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

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Выдача предупреждений для экранируемых предложений в компиляторе Рефала-5λ»

Студент ИУ9-82(Б) _____ А. Б. Барлука
(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 _____ С. Ю. Скоробогатов
(Подпись, дата) (И.О.Фамилия)

2020 г.

АННОТАЦИЯ

Объем данной дипломной работы составляет 51 страница. Для ее написания было использовано 13 источников. В работе содержится 1 рисунок, 3 таблицы, 17 листингов.

В дипломную работу входят 4 главы. В первой главе приведен обзор предметной области. Во второй главе изложена разработка алгоритма. В третьей главе описано тестирование реализованного алгоритма. В четвертой главе приведено руководство пользователя по использованию новых возможностей компилятора Рефала-5λ.

Объектом исследования данной ВКР являются алгоритмы распознавания экранирования предложений.

Цель работы — расширение возможностей компилятора языка Рефала-5λ путем реализации в нем поддержки выдачи предупреждений об экранировании предложений. Поставленная цель достигается за счет разработки механизма предупреждений в компиляторе Рефала-5λ и реализации алгоритма проверки экранирования.

В работе показано, что на данный момент не существует метода решения поставленной задачи в общем случае, однако, применяя различные методы, можно покрыть широкое подмножество проблемных случаев.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 2
1 Обзор предметной области 4
1.1 Обзор диалекта Рефал-5λ 4
1.2 Понятие экранирования предложений 7
1.3 Анализ требований к реализуемому алгоритму 9
2 Разработка 12
2.1 Реализация выдачи предупреждений пользователю 12
2.2 Реализация алгоритма проверки экранирования 16
2.2.1 Метод обобщенного сопоставления с образцом 17
2.2.2 Понятия теории языков образцов 18
2.2.3 Метод, основанный на теории языков образцов 21
3 Тестирование 25
3.1 Автотесты компилятора 25
3.2 Замеры производительности 30
3.2.1 Проверка кодовых баз нескольких крупных программ на Рефале-5(λ) 32
4 Руководство пользователя 36
ЗАКЛЮЧЕНИЕ 38
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 40
ПРИЛОЖЕНИЕ А 42

ВВЕДЕНИЕ

Предупреждения компилятора — это сообщения компилятора о потенциально ошибочных участках исходного кода программы, которые носят рекомендательный характер и не прерывают процесс компиляции. Как правило, подозрительные места в программе выявляются в ходе статического анализа кода.

С точки зрения языка программирования предупреждения не являются ошибками. При этом многие современные компиляторы позволяют отождествлять предупреждения с ошибками (например, GCC [1]), т.к. это позволяет сократить затрачиваемое на ожидание компиляции время при отладке программ.

Целью данной работы является выдача предупреждений для экранируемых предложений в компиляторе языка Рефала-5λ [2]. Экранируемые предложения — это предложения, которые никогда не выполняются и могут быть написаны лишь по ошибке программиста. Зачастую такие ошибки могут быть неочевидными. Распознавание случаев экранирования позволит значительно ускорить отладку исходного кода.

Актуальность работы подтверждается отсутствием на данный момент средства для решения задачи распознавания экранирования в компиляторе Рефала-5λ.

В рамках работы требуется решить следующие задачи:

- Проанализировать существующие в компиляторе решения по выдаче предупреждений и агрегировать их в единый фреймворк;
- Разработать алгоритм проверки экранирования образцов;
- Предусмотреть выдачу предупреждения об экранируемых предложениях.

Очевидно, что дополнительная фаза анализа исходной программы не может рассматриваться как какая-либо оптимизация времени выполнения — наоборот, лишь его увеличивает. Поэтому необходимо сделать все вычисления опциональными, чтобы не терять производительность в тех случаях, когда дополнительный анализ исходной программы не требуется.

1 Обзор предметной области

Рассмотрим особенности языка Рефала-5λ. Перечислим основные синтаксические конструкции и типы данных. Затем сформулируем задачу с учетом специфики языка.

РЕФАЛ (Рекурсивных Функций Алгоритмический язык) — функциональный язык программирования, в основе принципа работы которого лежат символьные вычисления. Язык Рефал-5λ — точное надмножество языка Рефала-5 [2]. Главное отличие заключается в поддержке функций высшего порядка. При этом исходный код, написанный на Рефале-5λ, на этапе рассахаривания преобразуется в более примитивные конструкции, что и позволяет языку сохранять статус точного надмножества.

1.1 Обзор диалекта Рефал-5λ

Отличительной особенностью языка является использование специальной структуры данных — *объектного выражения*, в реализации представляющее из себя двунаправленный список, а не однонаправленный, как во многих других функциональных языках. Данный факт предполагает эффективную обработку как левого, так и правого краев произвольной последовательности.

Объектные выражения — данные языка Рефал. Каждое объектное выражение либо пусто, либо содержит неограниченное число объектных термов.

Объектный терм представляется либо *символом*, т.е. атомным элементом данных (литерой, числом, словом, и т.д.), либо выражением, заключенным в

квадратные (объявление АТД — абстрактного типа данных) или круглые (структурные) скобки — *составным термом*.

Пример абстрактного типа данных `t.ErrorList` из исходного кода компилятора:

```
[ErrorList (e.FileName) t.Errors (e.Warnings)]
```

Квадратные скобки нужны для инкапсуляции данных; доступ к их содержимому возможен только в той единице трансляции, где они были объявлены.

Символы подразделяются на несколько категорий:

- Идентификатор (`True, False`, …);
- Печатный знак (`'ABC'`);
- Неотрицательное целое число;
- Указатель на функцию (`&Func`) ;
- Символ-замыкание, записываемый как вложенная функция в результатном выражении.

Наибольший интерес имеем к функциям и их телам, состоящим из набора предложений. Пример функции представлен на Листинге 1. В данном примере тело функции содержит 4 *предложения* — правила, описывающих вычисление функции при заданных ограничениях на аргумент. Предложение состоит из левой и правой частей. Левая часть содержит образец и, возможно, условия. При успешном сопоставлении с образцом выполняется правая часть (*результат*).

Как образцовые, так и результатные выражения могут содержать *переменные*. Всего существует 3 типа переменных:

- *s*-переменные — символ;

- t-переменные — терм;
- e-переменные — объектное выражение (0 и более термов).

```
Palindrom {

     s.1 e.2 s.1 = <Palindrom e.2>;

     s.1 = True;

     /* пусто */ = True;

     e.1 = False;

 }
```

Листинг 1 — Функция, проверяющая, является ли выражение палиндромом

Более того, в языке Рефал-5λ переменная может быть сокрыта новой переменной с символом ^ в конце имени. Такое сокрытие действительно только внутри текущей области видимости и не распространяется на объемлющие области видимости.

Образец — описание шаблона, как выражения с переменными. Для успешного сопоставления некоторого выражения с образцом необходимо существование подстановки переменных образца, переводящей его в данное выражение. В результате переменным из выражений-образцов присваиваются сопоставленные значения.

