externs ::= ExternKeyword 'Name' (',' 'Name')* ';';
ExternKeyword ::= '$EXTERN' | '$EXTRN' | '$EXTERNAL';
Function ::= ('$ENTRY')? 'Name' Body;
Body ::= '{' Sentences '}';
Sentences ::= Sentence (';' Sentences?)?;
Sentence ::= Pattern ( ('=' Result) | (',' Result ':' (Sentence | Body)) );
Pattern ::= PatternTerm*;
PatternTerm ::= Common | '(' Pattern ')';
Common ::= 'Name' | ''chars'' | '123' | 's.Var' | 't.Var' | 'e.Var';
Result ::= ResultTerm*;
ResultTerm ::= Common | '(' Result ')' | '<Name' Result '>';
```

Почти аналогично РЕФАЛ+, попробуем определить BNF грамматику языка описания типов функций РЕФАЛ-5.

Листинг 6. Грамматика языка типов РЕФАЛ-5.

```
File ::= Function*
Function ::= 'Name' Format '=' Format ';'
Format ::= Common ('e.Var' Common)?
Common ::= ('Name' | ''chars'' | '123' | 't.Var' | 's.Var' | '(' Format ')')*
```

Для реализации синтаксического анализатора будем использовать метод рекурсивного спуска. В синтаксическом анализаторе, написанном методом рекурсивного спуска, каждому нетерминалу грамматики соответствует отдельная функция, в которой закодирован эффект применения соответствующего нетерминалу правила (мы будем считать, что такое правило единственно). Цель этой функции – анализ последовательности токенов, которые по её запросу выдаёт лексический анализатор, и проверка соответствия этой последовательности правилу грамматики.

Функция, соответствующая нетерминалу X грамматики, составляется с учётом того, что:

- существует глобальная переменная *Sym*, в которую ДО вызова функции помещается токен, соответствующий первому символу цепочки, в которую раскрывается нетерминал *X*;

- функция должна либо полностью потребить последовательность токенов, в которую раскрывается нетерминал *X*, либо сгенерировать сообщение об ошибке;
- после завершения функции переменная *Sym* должна содержать токен, непосредственно следующий за раскрытым нетерминалом *X*.

Приведем пример реализации синтаксического анализатора на Python.

Листинг 7. Синтаксический анализатор языка описания типов функций РЕФАЛ-5.

```
# Common ::= ('Name' | ''chars'' | '123' | 's.Var' | 't.Var' | '(' Format ')')* | ε
def parse_common(self):
    common_term = []
    while self.cur_token.tag == DomainTag.Ident or \
            self.cur_token.tag == DomainTag.Number or \
            self.cur_token.tag == DomainTag.Characters or \
            self.cur_token.tag == DomainTag.Composite_symbol or \
            (self.cur_token.tag == DomainTag.Variable and
             (self.cur_token.value[0] == "s" or
              self.cur_token.value[0] == "t"
             )
            ) or \
            (self.cur_token.tag == DomainTag.Mark_sign and
             self.cur_token.value == "("):
        if self.cur_token.tag == DomainTag.Mark_sign and \
                self.cur_token.value == "(":
            self.cur_token = next(self.iteratorTokens)
            common_term.append(StructuralBrackets(self.parse_format().terms))
            if self.cur_token.tag == DomainTag.Mark_sign and \
                    self.cur_token.value == ")":
                self.cur_token = next(self.iteratorTokens)
            else:
                sys.stderr.write("Expected \")\" after declaring pattern\n")
                self.isError = True
        else:
            if self.cur_token.tag == DomainTag.Ident:
                token = self.cur_token
                self.cur_token = next(self.iteratorTokens)
                common_term.append(CompoundSymbol(token.value))
            ...
    return common_term
```

Для упрощения дальнейшего написания семантического анализа синтаксическое дерево вывода будем хранить в виде структур классов. Обычно для каждого символа грамматики (терминального и нетерминального) придумывают свой класс.

Листинг 8. Структура абстрактного синтаксического дерева РЕФАЛ-5.

```
class AST(object):
    def __init__(self, functions):
        self.functions = [*default_functions, *functions]
...
```

```
class Function(ABC):
    def __init__(self, name, pos):
        self.name = name
        self.pos = pos
...
class Definition(Function):
    def __init__(self, name, pos, is_entry=False, sentences=None):
        super(Definition, self).__init__(name, pos)
        self.is_entry = is_entry
        self.sentences = sentences
...
class Term(ABC):
    def __init__(self, value):
        self.value = value
...
class Variable(Term):
    def __init__(self, value, type_variable, pos, index=-1, sentence_index=-1):
        super(Variable, self).__init__(value)
        self.type_variable = type_variable
        self.index = index
        self.pos = pos
        self.sentence_index = sentence_index
```

## 1.4. Семантический анализ

Семантический анализ – это фаза компиляции, выполняющая проверку синтаксического дерева на соответствие его компонентов контекстным ограничениям. Под контекстными ограничениями мы будем понимать такие вещи, как правила видимости идентификаторов, проверку типов выражений и т.п.

Определим основные проверки семантического анализа в тексте на Рефале-5:

- Все имена функций после символа «<» (скобки вызова) или встроенные, или описанные в текущем файле, или объявленные как $EXTERN;
- Для каждой переменной в результатном выражении должно быть её вхождение в образцовом выражении в предшествующей части предложения. Т.е. при чтении предложения образцовые части пополняют область видимости новыми именами переменных (включая тип, s.Var и t.Var − две разные переменные), в результатных частях могут быть переменные только из области видимости;
- Функция в исходном тексте должна быть определена один раз – или как функция, или как $EXTERN;

- В тексте спецификации типов семантический анализатор должен только проверять, что имена функций не повторяются – нет двух разных определений типов для одной функции. Больше ничего проверять не нужно;
- В паре файлов Рефал-5 и спецификация типов семантический анализ должен проверять, что все функции, для которых заданы спецификации типов, определены в исходном файле.

Приведем пример реализации семантического анализа на Python.

Листинг 9. Семантический анализ языка РЕФАЛ-5.

```
def semantics(self):
    names = set()
    for function in self.ast.functions:
        if function.name in names:
            sys.stderr.write("Error. Function %s already defined\n"
                             % function.name)
            self.isError = True
        else:
            names.add(function.name)