Существует особые выражения, составляющие подмножество в языке Рефал-5λ, называемые *L-выражениями*. L-выражения являются частными случаями объектных выражений и обладают следующими свойствами [3] [4]:

- Вхождение t- и e-переменных в выражение допускается не более одного раза;

- В каждом подвыражении данного выражения содержится не более одной не обрамленной структурными скобками e-переменной.

На данном определении завязаны реализованные в компиляторе Рефала-5λ алгоритмы преобразований функций. Определение L-выражения понадобится нам в дальнейшем.

1.2 Понятие экранирования предложений

Перечислив основные конструкции языка, перейдем к более детальному обсуждению поставленной проблемы.

Рассмотрим программу, представленную на Листинге 2.

```
$ENTRY Go {
  = <Test 'A'>
}
$ENTRY Test {
  s.X = True;
  'A' = False
}
```

Листинг 2 — Пример программы с экранированием предложений

Заметим, что при любом символе-аргументе функция `Test` вернет значение `True`. Т.е. сопоставление с образцом

```
'A' = False
```

никогда не будет выполнено. В таком случае будем говорить, что предложение

```
s.X = True
```

экранирует предложение

```
'A' = False
```

Экранирующие предложения такого вида могут быть написано только по ошибке. Причем зачастую такого рода ошибки не являются очевидными, что не позволяет быстро их отловить, и тем самым замедляет разработку ПО.

Далее рассмотрим пример функции на Листинге 3.

```
$ENTRY Test {
  e.x 'A' = 1;
  e.x t.1 t.2 t.3 e.y = 2;
  'A' t.1 = 3;
  'A' e.x0 = 4;
}
```

Листинг 3 — Пример функции с групповым экранированием

Предложение №4 экранируется группой предложений №1-3, но ни одним из них по отдельности. Случаи группового экранирования сложны для анализа и не предполагают эффективной реализации, поэтому в список требований к реализуемому алгоритму распознавание группового экранирование включено не будет [5].

Перечислим требования, предъявляемые к реализуемому алгоритму.

1.3 Анализ требований к реализуемому алгоритму

Как мы уже выяснили, предложение в программе может не выполняться, если его левая часть (образец + условия) описывает множество аргументов, являющихся подмножеством другого предложения, записанного выше.

Стоит остановиться на предложениях, содержащих условия. Такие предложения могут быть написаны намеренно ради побочного эффекта в условии и отката на последующие предложения. Поэтому будем считать, что подобные предложения могут экранироваться, но сами не могут экранировать.

Необходимо корректно обрабатывать связанные переменные. *Связанные* переменные — переменные из объемлющей области видимости, используемые в качестве контекста в текущей области видимости. В Листинге 4 приведен пример функции, проверяющей аргументы-термы на равенство.

Рассмотрим аналогичный пример, содержащий новые (переопределенные) переменные (Листинг 5). Предложение

```
t.Y^ = False
```

```
Eq {
  t.X t.Y
    , t.X
    : {
        t.Y = True;
        t.Other = False;
      }
}
```

Листинг 4 — Пример отсутствия экранирования с повторными переменными

означает, что левая часть представляет собой любой терм, т.к. имя `t.Y^` во время прохода рассахаривания будет преобразовано так, чтобы новое имя переменной не совпало ни с одним из уже используемых.

```
Eq {
  t.X t.Y
    , t.X
    : {
        t.Y = True;
        t.Y^ = False;
      }
```

```
}
```

Листинг 5 — Пример отсутствия экранирования с новыми переменными

Также необходимо учесть возможные повторные переменные. Исключение составляют лишь e-переменные: образцы с кратными e-переменными не буду проверяться ввиду многократного увеличения вычислительной сложности.

Алгоритм не должен порождать предупреждения, ошибочно сообщающие о несуществующем экранировании, чтобы не вводить в заблуждение пользователя.

2 Разработка

Практическая часть данной ВКР разделена на два этапа: технический — расширение возможностей компилятора Рефала-5λ путем реализации механизма выдачи предупреждений, и содержательный — разработка алгоритма распознавания экранирования.

Для добавления нового функционала проанализируем имеющиеся в компиляторе средства для работы с ошибками. На их основе напишем аналог — механизм предупреждений.

2.1 Реализация выдачи предупреждений пользователю

Компилятор Рефала-5λ имеет поддержку восстановления при ошибках, поэтому при компиляции файла может быть сформировано несколько сообщений, причем возникших на разных уровнях (лексический, синтаксический или семантический анализ). Для этой цели компилятор использует список сообщений об ошибках `t.ErrorList`, где они накапливаются, чтобы при завершении компиляции файла выводить их в порядке появления в исходном тексте. Формат `t.ErrorList` следующий:

```
[ErrorList (e.FileName) (t.SrcPos e.Message)*]

  t.SrcPos ::=

      NoPos

    | (FileLine s.LineNumber e.FileName)
```

```
    | (RowCol s.Row s.Col)
    | (FileRowCol (s.Row s.Col) e.FileName)
```

Создается список ошибок функцией `EL-Create`. Следующие функции выполняют действия над списком:

- Добавление ошибки в список `EL-AddError`, `EL-AddErrorAt`
- Склеивание двух списков `EL-Concat`
- Вывод сообщений об ошибках в стандартный вывод ошибок (англ. stderr) `EL-Destroy`

В компиляторе предусмотрена печать предупреждающего о нулевых байтах в составных символах сообщения. Принцип работы следующий. Осуществляется проход по токенам, сформированным лексическим анализатором, токены-предупреждения `TkWarning` распечатываются сразу после фазы лексического анализа и удаляются, остальные остаются. Функция `DisplayLexerWarnings` представлена на Листинге 6.

```
DisplayLexerWarnings {
  (e.FileName) e.Tokens
    = <Map
        {
          (TkWarning t.Pos e.Message)
            = <Prout e.FileName ':' <StrFromPos t.Pos>
              ':WARNING:' e.Message>;
          t.OtherToken = t.OtherToken;
        }
        e.Tokens
      >;
}
```

Листинг 6 — Функция `DisplayLexerWarnings`

Интерфейс было решено позаимствовать у известного проекта с открытым исходным кодом GCC [1]. Так программистам, знакомым с логикой взаимодействия с предупреждениями компиляторов GCC будет проще работать с компилятором Рефала-5λ.

Для хранения предупреждений в `t.ErrorList` необходимо было изменить формат типа данных (Листинг 7). Был добавлен абстрактный тип `t.Warning`, состоящий из символа `s.WarningId` — имени предупреждения, `t.SrcPos` — позиция предупреждения, `e.Message` — сообщение предупреждения.

Также в `t.Config` были добавлены 2 параметра:

- `(WarningIds nul-in-compound)` — имена предупреждений, которые будут проверяться при компиляции;
- `(WarningIdsAsErrors /* пусто */)` — имена предупреждений, которые следует считать ошибками и прерывать компиляцию.

```
[ErrorList (e.FileName) (t.Error*) (t.Warning*)]
t.Error ::= (t.SrcPos e.Message)
t.Warning ::= (s.WarningId t.SrcPos e.Message)
t.SrcPos ::=
      NoPos
    | (FileLine s.LineNumber e.FileName)
    | (RowCol s.Row s.Col)
    | (FileRowCol (s.Row s.Col) e.FileName)
s.WarningId ::=
      screening
    | nul-in-compound
    | init-final-entry
```

Листинг 7 — Представление в памяти списка ошибок `t.ErrorList`

Для добавления предупреждения в список были добавлены методы EL-AddWarning, EL-AddWarningAt. Модифицированные функции представлены в Приложении А (Листинг А.1).

Всего была реализована поддержка трех имен предупреждений:

- nul-in-compound;
- init-final-entry;
- screening.

Имя nul-in-compound означает наличие нулевого байта в составном символе. Под этим именем было интегрировано в общий фреймворк рассмотренное нами ранее соответствующее предупреждающее сообщение. Функция DisplayLexerMessages была заменена на FilterLexerWarnings (Листинг 8).

```
FilterLexerMessages {
  t.ErrorList e.Tokens
    = <MapAccum
        {
          t.ErrorList^
         (TkWarning t.Pos s.Type e.Message)
            = <EL-AddWarningAt
                t.ErrorList s.Type t.Pos e.Message
              > /* пусто */;
          t.ErrorList^ (TkError t.Pos e.Message)
            = <EL-AddErrorAt
                t.ErrorList t.Pos e.Message
              > /* пусто */;
          t.ErrorList^ t.OtherToken
            = t.ErrorList
              t.OtherToken
        }
        t.ErrorList e.Tokens
      >;
}
```

Листинг 8 — Функция DisplayLexerMessages

В ходе работы отдельно от основной задачи распознавания экранирования была реализована проверка, объявляются ли `$ENTRY` функции с именами `INIT` или `FINAL`. Данная проверка позволила сразу написать дополнительный работающий автотест для интерфейса предупреждений. Эта функция доступна под именем `init-final-entry`.

Алгоритм распознавания экранирования запускается при указании имени опции *`screening`*. Далее рассмотрим его.

2.2 Реализация алгоритма проверки экранирования

Приведем способы удовлетворения предъявленных к алгоритму требований.

Для каждой функции надо обрабатывать ее тело как последовательный набор предложений, а также рекурсивно обходить условия, присваивания, блоки и замыкания.

В полученном наборе предложений из тела каждой функции составляются пары предложений. Причем если «левое» предложение пары содержит хотя бы одно условие, то такая пара исключается из набора (предложение с условием может экранироваться, но не может экранировать).

К каждому предложению приписывается контекст в виде связанных переменных, причем каждая e-переменная обрамляется круглыми структурными скобками.

Решение конфликтов с новыми переменными (помеченными символом “^”) производится с помощью прохода рассахаривания — переименования переменных. Каждому имени переменной предварительно приписывается суффикс, указывающий на уровень вложенности области видимости, в которой

была создана данная переменная. Таким образом, перенумерованные новые переменные обрабатываются идентично обыкновенным переменным.

В силу особенностей языка (а именно широких возможностей для сопоставления у e- и t-переменных) задача распознавания экранирования в общем случае неразрешима. Однако, реализовав алгоритм на множестве наиболее используемых видов образцов, можно перекрыть удовлетворительное подмножество из всех случаев.

На этом основании было решено использовать два подхода для детектирования экранирования.

2.2.1 Метод обобщенного сопоставления с образцом

Первый заключается в следующем. Каждая пара предложений с контекстом проверяются алгоритмом обобщенного сопоставления с образцом, предложенным В.Ф. Турчиным. Функция `GenericMatch` была реализована в рамках ВКР П.А. Савельева [6] в 2018 году, а позже доработана в ВКР К.А Ситникова [4] в 2019 году:

```
<GenericMatch (e.Pattern) (e.LPattern)>
    == Clear (e.Val ':' t.Var)*
    == Contracted (((t.Var ':' e.Val)*)
                   ((e.Val ':' t.Var)*))*
    == Failure
    == Undefined
```

Алгоритм обобщенного сопоставления с образцом принимает на вход два аргумента — выражение общего вида и L-выражение, с которым производится

сопоставление. `Clear` обозначает, что сопоставление прошло успешно, `Contracted` — сопоставление есть, но присутствуют сужения, `Failure` и `Undefined` — решение найдено не было.

С учетом ограничения на L-выражение такой подход позволяет распознать лишь некоторое подмножество случаев экранирования. Стоит отметить, что в редких случаях `GenericMatch` распознает повторные t- и e-переменные, а также невозможность отождествления образцов, начинающихся с разных символов [4].

Рассмотрим пример:

```
$ENTRY Test {
  t.E e.X = 0;
  e.Y t.F = 1;
}
```

Предложение №2 экранирует предложение №1, т.к. оба предложения представляют из себя «один или более» терм, однако, используя алгоритм обобщенного сопоставления, проверить это не предоставляется возможным. Поэтому реализуем еще один алгоритм.

2.2.2 Понятия теории языков образцов

Вторая часть общего алгоритма основывается на теории языков образцов. Введем ряд определений.

Пусть V — множество переменных, Σ — алфавит констант.

Образец P назовем *плоским*, если его можно записать как строку из алфавита $V \cup \Sigma$ (т.е. не содержит круглых скобок).

Образец P называется *линейным*, если кратность каждой e-переменной в нём равна единице.

Подстановкой назовем морфизм

$$\sigma: (V \cup \Sigma) * \to (V \cup \Sigma) *$$

обладающим свойством сохранения констант:

$$\forall A \in \Sigma: \sigma(A) = A$$

Языком $L(P)$, распознаваемым образцом P, назовем множество элементов $\Phi \in \Sigma^*$, для которых существует подстановка $\sigma: \sigma(P) = \Phi$ [7]. Если допустима подстановка вида $\sigma(e.X) = \varepsilon$, то говорят, что язык $L(P)$ — *стирающий* (англ. EPL — erasing pattern language), иначе — *нестирающий* (англ. NEPL — non-erasing pattern language) [8] [9].

В нашем случае будут рассматриваться только стирающие языки, т.к. в Рефале-5 и Рефале-5λ нет типов переменных, допускающих 1 или более терм. Однако, такие переменные, называемые v-переменными, существуют в Рефале-4.

Теорема. Если $L(P_1) \subseteq L(P_2)$, то образец P_1 *сводится* к образцу P_2 (т.е. P_1 экранирует P_2) [10].

Рассмотрим произвольный образец P. *Якорем* будем называть подслово p образца P, такое, что:

1. p не содержит e-переменных
2. $\forall\, p'$, p' — надслово p, p содержит e-переменную

Например, образцу

$$P = e.x_1 \; \boldsymbol{A} \; \boldsymbol{B} \; e.x_2 \; e.x_3 \; \boldsymbol{C}$$

соответствует последовательность якорей, разделенных символом «||»:

$$\boldsymbol{A\ B\ ||\ C}$$

Пусть дан образец P, $t.x \in P$. Переменная $t.x$ называется *якорной*, если выполнено одно из следующих условий:

- $t.x$ имеет кратность ≥ 2;
- в P существует подслово p, не содержащее e-переменных, представимое в виде $p = p_1\ t.x\ p_2$, где p_1 и p_2 содержат как минимум один символ или t-переменную кратности ≥ 2.