    if not self.isError:
        for function in self.ast.functions:
            if isinstance(function, Definition):
                for sentence in function.sentences:
                    for term in sentence.result.terms:
                        if isinstance(term, CallBrackets):
                            if term.value not in names:
                                sys.stderr.write("Error. Function %s " +
                                                 "isn't defined"
                                                 % term.value)
                                self.isError = True
    if not self.isError:
        for function in self.ast.functions:
            if isinstance(function, Definition):
                self.semantics_variable(function.sentences)
```

# 2. Разработка и реализация алгоритма

## 2.1 Основные понятия

Для описания алгоритма вывода типов функции в РЕФАЛ-5 требуется ввести еще некоторые определения:

- **Подстановка** – множество замен переменных в образцовом выражении соответствующими образцами, замкнутая на множестве жестких образцов.

$$\sigma = \upsilon_i \rightarrow P_i, i = \overline{1 \dots N}$$

- **Уточнение** или сужение образца – операция замены переменных в образце. Например, даны образец $P = t.1\ e.2$ и подстановка $S = \{t.1 \rightarrow (e.4), e.2 \rightarrow s.5\ t.6,\ e.4 \rightarrow t.7\ s.8,\ t.7 \rightarrow s.9\}$, тогда уточнение образца $P_1$ получится подстановкой в образец $P$, а именно $P_1 = (s.9\ s.8\ )\ s.5\ t.6$
- Если при уточнении $P_1 \Rightarrow^+ P_3$ оказалось, что не существует такого образца $P_2$, что $P_1 \Rightarrow^+ P_2 \Rightarrow^+ P_3$, то такое уточнение минимальное: $P_1 \underbrace{\Rightarrow}_{min} P_3$.

Пример минимальных уточнений:

$$e \rightarrow \varepsilon,$$
$$e \rightarrow te,$$
$$e \rightarrow et,$$
$$t \rightarrow (e),$$
$$t \rightarrow s,$$
$$s \rightarrow X;$$

Данные уточнения представляют собой сами определений свободных переменных – s-переменная (переменная символа), t-переменная (переменная терма), e-переменная (переменная выражения). Доказательство того, что не существует других минимальных уточнений, очевидно. Например, для t-переменной возможна подстановка атома, но это можно заменить на последовательность минимальных уточнений: $t \rightarrow s \rightarrow X$. А. Для e-переменной возможна подстановка некоторой последовательностью термов, но это также заменяется последовательностью n длины.

- Сложность жесткого образца – количество минимальных уточнений, переводящих образец вида: $e$ в конкретное выражение. Определяется по формуле:

$$C(P) = n_t + 2 \times n_s + 3 \times n_X + 3 \times n_() - n_e + 1,$$

где $n_s, n_t$, $n_e$ – количество свободных переменных, соответственно, s-переменная (переменная символа), t-переменная (переменная терма), e-переменная (переменная выражения), $n_()$ – количество структурных скобок, $n_X$ – количество атомов.

- Если $P \Rightarrow^+ Q$ и существует 2 способа построения цепочки минимальных уточнений:

$$P \underbrace{\Rightarrow}_{min} R_1^1 \underbrace{\Rightarrow}_{min} \dots \underbrace{\Rightarrow}_{min} R_N^1 \underbrace{\Rightarrow}_{min} Q$$

$$P \underbrace{\Rightarrow}_{min} R_1^2 \underbrace{\Rightarrow}_{min} \dots \underbrace{\Rightarrow}_{min} R_M^2 \underbrace{\Rightarrow}_{min} Q$$

то $M = N$. Из равенства длин цепочек не следует равенство самих цепочек. Цепочки имеют одинаковые начальные и конечные образцы, а также сами длины цепочек, но цепочки минимальных уточнений различны.

## 2.2. Алгоритм вывода типов для базисного РЕФАЛа

Приведем алгоритм вывода типов для базисного РЕФАЛа.

1. Выполняем сквозную нумерацию всех предложений. Ко всем индексам переменных приписываем номер предложения. Для уточнения, индексы на каждой итерации алгоритма будут обновляться.
2. Для каждой правой части строится набор уравнений. Извлекается один из вызовов функции, например *F*. Терм конкретизации заменяется на *out(F)*, аргумент сопоставляется с *in(F)*. Например, для правой части:

$$EA < F\ EB < G\ EC > ED < H\ EF > EG > EH$$

строится система уравнений:

$$\begin{cases} EB\ out(G)\ ED\ out(H)\ EG : in(F) \\ EC : in(G) \\ EF : in(H) \end{cases}$$

3. Начальным форматом всех функций полагаем:

$$in(F) = \perp$$

$$out(F) = \perp$$

где ⊥ – некоторое специальное значение (функция, которая не возвращает никакого значения). Для $EXTERN – форматы берутся из подсказок (для встроенных функций – тоже).

4. Обозначим:

   $Pat(f, i)$ – i-я левая часть функции *f*,

   $Res(f, i)$ – i-я правая часть *f*,

   $\overline{Res(f, i)}$ – i-я правая часть *f* с заменой вызовов функций на *out(g)*.

5. Пытаемся решить систему уравнений. Если у уравнения нет решений, то сообщаем об ошибке (предложение «условно» удаляем для обобщения). Если система имеет единственное решение, берём для переменных соответствующее значение. Если несколько, то тогда учитываем все решения (размножаем предложения).

   Решение уравнения означает то, что:

$$< F\ args > \ \Rightarrow args : in(F) \Rightarrow \ \exists\, s\ args|_s \preccurlyeq in(F)$$

6. Уточняем форматы. Если у функции подсказки нет, строим обобщение:

$$\begin{cases} in(f) = Gen\big(Pat(f, i)\big) \\ out(f) = \ Gen\left(\overline{Res(f, i)}\right) \end{cases}$$

   с подстановками значений переменных. Если у функции есть подсказка:

$$\begin{cases} in(f) = Gen\big(Pat(f, i), help_{in(f)}\big) \\ out(f) = Gen\left(\overline{Res(f, i)}, help_{out(f)}\right) \end{cases}$$

7. Повторяем решение системы уравнений до тех пор, пока *in(f)* и *out(f)* не перестанут меняться.

Разберём более подробно про значение ⊥. Основные свойства:

- $P$ - образцовое выражение, существует подстановка вида:

$$e.X \ \rightarrow \perp \Rightarrow P = \perp;$$

- $P\ \perp = \perp\ P = (\perp) = \perp$ (конкатенация);

- Сложность $\perp = \infty$;
- $Gen(P, \perp) = Gen(\perp, P) = P$.

Первые два свойства похожи на свойства 0 при умножении ($X \times 0 = 0 \times X = 0$), а последние – свойства 0 при сложении ($X + 0 = 0 + X = 0$).

Полный алгоритм обобщённого сопоставления описан в работах Турчина [5][11]. Рассмотрим алгоритм, где не надо рассматривать повторные s-переменные, которые дают «чужие» переменные и расщепления e-переменных, а также введением t-переменным (в базисном РЕФАЛе мы не будем учитывать присваивания). Простыми словами алгоритм можно описать так.

Вот есть сопоставление выражения общего вида $E$ (которое может включать даже вызовы функций) и жёсткого выражения $He$:

$$E : He$$

Жёсткое выражение имеет вид:

$$Ht_1' \dots Ht_n' \; e.X Ht_M'' \dots Ht_1''$$

либо

$$Ht_1 \dots Ht_n,$$

где $Ht$ – жёсткие термы, $e.X$ – e-переменная. Т.е. надо рассмотреть четыре случая:

- жёсткое выражение начинается на жёсткий терм;
- жёсткое выражение кончается на жёсткий терм;
- жёсткое выражение состоит из одной e-переменной;
- жёсткое выражение пустое.

Рассмотрим данные случаи более подробнее.

1. Жёсткое выражение начинается на жёсткий терм ($HtHe'$), где $Ht$ –жесткий терм. Если сопоставляемое выражение имеет вид: $TE'$, где $T$ – некий терм (символ, выражение в скобках, переменные s или t), то выполняем сопоставления соответственно: $T : Ht$ и $E' : He'$.

2. Жёсткое выражение оканчивается на жёсткий терм ($He'Ht$), где $Ht$ – жесткий терм. Аналогично предыдущему случаю.
3. Если уравнение имеет вид $e.X\ E : Ht\ He$, где $Ht$ – не e-переменная, то решаем две системы. В первом делаем подстановку $e.X \rightarrow \varepsilon$ (и уравнение обращается в $E: Ht\ He$), во втором – $e.X \rightarrow t.i\ e.j$ (т.е. уравнение обращается в $t.i\ e.j\ E: Ht\ He$, откуда $t.i : Ht$ и $e.j\ E: He$). Здесь $t.i$ и $e.j$ - переменные с новыми индексами.
4. Случай $E\ e.X: He\ Ht$ решается аналогично предыдущему (две системы с подстановками $e.X \rightarrow \varepsilon$ и $e.X \rightarrow e.i\ t.j$
5. В случае $E : \varepsilon$ если E имеет вид $e.1$ ... $e.N$, то получаем подстановки $e.1 \rightarrow \varepsilon, \dots, e.N \rightarrow \varepsilon$. В противном случае – решений нет.
6. Жёсткое выражение состоит из e-переменной. Присваиваем этой переменной сопоставляемое выражение ($E \leftarrow e.X$).

Сопоставление термов $T : Ht$ происходит по таблице 1.

Таблица 1. Сопоставление термов $T : Ht$.

| $T$ | $Ht$ | | | | |
|---|---|---|---|---|---|
| | $X$ | $Y \neq X$ | $s.X$ | $t.X$ | $(e.X)$ |
| $X$ | тождество | нет решений | $X \leftarrow s.X$ | $X \leftarrow t.X$ | нет решений |
| $s.Y$ | $s.Y \rightarrow X$ | $s.Y \rightarrow Y$ | $s.Y \leftarrow s.X$ | $s.Y \leftarrow t.X$ | нет решений |
| $t.Y$ | $t.Y \rightarrow X$ | $t.Y \rightarrow Y$ | $t.Y \rightarrow s.X$ | $t.Y \leftarrow t.X$ | $\begin{cases} t.Y \rightarrow e.N \\ e.N : e.X \end{cases}$ |
| $(e.Y)$ | нет решений | нет решений | нет решений | $(e.Y) : t.x$ | $e.\ Y : e.X$ |

Приведем пример реализации алгоритма сопоставления на Python.

Листинг 10. Алгоритм сопоставления.

```
def calculate_equation(self, equation, substitution, system):
    if eq.type_equation == EqType.Term:
        term_left = eq.left_part.terms[0]
        term_right = eq.right_part.terms[0]
        if isinstance(term_right, Variable) and \
           isinstance(term_left, Variable) and \
           term_left == term_right:
            return "Success", substitution, system, eq
        if is_symbol(term_left) and \
           is_symbol(term_right) and \
           term_left == term_right:
            return "Success", substitution, system, eq
        elif isinstance(term_right, Variable) and \
             term_right.type_variable == Type.t:
            if term_right.index != -1:
                substitution.append(Substitution(Expression([term_left]),
                                                 Expression([term_right])))
            return "Success", substitution, system, eq
        elif (isinstance(term_right, Variable) and \
              term_right.type_variable == Type.s) or \
              is_symbol(term_right):
            if (isinstance(term_right, Variable) and \
                term_right.index != -1) or \
                not isinstance(term_right, Variable):
                substitution.append(Substitution(Expression([term_left]),
                                                 Expression([term_right])))
            return "Success", substitution, system, eq
        elif isinstance(term_left, StructuralBrackets) and \
             isinstance(term_right, StructuralBrackets):
            eq.type_equation = EqType.Expr
            eq.left_part.terms = Expression(term_left.value).terms
            eq.right_part.terms = Expression(term_right.value).terms
            return self.calculate_equation(eq, substitution, system)
        else:
            return "Failure", [], system, eq
```

Далее разберём алгоритм обобщения. Определяется образец общего вида, анализируя внешний вид отдельных образцов. К этому способу можно отнести стратегию, примененную в суперкомпиляторе SCP4 и стратегию построения ГСО, реализованную ранее в РЕФАЛ-5λ. Стратегию, примененную в РЕФАЛ-5λ, не используем из-за сложности реализации, поэтому используем стратегию в суперкомпиляторе SCP4. [7]

Если у нас несколько (жёстких) образцов, то смотрим на левый и правый край каждого из них:

- если все левые края описывают термы (т.е. являются символами, скобками, s- или t-переменными) и все правые края описывают термы:
    - обобщаются все первые термы, обобщаются все последние термы;
    - выбирается, с какой стороны сложность больше;

- если больше слева – выписываем обобщение левых термов как очередной слева терм обобщения, у всех образцов срезается первый терм;
- если справа – симметрично;

- если слева у хотя бы одного из образцов есть e-переменная, а справа все края описывают термы:
  - обобщаем все последние термы;
  - результат обобщения пишем, как правый край результата;
  - обрезаем у всех образцов правый терм;
- если все левые края описывают термы, а справа хотя бы у одного из образцов есть e-переменная:
  - симметрично предыдущему случаю;
- если слева есть хотя бы у одного образца e-переменная и справа есть хотя бы одна e-переменная:
  - результат обобщения – $e$;
- если все образцы пустые:
  - результат обобщения – $\varepsilon$;
- если есть хотя бы один пустой:
  - результат обобщения – $e$;

Обобщения пары термов по таблице 2:

Таблица 2. Обобщение термов.

| $P_1$ | $P_2$ | | | | |
|---|---|---|---|---|---|
| | $X$ | $Y \neq X$ | $s$ | $t$ | $(P_2)$ |
| $X$ | $X$ | $s$ | $s$ | $t$ | $t$ |
| $s$ | $s$ | $s$ | $s$ | $t$ | $t$ |
| $t$ | $t$ | $t$ | $t$ | $t$ | $t$ |
| $(P_1)$ | $t$ | $t$ | $t$ | $t$ | $Gen(P_1, P_2)$ |

Приведем пример реализации алгоритма обобщения термов на Python.

Листинг 11. Алгоритм обобщения

```
def generalization_term_rec(self, term_left, term_right):
    if is_symbol(term_left) and \
       is_symbol(term_right):
        if term_left == term_right:
            return term_left
        else:
            index = generate_index()
            return Variable("generated%d".format(index), Type.s, None, index)
    if (isinstance(term_left, Variable) and \
        term_left.type_variable == Type.s) and \
        is_symbol(term_right):
        return term_left
    if (isinstance(term_right, Variable) and \
        term_right.type_variable == Type.s) and \
        is_symbol(term_left):
        return term_right
    if (isinstance(term_right, Variable) and \
        term_right.type_variable == Type.s) and \
        (isinstance(term_left, Variable) and \
         term_left.type_variable == Type.s):
        return term_left
    if isinstance(term_left, Variable) and \
       term_left.type_variable == Type.t:
        return term_left
    if isinstance(term_right, Variable) and \
       term_right.type_variable == Type.t:
        return term_right
    if isinstance(term_left, StructuralBrackets) and \
       isinstance(term_right, StructuralBrackets):
        result_terms = self.generalization(
            [Expression(term_left.value),
             Expression(term_right.value)
            ]).terms
        if result_terms == [SpecialVariable("@",SpecialType.none)]:
            return SpecialVariable("@",SpecialType.none)
        else:
            return StructuralBrackets(result_terms)
    if isinstance(term_left, StructuralBrackets):
        index = generate_index()
        return Variable("generated%d".format(index), Type.t, None, index)
    if isinstance(term_right, StructuralBrackets):
        index = generate_index()
        return Variable("generated%d".format(index), Type.t, None, index)
```

Рассмотрим на примерах данный алгоритм вывода типов функций на РЕФАЛ-5.

Например, дана программа в РЕФАЛ-5:

Листинг 12. Пример программы на РЕФАЛ-5