Если ни одно из вышеперечисленных условий не выполняется, то переменная $t.x$ называется *плавающей.* Плавающая переменная — указатель на то, что в соответствующий фрагмент образца нельзя подставить пустое слово.

Пусть дан образец P следующего вида:

$$P = t.y_1\ \boldsymbol{t.y_2}\ e.x_1\ t.y_3\ t.y_4\ \boldsymbol{t.y_2}\ e.x_2\ t.y_5$$

Последовательность якорей образца P:

$$t.y_1\ \boldsymbol{t.y_2}\ ||\ t.y_3\ t.y_4\ \boldsymbol{t.y_2}\ ||\ t.y_5$$

где $t.y_1\ \boldsymbol{t.y_2}$ — якорные переменные**.**

Важным аспектом в реализации алгоритма является понятие является нормальной формы образца.

Считаем, что линейный образец P находится в *нормальной форме*, если:

- P не содержит ≥ 2 идущих подряд e-переменных;

- Каждая плавающая t-переменная в P предшествует e-переменной и следует за e-переменной.

Каждому линейному образцу P соответствует (с точностью до переименования переменных) ровно один образец P' в нормальной форме такой, что

$$L(P) = L(P')$$

Например, для образца

$$P = t.y_1\ t.y_2\ e.x_1\ t.y_3\ t.y_4\ t.y_2\ e.x_2\ t.y_5$$

где $t.y_1\ t.y_2$ — якорные переменные, а $t.y_3\ t.y_4\ t.y_5$ — плавающие, нормальной формой будет образец

$$P_N = t.y_1\ t.y_2\ e.x_1\ t.y_3\ \boldsymbol{e}._1\ t.y_4\ \boldsymbol{e}._2\ t.y_2\ e.x_2\ t.y_5\ \boldsymbol{e}._3$$

2.2.3 Метод, основанный на теории языков образцов

Единственным требование, которое предъявим к образцу-шаблону — отсутствие кратных e-переменных. Допускаются открытые e-переменные.

Семантическое вложение, т.е. вложение множества строк, описываемых образцами, сводится к синтаксическому, когда образец-шаблон линеен и находится в нормальной форме.

Алгоритм можно записать следующим образом:

1) Образец-шаблон P проверяется на линейность и приводится в нормальную форму;

2) Для пары исходных образцов E и P строится общий формат путем нахождения глобального сложнейшего обобщения (ГСО) [6]

$$G = < g_1 , \quad \dots , g_k >$$

где g_i — i-тый элемент ГСО;

3) Составляется кортежи (E_i, P_i) из соответствующих элементов g_i обобщения G пары образцов E и P;
4) Элементы кортежей сравниваются, т.е. для каждого решается уравнение вида $E_i : P_i$;
5) Решение уравнения $E_i : P_i$ порождает присваивания переменных образца вида $e_{i1} \dots e_{in} : p_i$;
6) В случае наличия в образце E повторных переменных, решения уравнений будут содержать присваивания с одинаковой правой частью: $e_{i1} \dots e_{in} : p_i$ $e_{j1} \dots e_{jm} : p_i$. Если $e_{i1} \dots e_{in}$ в точности не совпадает с $e_{j1} \dots e_{jm}$, экранирования нет и на этом проверку можно закончить;
7) Если ни для одной пары присваиваний не нашлось противоречий, образец P экранируется образцом E.

Функция `PatternLanguagesCheck` представлена на Листинге 9.

```
PatternLanguagesCheck {
  (e.ContextVars) (e.Pattern-L) (e.Pattern-R)
    , <FlattenPattern (e.Pattern-R)>
    : e.FlattenedPattern-R
    , <IsPatternLinear (e.FlattenedPattern-R)> : True
    = e.ContextVars e.Pattern-L : e.Pattern-L^
    = e.ContextVars e.Pattern-R : e.Pattern-R^
    = <Generalize (e.Pattern-L) (e.Pattern-R)>
```

```
    : e.Generalized
    = <RenameVars () e.Generalized> : e.Generalized^
    = <GenericMatch (e.Pattern-R) (e.Generalized) >
    : Clear e.Tuple-R
    = <GenericMatch (e.Pattern-L) (e.Generalized) >
    : Clear e.Tuple-L
    = <CreateTuplePairs (e.Tuple-L) (e.Tuple-R)>
    : e.Tuples
    = <CheckEveryMatch e.Tuples ()>;

  (e.ContextVars) (e.Pattern-L) (e.Pattern-R)
    = False;
}
```

Листинг 9 — Функция `PatternLanguagesCheck`

Рассмотрим шаг 3) подробнее. Опишем алгоритм решения уравнения с открытыми e-переменными.

Знаки E, P (с индексами и штрихами) будут означать выражения слева и справа, T и T_p — термы слева и справа, S — символы или s-переменные.

Алгоритм:

1) $T\ E' : T_p\ P'$ — добавляем уравнения $T : T_p$ и $E' : P'$;
2) $E'\ T : P'\ T_p$ — аналогично 1);
3) $\varepsilon : \varepsilon$ — успех;
4) $E : e.X$ — успех;
5) $E : e.X\ P'\ e.Y$ — осуществляем перебор всевозможных разбиений E как конкатенации $E = E'\ E''$ и добавляем уравнения $E'' : P'\ e.Y$;
6) $T : t.X$ — успех, добавляем подстановку $T \leftarrow t.X$;
7) $S : s.X$ — успех, добавляем подстановку $S \leftarrow s.X$;
8) $T : T$ — если слева и справа одинаковые значения, успех;

Иначе решений нет.

Функция обобщенного сопоставления объектного выражения с плоским линейным образцом получила название `ObjectMatch` (Листинг 10).

```
$ENTRY ObjectMatch {
  (e.Pattern) (e.LPattern)
    = <Solve
        (<ExtractVariables
          ((e.Pattern e.LPattern) (/* пусто */))>)
          ((e.Pattern) ':' (e.LPattern))
      >
    : {
        Success (() (e.Assigns))
          = Clear <FormatAssigns e.Assigns>;
        Success e.Solutions = Contracted e.Solutions;
        Failure = Failure;
        Undefined = Undefined;
      }
}
```

Листинг 10 — Функция `ObjectMatch`

В задаче распознавания экранирования нас не интересуют случаи сопоставления с сужением (`Contracted`), т.к. в этом случае считаем, что экранирования нет.

Плюсом описанного подхода является возможность работать с открытыми *e*-переменными.

3 Тестирование

Первоначальная сборка компилятора из исходного кода, разработка и тестирование производились на компьютере со следующей конфигурацией:

- Процессор: Intel(R) Core(TM) i5-8400 CPU @ 2.80GHz;
- ОЗУ: 8 Гбайт DDR4;
- ОС: Windows 10 (x64).

Используемый компилятор C++: MinGW (gcc version 9.2.0 MinGW.org GCC Build-20200227-1).

Компилятор Рефала-5λ собирался без оптимизаций.

Модификация и отладка платформозависимых скриптов из набора утилит компилятора дополнительно проводились в подсистеме WSL (Windows Subsystem for Linux) с установленным дистрибутивом Ubuntu 18.04.4 LTS.

Так как исходный код компилятора на момент написания данной работы размещен в репозитории [2], то для воспроизводимости полученных результатов приведем хеш коммита: `828142f3`.

3.1 Автотесты компилятора

Дерево исходников компилятора содержит средства для автоматизированного тестирования — коллекция самопроверяющихся тестов в папке `autotests`. Далее эти самопроверяющиеся тесты мы для краткости будем называть автотестами. Тесты запускаются скриптами `run.bat` (для платформы Windows) и `run.sh` (для платформы POSIX), и при возникновении ошибки выполнение всего набора прерывается.

Для тестирования необходимо было дополнить коллекцию автотестов и модифицировать существующие.

Был добавлен специальных формат для файлов, имена которых имеют суффикс вида «.WARNING.(s)ref». Такие автотесты запускаются два раза: первый раз с флагом -W<name>, где <name> — имя предупреждения из первого комментария в файле автотеста вида

```
* WARNING name
```

Пример автотеста для предупреждения о нулевых байтах в составных символах приведен в Листинге 11.

```
* WARNING nul-in-compound
$ENTRY Go {
  = <Eq ABC "ABC\x00DEF" "ABC\x00123">
}
Eq { s.1 s.1 s.1 = }
```

Листинг 11 — Автотест zerocompound.WARNING.ref

Был составлен набор автотестов, как содержащих случаи экранирования, так и полностью корректных. Приведем некоторые из них (а также основные сообщения из логов запуска) в качестве примеров.

На Листинге 12 представлен пример программы, когда экранируемые предложения расположены не непосредственно в теле функции, а в присваивании.

```
Test {
  /* empty */
    = {
        e.Var = 1;
        'I am screened' = 2;
      }
    : e.Var
    = e.Var;
}
Результат запуска:

Passing screening-assign.WARNING.ref (flags -Werror=screening)...
screening-assign.WARNING.ref:7:1: ERROR: Function `Test':
sentence $1=1\1$1 screens sentence $1=1\1$2 [-Werror=screening]
```

Листинг 12 — Автотест `screening-assign.WARNING.ref`

В следующем примере представлена программа, которая должна выполняться без предупреждений (Листинг 13).

```
AssignTest {
  e.X A e.Y
    = e.X : e.F
    = {
        e.Y = e.Y;
        e.F = e.X;
        'I am no screened' = e.Y
      };
}
Результат запуска:

Passing screening-assign.ref (flags -Wall -Werror)...
** Compilation succeeded **
```

Листинг 13 — Автотест `screening-assign.ref`

Здесь в присваивании создается переменная `e.F`, используемая далее в теле функции в качестве контекстной переменной, что и позволяет алгоритму корректно обработать последовательность предложений

```
e.Y = e.Y;
e.F = e.X;
'I am no screened' = e.Y
```

где их левые части состоят из контекстно-захваченных e-переменных, поэтому не экранируют последнее.

Алгоритм также распознает экранируемые предложения как в классических блоках (Листинг 14),

```
BracketsInPatternTest {
  s.X = True;
  (('a')) = :{'a' = 1; 'A' = 2; 'a' = 3};
  e.Y = False;
}
Результат запуска:

Passing screening-blocks.WARNING.ref (flag -Werror=screening)...
screening-blocks.WARNING.ref:7:1: ERROR: Function
`BracketsInPatternTest': sentence $2:1$1 screens sentence $2:1$3
[-Werror=screening]
```

Листинг 14 — Автотест `screening-blocks.WARNING.ref`

так и в замыканиях (Листинг 15).

```
BracketsClosureTest {
  'a' = 0;
  s.X = ({e.Y = 1; s.x = 2});
}
Результат запуска:

Passing screening-closure.WARNING.ref (flag -Werror=screening)...
screening-closure.WARNING.ref:7:1: ERROR: Function
`BracketsClosureTest': sentence $2\1$1 screens sentence $2\1$2 [-
Werror=screening]
```

Листинг 15 — Автотест `screening-closure.WARNING.ref`

Плавающие t-переменные также выявляются с помощью алгоритма, основанного на языках образцов (Листинг 16). Здесь и первое, и второе предложения состоят из 1 и более терма.

```
* WARNING screening
…
FLoatingTVarTest {
  t.X1 e.Y1 = 1;
  e.X2 t.Y2 = 2;
}
Результат запуска:

Passing screening-floating-t-var.WARNING.ref (flag -
Werror=screening)...
screening-floating-t-var.WARNING.ref:7:1: ERROR: Function
`FLoatingTVarTest': sentence $1 screens sentence $2 [-
Werror=screening]
```

Листинг 16 — Автотест `screening-floating-t-var.WARNING.ref`

Но если заменить t-переменные на s-переменные, то предупреждение не будет выведено, так как не каждый терм является символом. (Листинг 17).

```
EVarsTest {
  e.X s.Y = True;
  s.Y e.X = False
}
Результат запуска:

Passing screening-e-vars.ref (flags -Wall -Werror)...
** Compilation succeeded **
Special warning tests passed
```

Листинг 17 — Автотест `screening-e-vars.ref`

Таким образом, тестирование показало, что алгоритм работает верно на всех синтаксических конструкциях языка.

3.2 Замеры производительности

После проверки корректности алгоритма было решено его применить к реальным программам на Рефале, а также замерить скорость работы с включенными предупреждениями и без них.

Несмотря на доступный флаг компиляции `-Wall`, использовать будем `-Wscreening`, так как алгоритмы остальных предупреждений являются алгоритмически простыми и, следовательно, вычислительно быстрыми.

Так как у одного замера точность может не отражать реальные показатели производительности, проведем серию опытов. Будем использовать метод, основанный на вычислении медианы и границ двух квартилей. На Рисунке 1 изображена схема метода.

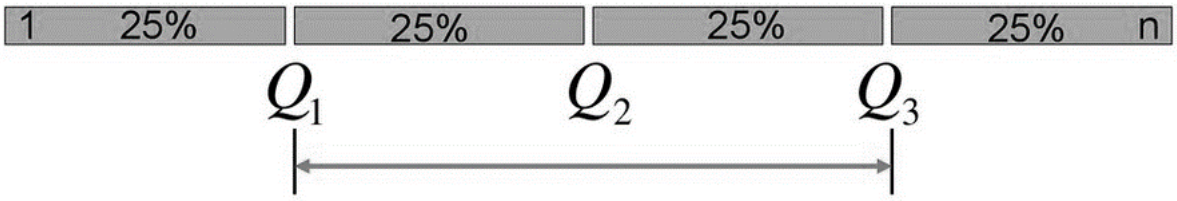

Рисунок 1 — Схема метода замера. Q_1 и Q_3 обозначают границы доверительных интервалов, Q_2 — медиана (изображение с сайта cf.ppt-online.org)

Суть заключается в следующем. Проводится серия из n измерений:

$$p = \{p_1, \dots, p_n\}$$

Полученные данные сортируются по возрастанию ключа (например, время выполнения программы, число шагов, и т.д.). Набор значений условно делится на 4 части, причем если $n = 4k + 1$, то значение p_{2k+1} называется *медианой*, а p_{k+1} и p_{3k+1} — *границами* квартилей.