```
F { t.X e.Y = t.X; }
G { e.X t.Y = t.Y; }

H {
  e.A s.B = e.A;
  e.X = <F e.X> <G e.X>;
}
```

Первый шаг итерации очевиден, и мы получим такие форматы функции:

$$in(F) = Gen(t.X\ e.Y) = t\ e$$
$$out(F) = Gen(t.X) = t$$
$$in(G) = Gen(e.X\ t.Y) = e\ t$$
$$out(G) = Gen(t.Y) = t$$
$$in(H) = Gen(e.A\ s.B, \perp) = e\ s$$
$$out(H) = Gen(e.X, \perp) = e$$

На втором шаге итерации алгоритма мы получим два решения (сужения):

$$\begin{bmatrix} e.X \rightarrow t.1 \\ e.X \rightarrow t.1\ e.2\ t.3 \end{bmatrix}$$

Тогда входной формат функций будет таким:

$$in(F) = Gen(t.X\ e.Y) = t\ e$$
$$out(F) = Gen(t.X) = t$$
$$in(G) = Gen(e.X\ t.Y) = e\ t$$
$$out(G) = Gen(t.Y) = t$$
$$in(H) = Gen(e.A\ s.B, t.1, t.1\ e.2\ t.3) = e\ t$$
$$out(H) = Gen(e.X, t.4\ t.5) = e$$

В случае, неразрешимости системы приведем пример:

Листинг 13. Пример программы на РЕФАЛ-5

```
G {(e.1) e.2 = e.1}
F {
  s.3 e.4 = s.3;
  e.5 = <G e.5> <F e.5>;
}
```

Первый шаг итерации очевиден, и мы получим такие форматы функции:

$$in(F) = Gen(s.3\ e.4, \perp) = s\ e$$
$$out(F) = Gen(s.3) = s$$
$$in(G) = Gen\big((e.1)\ e.2\big) = (e)\ e$$
$$out(G) = Gen(e.1) = e$$

На втором шаге итерации алгоритма мы получим систему уравнений вида:

$$\begin{cases} e.5 : in(G) \equiv (e.6)\ e.7 \\ e.5 : in(F) \equiv s.8\ e.9 \end{cases}$$

Можно сказать заранее о том, что данная система не имеет решений, но при этом формат функции правильно аппроксимируется, то есть формат образца функции $F$ будет таким: $in(F) = Gen(s.3\ e.4) = s\ e$. Данная модификация для верификатора («условное» удаление предложений) будет давать правильную

аппроксимацию функции, то есть найдется такая область определения функции, где функция «не падает» (аварийно завершается).

Листинг 14. Пример функции на РЕФАЛ-5

```
F {
    ((((s.X)))) = s.X;
    (e.X) = <F e.X>;
}
```

Предположим, $in(F) = \bot, out(F) = \bot$.

На первой итерации получается уравнение вида $e.X : in(F)$ следовательно с заменой $e.X : \bot$, а значит в итоге выводится подстановка вида $e.X \to \bot$. Теперь формат образца имеет вид при обобщении: $\begin{cases} ((((s)))) \\ (\bot) \equiv \bot \end{cases}$, а значит получается $in(F) = ((((\mathrm{s}))))$. Формат результата: $\begin{cases} s \\ (\bot) \end{cases}$, $out(F) = s$.

На второй итерации $e.X : ((((s))))$, а значит в итоге выводится подстановка вида $e.X \to ((((s))))$.Теперь при обобщении формат образца имеет вид: $\begin{cases} ((((s)))) \\ (((((s))))) \end{cases}$, а значит получается $in(F) = ((((t))))$. Формат результата: $\begin{cases} s \\ s \end{cases}$, а значит получается $out(F) = s$.

Третья итерация – неподвижная точка. Итоговый формат функции:

$$F\ ((((t)))) = s$$

Приведём другой пример.

Листинг 15. Пример программы на РЕФАЛ-5

```
F {
  e.X = <G e.X> <H e.X>;
}

G { s.Y e.Z = s.Y }
H { e.Y (e.Z) = e.Z }
```

Предположим, $in(F) = \bot, out(F) = \bot,\ in(G) = \bot, out(G) = \bot,\ in(H) = \bot, out(H) = \bot$.

Первая итерация очевидна.

$$in(F) = Gen(e.X) = e$$
$$out(F) = Gen(\perp\ \perp) = \perp$$
$$in(G) = Gen(s.Y\ e.Z) = s\ e$$
$$out(G) = Gen(s.Y) = s$$
$$in(H) = Gen(e.Y\ (e.Z)) = e\ (e)$$
$$out(H) = Gen(e.Z) = e$$

На втором шаге итерации, получаем систему: $\begin{cases} e.X : s.1\ e.2 \\ e.X : e.3\ (e.4) \end{cases}$.

Первое уравнение имеет вид $e.X\ E : Ht\ He$, строим две подстановки: $e.X \rightarrow \varepsilon$ и $e.X \rightarrow t.5\ e.6$:

$$\left[\begin{array}{l} \begin{cases} \varepsilon : s.1\ e.2 \\ \varepsilon : e.3\ (e.4) \end{cases} \\ \begin{cases} t.5\ e.6 : s.1\ e.2 \\ t.5\ e.6 : e.3\ (e.4) \end{cases} \end{array}\right.$$

Первая система несовместна, поскольку оба уравнения решений не имеют. Отбрасываем её. Для первого предложения второй системы есть правило $T\ E : Ht\ He'$:

$$\begin{cases} t.5 : s.1 \\ e.6 : e.2 \\ t.5\ e.6 : e.3\ (e.4) \end{cases}$$

Первое уравнение даёт подстановку $t.5 \rightarrow s.7$, подставляем её:

$$\begin{cases} s.7 : s.1 \\ e.6 : e.2 \\ t.5\ e.6 : e.3\ (e.4) \end{cases}$$

Присваивания нам не нужны, поскольку с ними ничего не делаем, их отбрасываем. Первые два уравнения разрешаются в присваивания:

$$\{s.7\ e.6 : e.3\ (e.4)$$

Последнее (и единственное) уравнение имеет вид $E\ e.6 : He'Ht$ рассматриваем две подстановки $e.6 \rightarrow \varepsilon$ и $e.6 \rightarrow e.8\ t.9$:

$$\left[\begin{array}{l} \{s.7 : e.3\ (e.4) \\ \{s.7\ e.8\ t.9 : e.3\ (e.4) \end{array}\right.$$

Уравнения обоих систем имеют вид $E\ T : He'Ht$, разделяем каждое на два $T : Ht$ и $E : He$.

$$\begin{bmatrix} \begin{cases} s.7 : (e.4) \\ \varepsilon : e.3 \end{cases} \\ \begin{cases} t.9 : (e.4) \\ s.7\ e.8 : e.3 \end{cases} \end{bmatrix}$$

Первая система несовместна, поскольку уравнение $s.7 : (e.4)$ не имеет решений, отбрасываем систему. Первое уравнение второй системы даёт сужение $t.9 : (e.10)$, подставляем его:

$$\begin{cases} (e.10) : (e.4) \\ s.7\ e.8 : e.3 \end{cases}$$

Первое уравнение преобразуем по правилу $(E) : (He) \to E : He$:

$$\begin{cases} e.10 : e.4 \\ s.7\ e.8 : e.3 \end{cases}$$

Два последних уравнения всегда успешны, разрешаются в присваивания, а поскольку присваивания нам не нужны, мы их просто отбрасываем. Получаем решение:

$$\begin{bmatrix} e.X \to t.5\ e.6 \\ t.5 \to s.7 \\ e.6 \to e.8\ t.9 \\ t.9 \to (e.10) \end{bmatrix}$$

Если эти подстановки мы последовательно применим к левой части (которая имеет вид $e.X$), получим формат левой части $s.7\ e.8\ (e.10)$.

Итоговые форматы функции:

$$F\ s\ e\ (e) = s\ e$$
$$G\ s\ e = s$$
$$H\ e\ (e) = e$$

А вот теперь рассмотрим функцию с ошибкой.

Листинг 16. Пример ошибочной программы на РЕФАЛ-5

```
F {
  A A A A e.X = <F A A e.X>;
  A = A;
}
```

Предположим, $in(F) = \bot, out(F) = \bot$. Для первого предложения находим, что $e.X \to \bot$ , поэтому он не участвует в выводе формата. Выводим формат по последнему предложению, получается $A = A$.

На второй итерации решаем первое уравнение: $A\ A\ e.X : A$.

Решений нет! Потому что уравнение несовместно. Оно же является неподвижной точкой. На стадии верификации первое уравнение не будет иметь решений – верификатор обнаружит ошибку в функции.

Проведем анализ с работой Романенко. У Романенко, как и у нас, вводится полурешётка – множество с вершиной и дном. Романенко рассматривает в статье вывод типов для варианта Лиспа, вывод осуществляет на основе абстрактной интерпретации. На каждой итерации типы функций уточняются. Также описано, что всем функциям, кроме первой типы параметров задаются как «дно», первой функции – «верхушка», any. Если параметр функции в неподвижной точке остался дном, то эта функция никогда не вызывается. Тип выводится не от семантики самой функции, а только от её использования. Условно, если функция всегда получает вторым параметром список из трёх элементов, а третьим – константу (одну и ту же во всех вызовах), то второй параметр можно заменить тремя параметрами, а третий вообще выкинуть.

Такой повышатель местности возник в контексте частичных вычислений – в остаточной программе оставалась функция из исходной программы, но она всегда вызывалась с одним и тем же аргументом по структуре и эту структуру разбирала. Поэтому логично было разделить параметр на несколько. [6][12]

Также есть важный нюанс в выводе типов и верификации. Нюанс в том, что обе процедуры нужно выполнять отдельно и по-разному реагировать на несовместные системы. Если для некоторого предложения на стадии вывода типов мы получили несовместную систему, то всем переменным назначаем значение «дно», ⊥. Соответственно, левая и правая части становятся дном и не участвуют в обобщении. На второй стадии, верификации, если уравнение решить не удалось, то сообщаем об ошибке в некотором предложении.

### 2.3. Модификация алгоритма вывода типов в полном РЕФАЛ-5

Для полного РЕФАЛ-5 потребуется дополнительное понятие такое, как присваивания. Присваивания отличаются от сужений тем, что они применяются параллельно к образцу, а не последовательно в случае сужений. Но при этом,

присваивание – это такая же операция замены переменных, только записывается иначе:

$$\sigma = P_i \leftarrow v_i, i = \overline{1 \dots N}$$

Теперь при решении системы уравнений присваивания не отбрасываем, а сохраняем для дальнейшего решения.

Условия в РЕФАЛ-5 для алгоритма будем рассматривать как уравнения, и рассматривать будем лишь только те условия, которые являются жёсткими образцами (где все переменные новые, нет повторных и открытых переменных).

Пусть дано предложение

$$Pat,\ R_1 : P_1 = Res;$$

Тогда для этого случая модифицируем:

1. Решаем уравнение из $R_1$ (уравнения для вызовов функций из $R_1$). Получаем набор решений. Для каждого из решений создаём новый экземпляр предложения с подстановкой $Pat', R_1' : P_1' = Res'$;
2. «Ожесточаем» $P1'$ - делаем из него жёсткое выражение $H[P_1']$. Определим функцию ожесточения $H$:
    - $H[sym\ E] = sym\ H[E]$
    - $H[E\ sym] = H[E]\ sym$
    - $H[st.var\ E] = st.var\ H[E]$ (индекс переменной сохраняется)
    - $H[E\ st.var] = H[E]\ st.var$ (индекс переменной сохраняется)
    - $H[(E')\ E''] = (H[E'])\ H[E'']$
    - $H[E'(E'')] = H[E'](H[E''])$
    - $H[e.index] = e.index$ (индекс переменной сохраняется)
    - $H[\varepsilon] = \varepsilon$
    - $H[E] = e.new$ (новый индекс переменной)

Если $H[P_1']$ содержит повторные переменные, то данный случай не рассматривается, иначе же решаем уравнение $\overline{R_1}' : H[P_1']$. Получаем набор сужений и присваиваний. Если в присваиваниях и сужениях одинаковые

переменные, то также не рассматриваем. Если их нет, применяем к предложению сужения и присваивания: $Pat'', R_1'' : P_1'' = Res''$;

1. После, используем алгоритм вывода для базисного РЕФАЛа, то есть решаем уравнения для вызовов функции из $Res''$, получаем набор решений подставляем в предложение: $Pat''', R_1''' : P_1''' = Res'''$;
2. Значит формат образца получается из $Pat'''$, а формат результата из $\overline{Res'''}$

Приведем реализацию функции ожесточения на Python.

Листинг 17. Функция ожесточения.