Далее меняются условия эксперимента и проводится повторная серия измерений:

$$p' = \{p_1', \dots, p_n'\}$$

Если

$$(Q_1, Q_3) \cap (Q_1', Q_3') = \emptyset$$

то результаты измерений считаются статистически достоверными.

Для проведения тестов использовалось значение $n = 13\ (= 4 \cdot 3 + 1)$.

3.2.1 Проверка кодовых баз нескольких крупных программ на Рефале-5(λ)

MSCP-A — модельный суперкомпилятор для алгоритмически полного подмножества базисного Рефала [12]. SCP4 — суперкомпилятор, написанный на функциональном языке Рефал-5 [13]. Так как самоприменимый компилятор Рефала-5λ является надмножеством Рефала-5, применим разработанный алгоритм к исходному коду вышеупомянутых суперкомпиляторов, а также к самому себе.

Проведем три эксперимента, используя в качестве входных данных:

1) исходный код MSCP-A;
2) исходный код SCP4;
3) исходный код Рефала-5λ.

Замеры производительности в первом эксперименте отражены в Таблице 1.

Параметры замера	Медиана Q_2, с	Доверительный интервал $Q_1 Q_3$, с

Исходный код MSCP-A	13.847	(13.758, 14.107)
Исходный код MSCP-A, с проверкой экранирования	18.497	(18.467, 18.662)

Таблица 1 — Результаты замеров времени компиляции исходного кода суперкомпилятора MSCP-A

Как видно из таблицы, при изменении параметров замера доверительные интервалы не пересеклись. При включенном режиме распознавания экранирования, процесс компиляции занял на 33,5% больше времени.

В результате работы алгоритм выдал 15 предупреждающих сообщений. 12 из них соответствуют реальным ошибкам в коде. Три других предупреждения связаны со следующей функцией:

```
$ENTRY SubdirSign {
    * Windows
        = '\\';
    * Linux
        = '/';
    * Macintosh (Apple)
        = ':' ;
}
```

Управляющие комментарии организуют условную компиляцию в зависимости от используемой операционной системы, поэтому в действительности это не является ошибкой. Добавление обработки таких комментариев не доступно на данный момент без расширения синтаксического дерева, что выходит за рамки данной работы, но может послужить одним из путей дальнейшего развития ВКР.

Во втором эксперименте время компиляции в режиме обнаружения предупреждений почти в 2 раза больше (на 90%) по сравнению с компиляцией без параметров (Таблица 2).

Параметры замера	Медиана Q_2, с	Доверительный интервал Q_1Q_3, с
Исходный код SCP4	23.011	(22.872, 23.185)
Исходный код SCP4, с проверкой экранирования	43.79	(43.406, 43.909)

Таблица 2 — Результаты замеров времени компиляции исходного кода суперкомпилятора SCP4

Было обнаружено 10 предупреждений. Все предупреждения корректны.

Результаты третьего эксперимента представлены в Таблице 3.

Параметры замера	Медиана Q_2, с	Доверительный интервал Q_1Q_3, с
Исходный код Рефала-5λ	22.015	(21.901, 22.089)
Исходный код Рефала-5λ, с проверкой экранирования	45.165	(44.833, 45.389)

Таблица 3 — Результаты замеров времени компиляции исходного кода компилятора Рефала-5λ

Было обнаружено 3 предупреждений. Время компиляции в режиме обнаружения экранирования увеличилось вдвое (на 105%).

Значительная разница в приросте времени компиляции между экспериментами обосновывается тем, что в исходном коде SCP4 тела функций состоят из большего числа предложений, чем в MSCP-A; более того, присутствует несколько функций, тела которых содержат до 100 образцов. Исходный код Рефала-5λ содержит меньше функций с большими телами, однако общий объем исходного кода значительно больше, чем в SCP4.

Так как алгоритм генерирует пары образцов $(P_i;\ P_j),$ следующим образом:

$$(P_i;\ P_j), i = 1, \dots N;\ j\ =\ i, \dots, N$$

общее число пар оценивается как

$$O\left(\frac{n^2}{2}\right) = O(n^2)$$

При увеличении набора сравниваемых предложений внутри одного тела, условия, присваивания или блока, вычислительная сложность возрастает квадратично.

Таким образом, тестирование на реальных программах показало, что инструмент способен. В больших программах при каждой сборке не следует использовать флаг `-Wscreening (-Wall)`, т.к. ввиду нетривиальной алгоритмической сложности время компиляции может возрасти в 2 раза. Однако если запускать алгоритм с некоторой периодичностью, либо работать над небольшим проектом, вполне допустимо проверять экранирование по умолчанию с целью своевременного выявления ошибок.

4 Руководство пользователя

Приведем краткое руководство по использованию реализованного механизма предупреждений. Управление предупреждениями производится через флаг `-W` командной строки. Опция указывается сразу после `-W`. Доступные опции:

- `-Wall` — включение всех предупреждений компилятора;
- `-W<name>` — включение конкретного предупреждения по имени `name`;
- `-W[no-]error[=<name>]` — включить/отключить режим, в котором все предупреждения (или конкретное предупреждение) считаются ошибками и прерывают компиляцию (квадратные скобки `[...]` означают то, что их содержимое указывать не обязательно).

Если была указана опция `-Werror=<name>`, и указанное предупреждение было обнаружено, компилятор дополнительно выведет сообщение следующего вида:

```
rlc.bat -Werror=nul-in-compound -Wall hello.ref
  =>
hello.ref:2:23: ERROR: Zero byte in compound symbol [-
Werror=nul-in-compound]
hello.ref: some warnings being treated as errors
```

а если была указана опция *`-Werror`* без имени, то вывод изменится:

```
rlc.bat -Werror -Wall hello.ref
=>
```

```
hello.ref:2:23: ERROR: Zero byte in compound symbol [-
Werror=nul-in-compound]
hello.ref: all warnings being treated as errors
```

Сообщения об экранировании имеют следующий специальный формат:

```
example.ref:3:4:WARNING:Function `Foo': sentence
        $1=2:3$5 screens sentence $1=2:3$7
```

Это означает, что в функции `Foo` в пятое предложение (`$5`) в третьем блоке (`:3`) второго присваивания (`=2`) первого предложения (`$1`) экранирует седьмое предложение (`$7`) того же блока. Описанная система именования принята в компиляторе, а именно в отладочных дампах, поэтому цепочки суффиксов программисту на Рефале-5λ знакомы [11].

ЗАКЛЮЧЕНИЕ

В ходе работы был сделан обзор на виды предупреждающих сообщений в компиляторе языка Рефал-5λ. Добавлен механизм выдачи предупреждений. Были определены требования к алгоритму. Разработан подход к выявлению случаев экранирования предложений. Проведено тестирование нового функционала как на автотестах, так и на реальных программах, включая самоприменение к исходному коду компилятора. Таким образом, поставленные в начале работы цели реализованы.