```
def make_hard(self, expr):
    if len(expr.terms) == 0:
        return Expression([])
    elif is_symbol(expr.terms[0]):
        return Expression([expr.terms[0],
               *self.make_hard(Expression(expr.terms[1:])).terms])
    elif is_symbol(expr.terms[-1]):
        return Expression([
               *self.make_hard(Expression(expr.terms[:-1])).terms,
               expr.terms[-1]])
    elif isinstance(expr.terms[0], Variable) and \
         (expr.terms[0].type_variable == Type.s or \
          expr.terms[0].type_variable == Type.t):
        return Expression([expr.terms[0],
               *self.make_hard(Expression(expr.terms[1:])).terms])
    elif isinstance(expr.terms[-1], Variable) and \
         (expr.terms[-1].type_variable == Type.s or \
          expr.terms[-1].type_variable == Type.t):
        return Expression([
               *self.make_hard(Expression(expr.terms[:-1])).terms,
                expr.terms[-1]])
    elif isinstance(expr.terms[0], StructuralBrackets):
        return Expression([StructuralBrackets([
            *self.make_hard(Expression(expr.terms[0].value)).terms]),
            *self.make_hard(Expression(expr.terms[1:])).terms])
    elif isinstance(expr.terms[-1], StructuralBrackets):
        return Expression([
            *self.make_hard(Expression(expr.terms[:-1])).terms,
            StructuralBrackets([
            *self.make_hard(Expression(expr.terms[-1].value)).terms])])
    elif len(expr.terms) == 1 and \
         isinstance(expr.terms[0], Variable) and \
         expr.terms[0].type_variable == Type.e:
        return Expression([expr.terms[0]])
    elif len(expr.terms) == 1 and \
         isinstance(expr.terms[0], SpecialVariable) and \
         expr.terms[0].type_variable == SpecialType.none:
        return Expression([expr.terms[0]])
    else:
        index = generate_index()
        e_variable = Variable("%d".format(index), Type.e, None, index)
        return Expression([e_variable])
```

В случае, приведем пример программы с условиями:

Листинг 18. Пример программы с условиями на РЕФАЛ-5

```
F {
    (e.X) (e.Y)
    , <G e.X> : e.Z
    , e.X e.Y : s.X s.Y s.Z
    = e.X e.Z e.Y ;
}

G {
    s.X e.Y = s.X ;
}
```

Предположим, $in(F) = \bot, out(F) = \bot, in(G) = \bot, out(G) = \bot$.

Первая итерация очевидна.

$$in(F) = Gen(\bot) = \bot$$
$$out(F) = Gen(\bot) = \bot$$
$$in(G) = Gen(s.X\ e.Y) = s\ e$$
$$out(G) = Gen(s.X) = s$$

На второй итерации, первым шагом мы решаем уравнение: $e.X : in(G) \equiv s.5\ e.6$. Получаем сужение: $e.X \rightarrow s.5\ e.6$. Далее решаем уравнение условий: $out(G) \equiv s.7 : e.Z$. Также получаем присваивание: $s.7 \leftarrow e.Z$. Во втором случае, уравнение будет иметь вид: $e.X\ e.Y : s.1\ s.2\ s.3$. Применяя подстановки, получаем: $s.5\ e.6\ e.Y : s.1\ s.2\ s.3$. Уже при решении нового уравнения имеем решения:

$$\left[\begin{cases} s.5 \leftarrow s.1 \\ e.6 \rightarrow \varepsilon \\ e.Y \rightarrow s.2\ s.3 \end{cases}\right.$$
$$\left[\begin{cases} s.5 \leftarrow s.1 \\ e.6 \rightarrow s.2 \\ e.Y \rightarrow s.3 \end{cases}\right.$$
$$\left[\begin{cases} s.5 \leftarrow s.1 \\ e.6 \rightarrow s.2\ s.3 \\ e.Y \rightarrow \varepsilon \end{cases}\right.$$

Подставляя все решения в изначальную функцию, получаем функцию вида:

$$(s.5)\ (s.2\ s.3)\ \ldots = s.5\ s.7\ s.2\ s.3$$
$$(s.5\ s.2)\ (s.3)\ \ldots = s.5\ s.2\ s.7\ s.3$$
$$(s.5\ s.2\ s.3)\ (\,)\ \ldots = s.5\ s.2\ s.3\ s.7$$

При обобщении входных и выходных форматов функций получаем:

$$in(F) = Gen(\,(s.5)(s.2\ s.3), (s.5\ s.2)(s.3), (s.5\ s.2\ s.3)(\,)\,) = (s\ e)\ (e)$$

$$out(F) = Gen(\,s.5\ s.7\ s.2\ s.3, s.5\ s.2\ s.7\ s.3, s.5\ s.2\ s.3\ s.7\,) = s\ s\ s\ s$$

Это и будет являться неподвижной точкой. Значит формат функции $F$ будет иметь вид такой:

$$F\ (s\ e)\ (e) = s\ s\ s\ s$$

В случае блоков нужно сделать преобразование. Пусть дано предложение

$$Pat,\ R_1 : P_1,\ Res : \{\ P_{11} = R_{11}; P_{21} = R_{21};\}$$

Преобразуем его в предложения с условиями:

$$Pat,\ R_1 : P_1,\ Res : P_{11} = R_{11};$$

$$Pat,\ R_1 : P_1,\ Res : P_{21} = R_{21};$$

Далее, применяем алгоритм вывода для полного РЕФАЛа (условия).

Насчёт данного преобразования нельзя сказать, что оно является эквивалентным по причине того, что блоки в РЕФАЛе работают иначе. Но для данного алгоритма это преобразование можно считать приемлемым.

Приведём пример реализации преобразования блоков в условия в языке РЕФАЛ-5.

Листинг 19. Преобразование блоков в условия в РЕФАЛ-5

```
def block_to_condition(self):
    for function in self.ast.functions:
        if isinstance(function, Definition):
            sentence_result_ = []
            while len(function.sentences) > 0:
                new_sentence = deepcopy(function.sentences[0])
                del function.sentences[0]
                if new_sentence.block:
                    for sentence_block in new_sentence.block:
                        sentence_copy = deepcopy(new_sentence)
                        sentence_copy.block = []

                        sentence_result = sentence_copy.result
                        sentence_copy.conditions.append(
                            Condition(sentence_result,
                                      sentence_block.pattern)
                        )
                        sentence_copy.result = sentence_block.result

                        sentence_result_.append(sentence_copy)
                else:
                    sentence_result_.append(new_sentence)
            function.sentences.extend(sentence_result_)
```

В случае, приведем пример программы с блоком:

Листинг 20. Пример программы с блоком на РЕФАЛ-5

```
HexDigit {
  s.Digit, <Type s.Digit>:
    {
      'D0' s.Digit = <Numb s.Digit>;
      'Lu' s.Digit = <DoHexDigit 'ABCDEF' 10 s.Digit>;
      'Ll' s.Digit = <DoHexDigit 'abcdef' 10 s.Digit>;
      s.Other = /* пусто */;
    };
}
DoHexDigit {
  s.Digit e.Samples s.Val s.Digit = s.Val;

  s.OtherDigit e.Samples s.Val s.Digit =
    <DoHexDigit e.Samples <Add s.Val 1> s.Digit>;

  /* нет образцов */ s.BadVal s.Digit = /* пусто */;
}
```

После применения преобразования (block_to_condition) функция примет вид:

Листинг 21. Пример результата преобразования программы с блоками в условия на РЕФАЛ-5

```
HexDigit {
  s.Digit, <Type s.Digit>: 'D0' s.Digit = <Numb s.Digit>;
  s.Digit, <Type s.Digit>: 'Lu' s.Digit = <DoHexDigit 'ABCDEF' 10 s.Digit>;
  s.Digit, <Type s.Digit>: 'Ll' s.Digit = <DoHexDigit 'abcdef' 10 s.Digit>;
  s.Digit, <Type s.Digit>: s.Other = /* пусто */;
}

DoHexDigit {
  s.Digit e.Samples s.Val s.Digit = s.Val;

  s.OtherDigit e.Samples s.Val s.Digit =
    <DoHexDigit e.Samples <Add s.Val 1> s.Digit>;

  /* нет образцов */ s.BadVal s.Digit = /* пусто */;
}
```

Предположим, $in(HexDigit) = \bot, out(HexDigit) = \bot$, $in(DoHexDigit) = \bot, out(DoHexDigit) = \bot$.

Первая итерация очевидна.

$$in(HexDigit) = Gen(\bot) = \bot$$
$$out(HexDigit) = Gen(\bot) = \bot$$
$$in(DoHexDigit) = Gen(s.Digit\ e.Samples\ s.Val\ s.Digit, s.BadVal\ s.Digit) = s\ e\ s$$
$$out(DoHexDigit) = Gen(s.Val, \varepsilon) = s$$

На второй итерации, уравнение $out(Type) : s.Other$ не разрешимо, так как выходной формат функции Type имеет вид $Type\ e = s\ s\ e$. Поэтому, наблюдается ошибка в функции DoHexDigit.

# 3 Тестирование

## 3.1 Руководство пользователя

Системные требования.

Для запуска программы необходимо следующее ПО:

- GNU/Linux, MacOS, Windows
- Python 3.6 и выше
- Библиотеку pytest (для запуска тестов)

Установка и запуск

Данную программу можно установить двумя способами:

1. Из репозитория с Github [13]
2. Используя официальный установщик пакетов (pip)

Рассмотрим более подробнее каждый из способов.

Github – самый популярный веб-сервис для хостинга IT-проектов, основанный на системе контроля версий. Выполним команду для загрузки проекта на наш компьютер (листинг №22):

Листинг 22. Клонирование репозитория

```
MacBook-Pro-Georgij:~ geoiva$ git clone -b dev
https://github.com/runnerpeople/Refal5.git
```

Далее необходимо запустить «установщик» (листинг №23):

Листинг 23. Запуск «установщика»

```
MacBook-Pro-Georgij:~ geoiva$ python3 setup.py install
```

В Python имеется очень удобный инструмент – система скриптов установки. Документация на данный момент слишком размыта, так как имеется большой набор утилит (setuptools, distutils, distribute), выполняющие одни и те же функции. Файл дистрибьюции (setup.py) описывает метаданные и python кода проекта. Хорошим тоном является вынесение данного файла в корневую директорию проекта, который также будет мета-данные пакета и вспомогательные файлы (лицензию, документацию и т.д.), а сам модуль

программы будет являться поддиректорией проекта. Также хорошим тоном будет являться то, что содержится описание проекта в файле README.txt.

Листинг 24. Файл дистрибьюции (setup.py)

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from setuptools import find_packages, setup, Command

...

setup(
    name=NAME,
    version=VERSION,
    description=DESCRIPTION,
    long_description=long_description,
    long_description_content_type='text/markdown',
    author=AUTHOR,
    author_email=EMAIL,
    python_requires=REQUIRES_PYTHON,
    url=URL,
    packages=find_packages(),
    include_package_data=True,

    install_requires=REQUIRED,
    extras_require=EXTRAS,
    license='MIT',
    ...
    cmdclass={
        'upload': UploadCommand,
        'test': TestCommand
    },
    entry_points={
        'console_scripts': ['refalcheck=refal.refalcheck:main'],
    },
    tests_require=["pytest"],
    test_suite='tests'
)
```

Файл setup.py (листинг №24) обязательно содержит важную функцию *setup*, которая и выполняет установку пакета. Параметрами этой функции могут быть название модуля, версии программы, описание, данные об авторе, URL проекта, необходимые модули, необходимая версия Python, лицензия, а также собственные команды для setup.py (в листинге №24 описаны дополнительные команды *upload*, которая делает выгрузку модуля в PyPi, и *test*, которая запуск тесты) и собственные «точки вызова» программы (создание скриптов на уровне /usr/local/bin) и т.д. Существует множество систем наименования версий программы в python, но обычно рекомендуется использовать PEP386. Чаще всего

обозначение версии состоит из номера мажорного, минорного релизов, разделенных точками. В последней части версии разрешается использовать буквы латинского алфавита (для бета-версии используется “b”). Также для установки модуля, содержащего не только python-файлы, необходимо создать файл MANIFEST.in и указать в нём все файлы, которые будут необходимы при использовании нашей программы.

Если же данный способ является несколько сложным, то можно использовать установщик пакетов pip. Данный проект был опубликован на сайте PyPi.org, при помощи файла дистрибьюции (setup.py). Для скачивания проекта в этом случае понадобится (листинг №25):

Листинг 25. Скачивание модуля из PyPi

```
MacBook-Pro-Georgij:~ geoiva$ pip3 install refalchecker
```

В любом случае установки мы получим консольное приложение (refalcheck). На вход подается имя файла *.ref, который содержит программу, написанную на РЕФАЛ-5, и файлы *.type, содержащие форматы используемых функций. Например (листинг 26):

Листинг 26. Пример запуск программы

```
MacBook-Pro-Georgij:~ geoiva$ refalcheck R05-Parser.ref LibraryEx.type R05-
Lexer.type
```

В файле constants.py используются константы, конфигурируемые при запуске верификатора (например, для вывода отладочной информации). Есть разные режимы работы верификатора. Для подробной отладочной информации (списки токенов и др.), необходимо указать в файле constants.py значение переменной *DEBUG_MODE* в True.

Для проверки работоспособности верификатора необходимо можно запустить пакетное тестирование (листинг №27):

Листинг 27. Пакетное тестирование.

```
MacBook-Pro-Georgij:~ geoiva$ python3 setup.py test
```

При успешной работе вывод должен быть таким (листинг №28):

Листинг 28. Вывод работы пакетного тестирования.

```
MacBook-Pro-Georgij:~ geoiva$
============================================================================
========== test session starts
============================================================================
==========

platform darwin -- Python 3.7.0, pytest-4.5.0, py-1.8.0, pluggy-0.11.0
rootdir: /Users/geoiva/Desktop/Учеба/Учеба (8 сем)/Диплом
collected 29 item

        tests/refal_test.py .
100%

============================================================================
======= 29 passed in 14.32 seconds
============================================================================
========
```

### 3.2 Тестирование работы программы

Рассмотрим пример работы программы. Например, дан файл (листинг №29) и файл спецификации типов (листинг №30):

Листинг №29. Пример программы на РЕФАЛ-5

```
*$FROM LibraryEx
$EXTERN MapAccum;

$ENTRY generator_GenSentence {
  ((e.Pattern) (e.Result)) =
    ()
    ('  do {')
    <SkipIndentAccum
      <MapAccum
        generator_GenCommand
        ('    ' /* отступ */)
        <CompileSentence (e.Pattern) (e.Result)>
      >
    >
    ('  } while (0);');

  ReturnRecognitionImpossible =
    ()
    ('  r05_recognition_impossible();');
}

SkipIndentAccum {
  ('    ') e.Generated = e.Generated;
}

CompileSentence {
  (e.Pattern) (e.Result) = e.Pattern e.Result;
}
```

Листинг №30. Пример файла спецификации типов

```
SaveFile (e.FileName) e.Lines = ;
MapAccum t.Func t.Acc e.Terms = t.Acc e.Res;
Map t.Func e.Terms = e.Res;
```

То после работы верификатора вывод будет такой, как на листинге №31.

Листинг №31. Вывод верификатора

```
/usr/local/bin/python3.7 "/Users/geoiva/Desktop/Учеба/Учеба (8
сем)/Диплом/refal/refalcheck.py" "/Users/geoiva/Desktop/Учеба/Учеба (8
сем)/Диплом/tests/files/test18.ref" "/Users/geoiva/Desktop/Учеба/Учеба (8
сем)/Диплом/tests/files/test18.type"
Ok. Program satisfy grammar
Ok. Program-Type satisfy grammar
Ok. Program-Type satisfy grammar
/* result program refalcheck */

generator_GenSentence t = () ('  ' s s s s e) e
*SkipIndentAccum ('    ') e = e
*CompileSentence (e) (e) = e
```

Рассмотрим пример программы, где функция имеет ошибочный формат (например, суперкомпилятор MSCP-A).

Листинг №32. Фрагмент программы MSCP-A.

```
$ENTRY MytoBool {
    '+' = 'T';
    0 = 'F';
    '-' = 'F';
}
$ENTRY MyIfNotLess {
    Inf s.2 = 'T';
    s.1 s.1 = 'T';
    s.1 Inf = 'F';
    s.1 s.2, <Type s.1> : 'N' e.other, <Type s.2> : 'N' e.other2
        = <MytoBool <Compare s.1 s.2>>;
    '-' s.1 s.2 = 'F';
    '-' s.1 '-' s.1 = 'T';
    '-' s.1 '-' s.2 = <MytoBool <Compare '-' s.1 '-' s.2>>;
    t.1 t.2 = 'F';
}
DEDecreaseDim1 {
    s.XVal (AreEqual ((s.X t.X) e.Vars) (s.C Const)),
    <MyIfNotLess s.C s.X> : 'T'
        =
        <DEDecreaseDim1
            <MyIfNotLess
                <Sub s.C s.X>
                s.X
            >
            <Add s.XVal 1>
            (AreEqual ((s.X t.X) e.Vars) (<Sub s.C s.X> Const))
        >;
    s.XVal (AreEqual ((s.X t.X) e.Vars) (s.C Const)) = ;
}
```

То после работы верификатора вывод будет такой, как на листинге №33.

Листинг №33. Вывод верификатора

```
/usr/local/bin/python3.7 "/Users/geoiva/Desktop/Учеба/Учеба (8
сем)/Диплом/refal/refalcheck.py" "/Users/geoiva/Desktop/Учеба/Учеба (8
сем)/Диплом/tests/files/test44.ref"
Ok. Program satisfy grammar
Ok. Program-Type satisfy grammar
/* result program refalcheck */
(34,5)-(44,10) Function DEDecreaseDim1, sentence 0, there isn't solution for
equation => s s e (AreEqual ((s t) e) (s e Const)) : s (AreEqual ((s t) e)
(s Const))
```

В выводе представлена ошибка решения уравнения, с указанием позиции предложения в функции и индекс предложения в функции и само уравнение, в котором применились сужения и присваивания.

# ЗАКЛЮЧЕНИЕ

В рамках ВКР:

- Проведен обзор предметной области, а именно изучение языка РЕФАЛ и его диалект РЕФАЛ-5;
- Разработан и реализован алгоритм вывода типа функций, а также реализована проверка корректности вызовов функций в программе;
- Проведено тестирование на более, чем 50 тестах (из которых 10 тестов являются реальными программами, например, суперкомпилятор RMCC, MSCP-A, компилятор РЕФАЛа-05).

В ходе написания работы был разработан верификатор, который был реализован на языке Python3. Данный верификатор оформлен в виде python-библиотеки и был выгружен в менеджер пакетов Python PyPI. Требования, предъявленные к разрабатываемому верификатору, были выполнены.

При этом в дальнейшем планируется модификация алгоритма обобщения (использование ГСО), а также оптимизация работы верификатора.

Данный верификатор можно использовать в реальных программах.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. Ключников И. Г. Выявление и доказательство свойств функциональных программ методами суперкомпиляции. – Москва, 2010.
2. Романенко С. А. Машинно-независимый компилятор с языка рекурсивных функций [диссертация]. – 1978.
3. В.Ф. Турчин. Алгоритмический язык рекурсивных функций (РЕФАЛ). – ИПМ АН СССР, 1968.
4. Документация языка РЕФАЛ-5$\lambda$ [электронный ресурс]. – Режим доступа: https://github.com/bmstu-iu9/refal-5-lambda
5. В.Ф. Турчин. Эквивалентные преобразования рекурсивных функций, описанных на языке РЕФАЛ. – Киев-Алушта, 1972.
6. S.A. Romanenko. Arity Raiser and its Use in Program Specialization // 3rd European Symposium on Programming, London, May 15-18, 1990.
7. А.П. Немытых. Суперкомпилятор SCP4: Общая структура. – Издательство ЛКИ, 2007.
8. С.М.Абрамов, С.А.Романенко. Представление объектных выражений массивами при реализации языка Рефал. – М.:ИПМ им.М.В.Келдыша АН СССР, 1988, препринт N 186.
9. Скоробогатов С. Ю. Реализация Рефал-компилятора. Представление данных и язык сборки [рукопись]. – 2008.
10. Р. Гурин, С. Романенко. Язык программирования Рефал Плюс. Курс лекций. Учебное пособие для студентов университета города Переславля. – Университет города Переславля им. А. К. Айламазяна, 2006.
11. В.Ф. Турчин. Эквивалентные преобразования программ на РЕФАЛе // Труды ЦНИПИАСС Автоматизированная система управления строительством, выпуск 6, 1974.
12. Романенко С. А. Генератор компиляторов, порождённый самоприменением специализатора, может иметь ясную и естественную структуру. – М.:ИПМ им. М.В.Келдыша АН СССР, 1987, препринт N 26.