В качестве дальнейшего развития данной ВКР предполагается несколько подходов.

1) Расширение теории языков образцов для более обширного подмножества распознаваемых случаев экранирования (например, поддержка кратных e-переменных в частных случаях);
2) Модификация алгоритма под задачу детектирования группового экранирования предложений. Решение проблем такого класса позволит приблизить исходную задачу к более общему решению;
3) Вычислительная оптимизация используемых алгоритмов, например, перебора пар образцов, методов решения обобщенных уравнений;
4) Предусмотреть управляющие комментарии для ручного подавления предупреждений там, где программист намеренно допускает ошибку ради тех или иных целей, а также обеспечить обработку специальных комментариев для удаления побочных предупреждений об экранировании;
5) Часть данной работы, связанная с реализацией механизма выдачи предупреждений, также может послужить фундаментом для расширения возможностей компилятора, например, добавление других типов предупреждений в фреймворк.

6) Реализовать дополнительный опциональный более легковесный алгоритм проверки экранирования для эффективного использования при работе с большими проектами.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Warning Options (Using the GNU Compiler Collection (GCC)) [Электронный ресурс]. URL: https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html (Дата обращения: 30.05.2020).
2. Репозиторий компилятора РЕФАЛ-5λ: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/refal-5-lambda (Дата обращения: 30.05.2020).
3. Турчин В.Ф. Эквивалентные преобразования программ на РЕФАЛе // В сб.: Автоматизированная система управления строительством. Труды ЦНИПИАСС, N 6. —Москва: ЦНИПИАСС, 1974.
4. Ситников К.А. Встраивание функций в компиляторе РЕФАЛ5-λ: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/РП3_Ситников_Прогонка_2019.pdf (Дата обращения: 30.05.2020).
5. Kari, L., Mateescua, A., Paun, G., Salomaa, A.: Multi-pattern languages. In Theoretical Computer Science 141, 253-268 (1995).
6. Савельев П.А. Улучшенная оптимизация совместного сопоставления с образцом в компиляторе Рефала-5-лямбда: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/refal-5-lambda/blob/master/doc/OptPattern/Савельев_Записка_2018.pdf (Дата обращения: 30.05.2020).
7. Angluin, D.: Finding Patterns Common to a Set of Strings. In Journal of Computer and System Sciences. 21: 46—62 (1980).
8. Shinohara, T.: Polynomial time inference of pattern languages and its applications, Proceedings of the 7th IBM Symposium on Mathematical Foundations of Computer Science, 191—209, (1982).

9. Jiang, T., Salomaa, A., Salomaa, K. and Yu, S.: Inclusion is undecidable for pattern languages, In Proceedings of the 20th International Colloquium, ICALP’93, Lecture Notes in Computer Science, Springer-Verlag, 301—312, (1993).
10. Reidenbach, D.: Discontinuities in pattern inference. In Theoretical Computer Science 397, 166—193 (2008).
11. Суффиксы присваиваний, условий, блоков и вложенных функций: [Электронный ресурс]. URL: https://bmstu-iu9.github.io/refal-5-lambda/B-reference.html#суффиксы-присваиваний-условий-блоков-ивложенных-функций (Дата обращения: 30.05.2020).
12. Модельный суперкомпилятор MSCP-A: [Электронный ресурс]. URL: http://refal.botik.ru/mscp/ (Дата обращения: 30.05.2020).
13. Немытых А.П. Суперкомпилятор SCP4: Общая структура. — М.: Издательство ЛКИ, 2007. — 152 с.: ISBN 978-5-382-00365-8