13. Исходный код приложения refalchecker. URL: https://github.com/runnerpeople/Refal5

## Приложение А

Проверим работоспособность верификатора на компиляторе RMCC.

Листинг А1. Компилятор RMCC

```
/*****************************************************************************
 Real Men Compiler Constructor.

 Copyright (c) 2006 Sergei Yu. Skorobogatov. 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.

 THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "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 COPYRIGHT OWNER OR CONTRIBUTORS
 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.

 *****************************************************************************
 File:
     rmcc.ref

 Description:
     The compiler constructor written specially for Refal compiler.

 Author:
     Sergei Skorobogatov (Sergei.Skorobogatov@supercompilers.com)
 *****************************************************************************/

/*
 * Main part.
 */

$ENTRY GO {
    = <Prout 'Real Men Compiler Constructor'>
      <Prout '    source-file : ' <Arg 1>>
      <Prout '    token-file  : ' <Arg 2>>
      <Prout '    dest-file   : ' <Arg 3>>
      <Prout '    report-file : ' <Arg 4>>
      <Prout '----------------/'>
      <Open 'r' 1 <Arg 1>>
      <Open 'r' 2 <Arg 2>>
      <Evaluate
          (<SortTok <ParseTok <LoadFile 2>>>)
          <Parse <LoadFile 1>>
          <Open 'w' 3 <Arg 3>>
          <Open 'w' 4 <Arg 4>>
      >;
```

```
    }

Evaluate {
    t.Tokens e.Rules t.As t.Ts t.Ns
        , t.Tokens: ((s.100 t.MinTs) e.101)
        , t.Ns: (t.StartNs e.103)
        , <EvalFirst (e.Rules) t.Ns>: (e.First)
        , <Lenw e.First>: s.NsCount e.104
        , <EvalFollow (e.Rules) (e.First) t.Ns>: (e.Follow)
        , <EvalParse (e.Rules) (e.First) (e.Follow)>: (e.Cells)
        , <BuildMatrix (e.Cells) t.Tokens>: t.ParseMatrix
        , <BuildProductions e.Rules>: s.ProdSize (e.Prod) s.RowsSize (e.Rows)
        = <SaveFile 4
            '**** TOKENS' CAR_RET t.Tokens CAR_RET CAR_RET
            '**** FIRST' CAR_RET <DbgFormatSet e.First> CAR_RET
            '**** FOLLOW' CAR_RET <DbgFormatSet e.Follow> CAR_RET
            '**** PARSE TABLE' CAR_RET <DbgFormatParse e.Cells> CAR_RET
          >
          <SaveFile 3
            CAR_RET

            '/*' CAR_RET
            ' * Terminal symbols.' CAR_RET
            ' */' CAR_RET CAR_RET
            <FormatTs t.Tokens> CAR_RET

            '/*' CAR_RET
            ' * Terminal symbols strings.' CAR_RET
            ' */' CAR_RET
            'static char *ts_str[] =' CAR_RET
            '{' CAR_RET
            <FormatTsStrings t.Tokens> CAR_RET
            '};' CAR_RET CAR_RET

            '/*' CAR_RET
            ' * Nonterminal symbols.' CAR_RET
            ' */' CAR_RET CAR_RET
            <FormatNs 1000 t.Ns> CAR_RET

            '/*' CAR_RET
            ' * Nonterminal symbols strings.' CAR_RET
            ' */' CAR_RET
            'static char *ns_str[] =' CAR_RET
            '{' CAR_RET
            <FormatNsStrings t.Ns> CAR_RET
            '};' CAR_RET CAR_RET

            '/*' CAR_RET
            ' * Semantic actions symbols.' CAR_RET
            ' */' CAR_RET CAR_RET
            <FormatAs 1 t.As> CAR_RET

            '/*' CAR_RET
            ' * Terminal symbol with min. code.' CAR_RET
            ' */' CAR_RET
            '#define MIN_TERMINAL ' <FormatTsName t.MinTs> CAR_RET CAR_RET

            '/*' CAR_RET
            ' * Start nonterminal of the grammar.' CAR_RET
            ' */' CAR_RET
            '#define START_NONTERMINAL ' <FormatNsName t.StartNs> CAR_RET
CAR_RET

            '/*' CAR_RET
```

```
                ' * LL(1) parse table.' CAR_RET
                ' */' CAR_RET
                <FormatMatrix ('p7_table') t.ParseMatrix ('-' 1)> CAR_RET

*               '/*' CAR_RET
*               ' * FIRST sets table.' CAR_RET
*               ' */' CAR_RET
*               <FormatMatrix ('p7_first') <Set2Matrix (e.First) t.Tokens> (0)>
CAR_RET
*
*               '/*' CAR_RET
*               ' * FOLLOW sets table.' CAR_RET
*               ' */' CAR_RET
*               <FormatMatrix ('p7_follow') <Set2Matrix (e.Follow) t.Tokens> (0)>
CAR_RET

                '/*' CAR_RET
                ' * Synchronization sets table.' CAR_RET
                ' */' CAR_RET
                <FormatMatrix
                    ('p7_sync')
                    <BuildMatrix_FirstFollow (e.First) (e.Follow) t.Tokens>
                    (0)
                > CAR_RET

                '/*' CAR_RET
                ' * Nonterminals that can produce empty sequences.' CAR_RET
                ' */' CAR_RET
                'static short p7_eps[' <Symb s.NsCount> '] =' CAR_RET
                '{' CAR_RET
                <FormatEps <BuildEps e.First>>
                '};' CAR_RET CAR_RET

                '/*' CAR_RET
                ' * Production table.' CAR_RET
                ' */' CAR_RET
                'static short p7_prod[' <Symb s.ProdSize> '] =' CAR_RET
                '{' CAR_RET
                <FormatProductionTable e.Prod>
                '};' CAR_RET CAR_RET

                '/*' CAR_RET
                ' * Entries to the production table.' CAR_RET
                ' */' CAR_RET
                'static short p7_rows[' <Symb s.RowsSize> '] =' CAR_RET
                '{' CAR_RET
                <FormatRows 1 e.Rows>
                '};' CAR_RET
            >;
    }

SortTok {
    e.L (s.1 t.X) e.M (s.2 t.Y) e.R
        , <Sub s.2 s.1>: '-' s.3
        = <SortTok e.L (s.2 t.Y) e.M (s.1 t.X) e.R>;

    e.1 = e.1;
    }

FormatAs {
    s.C (t.X e.Xs)
        = <LeftJustify 33 '#define ' <FormatAsName t.X>>
          '(short)(-' <Symb s.C> ')' CAR_RET
          <FormatAs <Add (s.C) 1> (e.Xs)>;
```

```
    s.C () = ;
    }

FormatTs {
    ((s.N (TS e.Name)) e.Xs)
        = <LeftJustify 33 '#define ' <FormatTsName (TS e.Name)>>
          '(short)L_' e.Name CAR_RET
          <FormatTs (e.Xs)>;

    () = ;
    }

FormatTsStrings {
    ((s.N (TS e.Name)) e.Xs)
        = '    \"' <FormatTsName (TS e.Name)> '\"'
          <FormatTsStrings2 (e.Xs)>;

    () = ;
    }

FormatTsStrings2 {
    () = ;
    (e.Xs) = ',' CAR_RET <FormatTsStrings (e.Xs)>;
    }

FormatNs {
    s.C (t.X e.Xs)
        = <LeftJustify 33 '#define ' <FormatNsName t.X>>
          '(short)' <Symb s.C> CAR_RET
          <FormatNs <Add (s.C) 1> (e.Xs)>;

    s.C () = ;
    }

FormatNsStrings {
    (t.X e.Xs)
        = '    \"' <FormatNsName t.X> '\"'
          <FormatNsStrings2 (e.Xs)>;

    () = ;
    }

FormatNsStrings2 {
    () = ;
    (e.Xs) = ',' CAR_RET <FormatNsStrings (e.Xs)>;
    }

FormatAsName {
    (AS e.Name) = 'AS_' <Upper e.Name>;
    }

FormatTsName {
    (TS e.Name) = 'TS_' <Upper e.Name>;
    }

FormatNsName {
    (NS e.Name) = 'NS_' <FormatNsName2 <Upper e.Name>>;
    }

FormatNsName2 {
    '-' e.1 = '_' <FormatNsName2 e.1>;
    '\'' e.1 = '_' <FormatNsName2 e.1>;
    s.X e.1 = s.X <FormatNsName2 e.1>;
```

```
    = ;
    }

FormatRows {
    1 e.Rows = '    ' <FormatRows2 1 e.Rows>;
    17 e.Rows = CAR_RET <FormatRows 1 e.Rows>;
    s.C e.Rows = <FormatRows2 s.C e.Rows>;
    }

FormatRows2 {
    s.C s.X e.Xs
        = <LeftJustify 5 <Symb s.X> <FormatRows3 e.Xs>>
          <FormatRows <Add (s.C) 1> e.Xs>;

    1 = ;

    s.C = CAR_RET;
    }

FormatRows3 {
    = ;
    e.1 = ',';
    }

BuildEps {
    (t.X EPS e.1) e.2 = 1 <BuildEps e.2>;
    t.1 e.2 = 0 <BuildEps e.2>;
    = ;
    }

FormatEps {
    = ;
    e.1 = <FormatEps2 1 e.1>;
    }

FormatEps2 {
    1 e.1 = '    ' <FormatEps3 1 e.1>;
    17 e.1 = CAR_RET <FormatEps2 1 e.1>;
    s.N e.1 = <FormatEps3 s.N e.1>;
    }

FormatEps3 {
    s.N s.X = <Symb s.X> CAR_RET;
    s.N s.X e.1 = <Symb s.X> ', ' <FormatEps2 <Add (s.N) 1> e.1>;
    }

FormatProductionTable {
    ((e.Comment) (e.Code)) e.Prod
        = '    /* ' <FormatComment e.Comment> '*/' CAR_RET
          '    ' <FormatCode e.Code>
          <FormatProductionTable2 e.Prod>;

    = ;
    }

FormatProductionTable2 {
    = CAR_RET;
    e.Prod = ',' CAR_RET CAR_RET <FormatProductionTable e.Prod>;
    }

FormatComment {
    s.Num (e.Name) e.Ys
        = <Symb s.Num> '. ' e.Name ' ::= ' <FormatComment2 e.Ys>;
    }
```

```
FormatComment2 {
    (e.Name) e.Ys = e.Name ' ' <FormatComment2 e.Ys>;
    = ;
    }

FormatCode {
    s.Size e.Ys = <Symb s.Size> <FormatCode2 e.Ys>;
    }

FormatCode2 {
    (e.Name) e.Ys = ', ' e.Name <FormatCode2 e.Ys>;
    = ;
    }

LoadFile {
    s.1 = <LoadFile2 s.1 <Get s.1>>;
    }

LoadFile2 {
    s.1 0 = ;
    s.1 e.2 0 = e.2;
    s.1 e.2 = e.2 CAR_RET <LoadFile s.1>;
    }

SaveFile {
    s.1 = ;
    s.1 e.2 CAR_RET e.3 = <Put s.1 e.2> <SaveFile s.1 e.3>;
    s.1 e.2 = <Put s.1 e.2>;
    }

/*
 * Grammar parser.
 */
Parse {
    e.1 = <Parse2 () () ()
              <NormalizeBeginning
                  <NormalizeEnding
                      <RemoveExtraSpace
                          <ProcessLineWraps
                              <RemoveComments e.1>
                          >
                      >
                  >
              >
          >;
    }

RemoveComments {
    e.1 '/*' e.2 '*/' e.3    = e.1 <RemoveComments e.3>;
    e.1                      = e.1;
    }

ProcessLineWraps {
    e.1 '\\' e.2 CAR_RET e.3 = e.1 <ProcessLineWraps e.3>;
    e.1                      = e.1;
    }

RemoveExtraSpace {
    '  ' e.1                 = <RemoveExtraSpace ' ' e.1>;
    ' ' CAR_RET e.1          = <RemoveExtraSpace CAR_RET e.1>;
    CAR_RET CAR_RET e.1      = <RemoveExtraSpace CAR_RET e.1>;
    s.X e.1                  = s.X <RemoveExtraSpace e.1>;
                             = ;
```

```
    }

NormalizeBeginning {
    ' ' e.1                 = <NormalizeBeginning e.1>;
    CAR_RET e.1             = <NormalizeBeginning e.1>;
    e.1                     = e.1;
    }

NormalizeEnding {
    e.1 ' '                 = <NormalizeEnding e.1>;
    e.1 CAR_RET             = <NormalizeEnding e.1>;
    e.1                     = e.1 CAR_RET;
    }

Parse2 {
    (e.As) (e.Ts) (e.Ns) e.Name ' ::=' e.2 CAR_RET e.3
        , <Parse3 (e.As) (e.Ts) (e.Name) (e.3)>: t.As_ t.Ts_ e.Rules (e.4)
        = e.Rules <Parse2 t.As_ t.Ts_ (e.Ns (NS e.Name)) e.4>;
    (e.As) (e.Ts) (e.Ns) = (e.As) (e.Ts) (e.Ns);
    }

Parse3 {
    t.As t.Ts (e.Name) e.Rules (' ' e.1 CAR_RET e.2)
        , <Parse4 (0) e.1>: s.N e.R
        , <Update t.As t.Ts e.R>: t.As_ t.Ts_
        = <Parse3 t.As_ t.Ts_ (e.Name) e.Rules ((NS e.Name) s.N e.R) (e.2)>;

    t.As t.Ts (e.Name) e.Rules (e.2) = t.As t.Ts e.Rules (e.2);
    }

Parse4 {
    (0) '_epsilon_' = 0;
    (s.N e.R) e.X ' ' e.Y = <Parse4 (<Add s.N 1> e.R <RecognizeSym e.X>) e.Y>;
    (s.N e.R) e.X = <Add s.N 1> e.R <RecognizeSym e.X>;
    }

RecognizeSym {
    '_action_' e.1 = (AS e.1);
    'L_' e.1 = (TS e.1);
    e.1 = (NS e.1);
    }

Update {
    (e.As) (e.Ts) t.S e.1
        , t.S: {
              (AS e.2), e.As: e.L t.S e.R = <Update (e.As) (e.Ts) e.1>;
              (AS e.2) = <Update (e.As t.S) (e.Ts) e.1>;
              (TS e.2), e.Ts: e.L t.S e.R = <Update (e.As) (e.Ts) e.1>;
              (TS e.2) = <Update (e.As) (e.Ts t.S) e.1>;
              (NS e.2) = <Update (e.As) (e.Ts) e.1>;
              };

    (e.As) (e.Ts) = (e.As) (e.Ts);
    }

ParseTok {
    e.1 = <ParseTok2
              <NormalizeBeginning
                  <NormalizeEnding
                      <RemoveExtraSpace e.1>
                  >
              >
          >;
    }
```

```
ParseTok2 {
    e.Name ' ' e.Num CAR_RET e.1
        = (<Numb e.Num> (TS <FixName e.Name>)) <ParseTok2 e.1>;

    = ;
    }

FixName {
    'END_OF_SLK_INPUT_' = 'END_OF_INPUT';
    'L_' e.1 = e.1;
    e.1 = e.1;
    }

/*
 * Functions that are common for FIRST and FOLLOW sets
 */

Merge {
    (EPS e.1) EPS e.2 = <Merge (EPS e.1) e.2>;
    (e.1) EPS e.2 = <Merge (EPS e.1) e.2>;

    (e.1) = e.1;

    (e.1) t.X e.2
        , e.1: {
              e.L t.X e.R = <Merge (e.1) e.2>;
              e.3 = <Merge (e.1 t.X) e.2>;
              };
    }

DbgFormatSet {
    ((NS e.X) e.Xs) e.Fs
        = e.X ': ' <DbgFormatSet2 e.Xs> CAR_RET <DbgFormatSet e.Fs>;
    = ;
    }

DbgFormatSet2 {
    EPS e.1 = '_epsilon_ ' <DbgFormatSet2 e.1>;
    (TS e.Y) e.1 = 'L_' e.Y ' ' <DbgFormatSet2 e.1>;
    = ;
    }

Set2Matrix {
    (e.Set) t.Tokens = <BuildMatrix (<Set2Matrix2 1 e.Set>) t.Tokens>;
    };

BuildMatrix_FirstFollow {
    (e.First) (e.Follow) t.Tokens
        , <Set2Matrix2 1 e.First>: e.FirstM
        , <Set2Matrix2 2 e.Follow>: e.FollowM
        , <SortTable e.FirstM e.FollowM>: e.Table
        = <BuildMatrix (e.Table) t.Tokens>;
    }

SortTable {
    e.1 (t.X t.Y s.A) e.2 (t.X t.Y s.B) e.3
        = <SortTable e.1 (t.X t.Y 3) e.2 e.3>;

    e.1 (t.X t.Y1 s.A) e.2 (t.X t.Y2 s.B) e.3
        = e.1 (t.X t.Y1 s.A) <SortTable (t.X t.Y2 s.B) e.2 e.3>;

    e.1 = e.1;
    }
```

```
Set2Matrix2 {
    s.Id (t.X e.Xs) e.Fs
        = <Set2Matrix3 s.Id t.X e.Xs> <Set2Matrix2 s.Id e.Fs>;
    s.Id = ;
    }

Set2Matrix3 {
    s.Id t.X EPS e.1 = <Set2Matrix3 s.Id t.X e.1>;
    s.Id t.X t.Y e.1 = (t.X t.Y s.Id) <Set2Matrix3 s.Id t.X e.1>;
    s.Id t.X = ;
    }

/*
 * FIRST set evaluator.
 */

EvalFirst {
    t.Rules t.Ns = <EvalFirst2 t.Rules () t.Ns>;
    }

EvalFirst2 {
    (e.Rules) (e.1) (t.X e.Ns)
        , e.Rules: {
              e.L (t.X 0) e.R = <EvalFirst2 (e.L e.R) (e.1 (t.X EPS)) (e.Ns)>;
              e.2 = <EvalFirst2 (e.Rules) (e.1 (t.X)) (e.Ns)>;
              };

    (e.Rules) (e.1) () = <EvalFirst_Loop (e.Rules) (e.1)>;
    }

EvalFirst_Loop {
    t.Rules t.First
        , <EvalFirst_Process t.Rules t.First>: {
              t.First = t.First;
              e.1 = <EvalFirst_Loop t.Rules e.1>;
              };
    }

EvalFirst_Process {
    ((t.X s.N e.Ys) e.Rules) (e.First)
        , e.First: e.L (t.X e.Fx) e.R
        , <Merge (e.Fx) <FIRST (e.Ys) (e.First)>>: e.Fx_
        = <EvalFirst_Process (e.Rules) (e.L (t.X e.Fx_) e.R)>;

    () t.First = t.First;
    }

FIRST {
    () (e.First) = EPS;

    (t.X e.Xs) (e.First)
        , t.X: {
              (AS e.1) = <FIRST (e.Xs) (e.First)>;
              (TS e.1) = t.X;
              (NS e.1)
                  , e.First: e.L (t.X e.Fx) e.R
                  , e.Fx: {
                        EPS e.2 = <Merge (e.2) <FIRST (e.Xs) (e.First)>>;
                        e.2 = e.2;
                        }
              };
    }
```

```
/*
 * FOLLOW set evaluator.
 */

EvalFollow {
    t.Rules t.First (t.X e.Ns)
        = <EvalFollow2 t.Rules t.First ((t.X (TS 'END_OF_INPUT'))) (e.Ns)>;

    t.Rules t.First () = ();
    }

EvalFollow2 {
    t.Rules t.First (e.1) (t.X e.Ns)
        = <EvalFollow2 t.Rules t.First (e.1 (t.X)) (e.Ns)>;

    t.Rules t.First (e.1) () = <EvalFollow_Loop t.Rules t.First (e.1)>;
    }

EvalFollow_Loop {
    t.Rules t.First t.Follow
        , <EvalFollow_Process t.Rules t.First t.Follow>: {
              t.Follow = t.Follow;
              e.1 = <EvalFollow_Loop t.Rules t.First e.1>;
              };
    }

EvalFollow_Process {
    ((t.X s.N e.Ys) e.Rules) t.First t.Follow
        , <EvalFollow_ProcessRule t.X (e.Ys) t.First t.Follow>: t.Follow_
        = <EvalFollow_Process (e.Rules) t.First t.Follow_>;

    () t.First t.Follow = t.Follow;
    }

EvalFollow_ProcessRule {
    t.X () t.First t.Follow = t.Follow;

    t.X (t.Y e.Ys) t.First t.Follow
        , t.Y: {
              (NS e.1)
                  , <FIRST (e.Ys) t.First>: {
                        EPS e.3
                            , <FOLLOW t.X t.Follow>: e.Fx
                            , t.Follow: (e.L (t.Y e.Fy) e.R)
                            , (e.L (t.Y <Merge (e.Fy) e.Fx>) e.R): t.Follow_
                            = <EvalFollow_ProcessRule2 t.X t.Y (e.Ys) (e.3)
t.First t.Follow_>;

                        e.3 = <EvalFollow_ProcessRule2 t.X t.Y (e.Ys) (e.3)
t.First t.Follow>;
                        };

              t.1 = <EvalFollow_ProcessRule t.X (e.Ys) t.First t.Follow>;
              };
    }

EvalFollow_ProcessRule2 {
    t.X t.Y (e.Ys) (e.FYs) t.First (e.L (t.Y e.Fy) e.R)
        , (e.L (t.Y <Merge (e.Fy) e.FYs>) e.R): t.Follow_
        = <EvalFollow_ProcessRule t.X (e.Ys) t.First t.Follow_>;
    }

FOLLOW {
    t.X (e.L (t.X e.Fx) e.R) = e.Fx;
```

```
    }

/*
 * Parse table evaluator.
 */

EvalParse {
    t.Rules t.First t.Follow = <EvalParse2 1 () t.Rules t.First t.Follow>;
    }

EvalParse2 {
    s.N (e.Cells) () t.First t.Follow = (e.Cells);

    s.N (e.Cells) ((t.A s.Size e.Alpha) e.Rules) t.First t.Follow
        , <FIRST (e.Alpha) t.First>: {
              EPS e.1
                  = <EvalParse2
                        <Add s.N 1>
                        (
                            e.Cells
                            <GenCells s.N t.A (e.1)>
                            <GenCells s.N t.A (<FOLLOW t.A t.Follow>)>
                        )
                        (e.Rules) t.First t.Follow
                    >;

              e.1 = <EvalParse2
                        <Add s.N 1>
                        (e.Cells <GenCells s.N t.A (e.1)>)
                        (e.Rules) t.First t.Follow
                    >;
              };
    }

GenCells {
    s.N t.A (t.X e.1) = (t.A t.X s.N) <GenCells s.N t.A (e.1)>;
    s.N t.A () = ;
    }

DbgFormatParse {
    t.Cell e.1
        , t.Cell: (t.X e.2)
        , t.X: (NS e.Name)
        = e.Name ':' <DbgFormatParse2 t.X (t.Cell e.1)>;

    = ;
    }

DbgFormatParse2 {
    t.X ((t.X (TS e.A) s.P) e.1)
        = '   L_' e.A ' ' s.P <DbgFormatParse2 t.X (e.1)>;

    t.X (e.1) = CAR_RET <DbgFormatParse e.1>;
    }

/*
 * Work with matrixes.
 */
BuildMatrix {
    (e.Table) t.Tokens
        , t.Tokens: ((s.1 t.X) e.M (s.2 t.Y))
        , <Group e.Table>: e.Groups
        , <Lenw e.Groups>: s.Height e.3
        , <Add (<Sub (s.2) s.1>) 1>: s.Width
```

```
        = (s.Height s.Width <BuildMatrix2 s.1 s.2 t.Tokens (e.Groups)>);
    }

Group {
    t.Cell e.1
        , t.Cell: (t.X e.2)
        , <Group2 t.X (t.Cell e.1)>: e.Grp (e.3)
        = (e.Grp) <Group e.3>;

    = ;
    }

Group2 {
    t.X ((t.X t.A s.P) e.1)
        = (t.A s.P) <Group2 t.X (e.1)>;

    t.X (e.1) = (e.1);
    }

BuildMatrix2 {
    s.1 s.2 t.Tokens (t.G e.Groups)
        = <BuildRow () s.1 s.2 t.Tokens t.G>
          <BuildMatrix2 s.1 s.2 t.Tokens (e.Groups)>;

    s.1 s.2 t.Tokens () = ;
    }

BuildRow {
    (e.Row) s.i s.max t.Tokens t.G
        , <Sub (s.max) s.i>: s.1
        , <SearchGroup s.i t.Tokens t.G>: s.R
        = <BuildRow (e.Row s.R) <Add (s.i) 1> s.max t.Tokens t.G>;

    (e.Row) e.2 = (e.Row);
    }

SearchGroup {
    s.i (e.1 (s.i t.X) e.2) (e.3 (t.X s.P) e.4) = s.P;
    e.1 = 0;
    }

FormatMatrix {
    (e.Name) (s.Height s.Width e.Matrix) t.Fix
        = 'static short ' e.Name '[' <Symb s.Height> '][' <Symb s.Width> '] ='
CAR_RET
          '{' CAR_RET
          <FormatMatrix2 t.Fix e.Matrix>
          '};' CAR_RET
    }

FormatMatrix2 {
    t.Fix (e.Row) e.Matrix
        = '    { ' <FormatRow t.Fix e.Row> <FormatMatrix3 t.Fix e.Matrix>;
    }

FormatMatrix3 {
    t.Fix (e.Row) e.Matrix = ' },' CAR_RET <FormatMatrix2 t.Fix (e.Row)
e.Matrix>;
    t.Fix = ' }' CAR_RET;
    }

FormatRow {
    t.Fix s.1 e.2 = <RightJustify 2 <Symb <Add t.Fix s.1>>> <FormatRow2 t.Fix
e.2>;
```

```
    }

FormatRow2 {
    t.Fix = ;
    t.Fix e.1 = ',' <FormatRow t.Fix e.1>;
    }

/*
 * Productions.
 */

BuildProductions {
    e.Rules = <BuildProductions2 0 () 0 () e.Rules>;
    }

BuildProductions2 {
    s.Pos (e.Prod) s.RowsSize (e.Rows) (t.X s.Size e.Ys) e.Rules
        , <BuildProdRow s.RowsSize t.X s.Size e.Ys>: t.Row
        = <BuildProductions2
              <Add (<Add (s.Pos) s.Size>) 1>
              (e.Prod t.Row)
              <Add (s.RowsSize) 1>
              (e.Rows s.Pos)
              e.Rules
          >;

    s.Pos t.Prod s.RowsSize t.Rows = s.Pos t.Prod s.RowsSize t.Rows;
    }

BuildProdRow {
    s.Num (NS e.Name) s.Size e.Ys
        = <BuildProdRow2 (s.Num (e.Name)) (s.Size) e.Ys>;
    }

BuildProdRow2 {
    (e.Comment) (e.Code) t.Y e.Ys
        = <BuildProdRow2
              (e.Comment <ToComment t.Y>)
              (e.Code <ToCode t.Y>)
              e.Ys
          >;

    t.Comment t.Code = (t.Comment t.Code);
    }

ToComment {
    (TS e.Name) = ('L_' e.Name);
    (NS e.Name) = (e.Name);
    (AS e.Name) = ('_action_' e.Name);
    }

ToCode {
    (TS e.Name) = (<FormatTsName (TS e.Name)>);
    (NS e.Name) = (<FormatNsName (NS e.Name)>);
    (AS e.Name) = (<FormatAsName (AS e.Name)>);
    }

/*
 * Justifications.
 */

LeftJustify {
    0 e.S = e.S;
    s.Width s.X e.S = s.X <LeftJustify <Sub (s.Width) 1> e.S>;
```

```
    s.Width = ' ' <LeftJustify <Sub (s.Width) 1>>;
    }


RightJustify {
    0 e.S = e.S;
    s.Width e.S s.X = <RightJustify <Sub (s.Width) 1> e.S> s.X;
    s.Width = <RightJustify <Sub (s.Width) 1>> ' ';
    }
```