ПРИЛОЖЕНИЕ А

В данном приложении приведены реализации основных функций алгоритма проверки экранирования.

```
$ENTRY CheckScreening {
 (e._ screening e._) (e.AST)
   = <FindFunctionsPosInAST e.AST>
   : (e.FunctionsPos) e.AST^
   = <Pass-RemovePos e.AST> : e.AST^
   = <Pass-EnumerateVariables e.AST> : e.AST^
   = <Zip (e.AST) (e.FunctionsPos)> : e.FunctionsWithPos
   = <Map &CheckFunction e.FunctionsWithPos>;

 (e._) (e.AST) = /* пусто */
}

FindFunctionsPosInAST {
  e.AST
    = <MapAccum
        {
          (e.Positions)
          (Function t.SrcPos s.ScopeClass (e.Name)
          Sentences e.Sentences)
            = (e.Positions t.SrcPos)
              (Function t.SrcPos s.ScopeClass (e.Name)
               Sentences e.Sentences);

          (e.Positions) t.Other = (e.Positions)
        }
        () e.AST
      >;
}

CheckFunction {
  ((Function s.ScopeClass (e.Name) Sentences
```

```
  e.Sentences) t.SrcPos)
    = <CheckSentences t.SrcPos (e.Name) () ()
      (e.Sentences)>
}

CheckPatternPairs {
  t.SrcPos (e.FuncName) (e.Prefix) (e.ContextVars)
  (e.Pairs)
    = <Map
      {
        (
          (((e.Pattern-L) e._ (e._) (e._)) s.Num-L)
          (((e.Pattern-R) e._ (e._) (e._)) s.Num-R)
        )
          = <Unique <ExtractVariables-Expr
            e.ContextVars>> : e.ContextVars^
          = <CreateContext NoMarkupContext
            e.ContextVars> : e.ContextVars^
          = <GenericMatch
              (e.ContextVars e.Pattern-L)
              (e.ContextVars e.Pattern-R)
            >
          : {
              e._, (e.Pattern-L) (e.Pattern-R) : () ()
                = <CreateScreeningWarning
                    t.SrcPos
                    (e.FuncName)
                    (e.Prefix '$' <Symb s.Num-R>)
                    (e.Prefix '$' <Symb s.Num-L>)
                  >;
              Clear e._
                = <CreateScreeningWarning
                    t.SrcPos
                    (e.FuncName)
                    ('GM' e.Prefix '$' <Symb s.Num-R>)
                    ('GM' e.Prefix '$' <Symb s.Num-L>)
                  >;

              e._ = <PatternLanguagesCheck
```

```
                        (e.ContextVars)
                        (e.Pattern-L)
                        (e.Pattern-R)
                      >
                    : {
                        True e.Other
                          = <CreateScreeningWarning
                              t.SrcPos
                              (e.FuncName)
                              ('PL' e.Prefix '$'
                               <Symb s.Num-R>)
                              ('PL' e.Prefix '$' <Symb
                              s.Num-L>)
                            >;

                        False = /* пусто */;
                    }
              }
      }
      e.Pairs
    >;

  /* пусто */ = /* пусто */;
}

PatternLanguagesCheck {
  (e.ContextVars) (e.Pattern-L) (e.Pattern-R)
    , <FlattenPattern (e.Pattern-R)>
    : e.FlattenedPattern-R
    , <IsPatternLinear (e.FlattenedPattern-R)> : True
    = e.ContextVars e.Pattern-L : e.Pattern-L^
    = e.ContextVars e.Pattern-R : e.Pattern-R^
    = <Generalize (e.Pattern-L) (e.Pattern-R)>
    : e.Generalized
    = <RenameVars () e.Generalized> : e.Generalized^
    = <GenericMatch (e.Pattern-R) (e.Generalized) >
    : Clear e.Tuple-R
    = <GenericMatch (e.Pattern-L) (e.Generalized) >
    : Clear e.Tuple-L
```

```
    = <CreateTuplePairs (e.Tuple-L) (e.Tuple-R)>
    : e.Tuples
    = <CheckEveryMatch e.Tuples ()>;

  (e.ContextVars) (e.Pattern-L) (e.Pattern-R)
    = False;
}

CheckEveryMatch {
  ((e.Pattern-L) (e.Pattern-R)) e.Tuples (e.Assigns)
    = <ObjectMatch (e.Pattern-L) (e.Pattern-R)>
    : {
        Clear e.NewAssigns
          = <AppendAssigns (e.Assigns) e.NewAssigns>
          : {
              Success e.UpdatedAssigns
                = <CheckEveryMatch e.Tuples
                  (e.UpdatedAssigns)>;

              Fails = False;
            };

        e.Other = False;
      };

  /* пусто */ (e.Assigns) = True (e.Assigns);
}

CreateTuplePairs {
  (e.Tuple-L) (e.Tuple-R)
    = <DelAccumulator
        <MapAccum
          {
            (t.L e.Tail-L) t.R
              , t.L : (e.Vars-L ':' e._)
              , t.R : (e.Vars-R ':' e._)
              = (e.Tail-L) ((e.Vars-L)
                (<Normalize e.Vars-R>));
          }
```

```
          (e.Tuple-L) e.Tuple-R
        >
      >;
}

Normalize {
  e.Pattern
    = <MarkupMultipleAnchorTVars e.Pattern>
    : e.MarkedPattern
    = <MarkupNeighbourAnchorTVars e.MarkedPattern>
    : e.MarkedPattern^
    = <InsertFakeEVars e.MarkedPattern>
    : e.NormalizedPattern
    = <DeleteMarkup e.NormalizedPattern>
}

DeleteMarkup {
  e.MarkedPattern
    = <Map
        {
          (Anchor t.Item) = t.Item;
          (Float t.Item) = t.Item;
          t.Item = t.Item;
        }
        e.MarkedPattern
      >;
}

InsertFakeEVars {
  e.Prefix
  (TkVariable 'e' e.Index1)
  (TkVariable 't' e.Index2)
  (TkVariable 'e' e.Index3)
  e.Suffix
    = <InsertFakeEVars
        e.Prefix
        (TkVariable 'e' e.Index1)
        (Float (TkVariable 't' e.Index2))
        (TkVariable 'e' e.Index3)
```

```
        e.Suffix
      >;

  e.Prefix
  (TkVariable 'e' e.Index1) (TkVariable 't' e.Index2)
  e.Suffix
    = <InsertFakeEVars
        e.Prefix
        (TkVariable 'e' e.Index1)
        (Float (TkVariable 't' e.Index2))
        (TkVariable 'e' e.Index2 'fake')
        e.Suffix
      >;

  e.Prefix
  (TkVariable 't' e.Index2) (TkVariable 'e' e.Index3)
  e.Suffix
    = <InsertFakeEVars
        e.Prefix
        (TkVariable 'e' e.Index2 'fake')
        (Float (TkVariable 't' e.Index2))
        (TkVariable 'e' e.Index3)
        e.Suffix
      >;

  e.Prefix
  (TkVariable 't' e.Index2)
  e.Suffix
    = <InsertFakeEVars
        e.Prefix
        (TkVariable 'e' e.Index2 'fake1')
        (Float (TkVariable 't' e.Index2))
        (TkVariable 'e' e.Index2 'fake2')
        e.Suffix
      >;

   e.Other = e.Other;
}
```

```
MarkupNeighbourAnchorTVars {
  e.MarkedPattern
    = <MarkupNeighbourHelper () (e.MarkedPattern)>;
}

MarkupNeighbourHelper {
  (e.Prefix) ((TkVariable 't' e.Index) e.Suffix)
    , <HasAnchorsPrefix e.Prefix> : True
    , <HasAnchorsSuffix e.Suffix> : True
    = <MarkupNeighbourHelper
        (e.Prefix (Anchor (TkVariable 't' e.Index)))
        (e.Suffix)
      >;
  (e.Prefix) (t.Item e.Suffix)
    = <MarkupNeighbourHelper
        (e.Prefix t.Item)
        (e.Suffix)
      >;

  (e.Prefix) () = e.Prefix;
}

HasAnchorsPrefix {
  e.Prefix (Anchor (TkVariable 't' e.Index))
    = True;

  e.Prefix (Symbol Char e.Index)
    = True;

  e.Prefix (TkVariable 'e' e.Index)
    = False;

  e.Prefix t.Other
    = <HasAnchorsPrefix e.Prefix>;

  /* пусто */ = True;
}

HasAnchorsSuffix {
```

```
  (Anchor (TkVariable 't' e.Index)) e.Suffix
    = True;

  (Symbol Char e.Index) e.Suffix
    = True;

  (TkVariable 'e' e.Index) e.Suffix
    = False;

  t.Other e.Suffix
    = <HasAnchorsPrefix e.Suffix>;

  /* пусто */ = True;
}

MarkupMultipleAnchorTVars {
  e.Pattern
    = <CountTVarsMultiplicity e.Pattern>
    : t.Multiplicity
    = <Map
        {
          (TkVariable 't' e.Index)
            , <HasMultiplicity
            (TkVariable 't' e.Index) t.Multiplicity>
            : True
            = (Anchor (TkVariable 't' e.Index));

          t.Other = t.Other;
        }
        e.Pattern
      >;
}

CountTVarsMultiplicity {
  e.Pattern
    = <MapAccum
        {
          (e.Accum) (TkVariable 't' e.Index)
            = <IncMultiplicity
```

```
                (TkVariable 't' e.Index)
                (e.Accum)
              >
              /* пусто */;

          (e.Accum) t._ = (e.Accum) /* пусто */;
        }
        () e.Pattern
      >
    : {
        t.Accum e._ = t.Accum;
      }
}

HasMultiplicity {
  (TkVariable 't' e.Index)
  (
    e.Accum-L
    ((TkVariable 't' e.Index) s.Count)
    e.Accum-R
  )
    = s.Count
    : {
        1 = False;
        s._ = True;
      };
}

IsPatternLinear {
  (e._ (TkVariable 'e' e.VarName) e._
       (TkVariable 'e' e.VarName) e._)
    = False;
  (e._)
    = True;
}

CreateScreeningWarning {
  t.SrcPos (e.FunName) (e.Screening) (e.Screened)
    = (Warning
```

```
          screening
          t.SrcPos
          Screening
          (e.FunName)
          (e.Screening)
          (e.Screened)
        )
}
```

Листинг А.1 — Основные функции алгоритма проверки экранирования