Результат работы верификатора можно наблюдать в листинге А2.

Листинг А2. Вывод верификатора.

```
/* result program refalcheck */

GO  = e
*Evaluate ((s t) e (s t)) e t t (t e) = e
*SortTok e = e
*FormatAs s (e) = e
*FormatTs (e) = e
*FormatTsStrings (e) = e
*FormatTsStrings2 (e) = e
*FormatNs s (e) = e
*FormatNsStrings (e) = e
*FormatNsStrings2 (e) = e
*FormatAsName (AS e) = 'AS_' e
*FormatTsName (TS e) = 'TS_' e
*FormatNsName (NS e) = 'NS_' e
*FormatNsName2 e = e
*FormatRows s e = e
*FormatRows2 s e = e
*FormatRows3 e = e
*BuildEps e = e
*FormatEps e = e
*FormatEps2 s e = e CAR_RET
*FormatEps3 s s e = e CAR_RET
*FormatProductionTable e = e
*FormatProductionTable2 e = s e
*FormatComment s (e) e = e
*FormatComment2 e = e
*FormatCode s e = e
*FormatCode2 e = e
*LoadFile s = e
*LoadFile2 s e = e
*SaveFile s e = e
*Parse e = e (e) (e) (e)
*RemoveComments e = e
*ProcessLineWraps e = e
*RemoveExtraSpace e = e
*NormalizeBeginning e = e
*NormalizeEnding e = e CAR_RET
*Parse2 (e) (e) (e) e = e (e) (e) (e)
*Parse3 (e) (e) (e) e (e) = (e) (e) e (e)
*Parse4 (s e) e = s e
*RecognizeSym e = (s e)
*Update (e) (e) = (e) (e)
*ParseTok e = e
*ParseTok2 e = e
*FixName e = e
*Merge (e) e = e
*DbgFormatSet e = e
*DbgFormatSet2 e = e
*Set2Matrix (e) t = (s s e)
```

```
*BuildMatrix_FirstFollow (e) (e) t = (s s e)
*SortTable e = e
*Set2Matrix2 s e = e
*Set2Matrix3 s t e = e
*EvalFirst t t = t
*EvalFirst2 (e) (e) (e) = t
*EvalFirst_Loop (e) t = t
*EvalFirst_Process (e) t = t
*FIRST () (e) = EPS
*EvalFollow t t (e) = t
*EvalFollow2 t t (e) (e) = t
*EvalFollow_Loop (e) t t = t
*EvalFollow_Process (e) t t = t
*EvalFollow_ProcessRule t (e) t t = t
*EvalFollow_ProcessRule2 t t (e) (e) t (e) = t
*FOLLOW t (e) = e
*EvalParse t t t = (e)
*EvalParse2 s (e) (e) (e) t = (e)
*GenCells s t (e) = e
*DbgFormatParse e = e
*DbgFormatParse2 t (e) = s e
*BuildMatrix (e) ((s t) e (s t)) = (s s e)
*Group e = e
*Group2 t (e) = (e) e
*BuildMatrix2 s s t (e) = e
*BuildRow (e) e = (e)
*SearchGroup e = s
*FormatMatrix (e) (s s e) t = 'static short ' e CAR_RET '};' CAR_RET
*FormatMatrix2 t (e) e = '    { ' e CAR_RET
*FormatMatrix3 t e = ' }' e CAR_RET
*FormatRow t s e = e
*FormatRow2 t e = e
*BuildProductions e = s t s t
*BuildProductions2 s t s t e = s t s t
*BuildProdRow s (NS e) s e = (t t)
*BuildProdRow2 t t e = (t t)
*ToComment (s e) = (e)
*ToCode (s e) = (s 'S_' e)
*LeftJustify s e = e
*RightJustify s e = e
```