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 ПРИКЛАДНАЯ МАТЕМАТИКА И ИНФОРМАТИКА

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ ПО КУРСУ
Конструирование компиляторов

НА ТЕМУ:
Выдача предупреждений о связанных переменных в образцах в Рефале-5λ

Студент ИУ9-71Б _____ Бакланов Л. В.
(Группа) (Подпись, дата) (И.О.Фамилия)

Руководитель курсовой работы _____ Коновалов А. В.
(Подпись, дата) (И.О.Фамилия)

2021

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Предупреждения компилятора — это сообщения, которые генерирует компилятор при анализе потенциально некорректных участков исходного кода программы. Они сигнализируют программисту о необходимости повторно проверить предположительно ошибочный код. При этом предупреждения, в отличие от ошибок, не должны прерывать процесс компиляции и носить рекомендательный характер.

Целью данной работы является выдача предупреждений компилятора для потенциально ошибочно связанных переменных в образцах в компиляторе языка Рефала-5λ [1]. В рамках работы требуется рассмотреть проблемные случаи со связанными переменными, разработать эвристику, позволяющую распознавать проблемные случаи и реализовать выдачу предупреждений.

1. Обзор предметной области

1.1. Обзор Рефала-5λ

РЕФАЛ (Рекурсивных Функций АЛгоритмический язык) — функциональный язык программирования, в основе принципа работы которого лежат символьные вычисления [2]. Язык Рефал-5λ — точное надмножество языка Рефала-5. Главным отличием языка Рефал-5λ является поддержка функций высшего порядка. Отличительная особенности компилятора Рефал-5λ — наличие удобного интерфейса с языком C++ и поддержка возможности компиляции в C++. При этом одной из основных целей языка является сохранение статуса точного надмножества классического Рефала-5. То есть любая программа на Рефал-5 должна идентично выполняться на Рефале-5λ.

Программа на Базисном Рефале состоит из набора функций. Функция определяется своим именем и телом. Тело функции состоит из списка предложений (Листинг 1).

```
ИмяФункции {
  предложение1;
  предложение2;
  ...
  предложениеN;
}
```

Листинг 1. Конструкция функции.

Каждое предложение представляет из себя выражение вида:

```
Образец = результат;
```

Образец задаёт множество значений аргумента функции, к которому применимо предложение. Сопоставление аргумента с образцом подразумевает проверку его вхождения в допустимую область определения. Результат задаёт значение, которое вернёт функция при удачном сопоставлении с образцом. В

случае неуспешного сопоставления с образцами во всех предложениях функции программа аварийно завершается.

В ходе выполнения программы производятся операции с данным Рефала - объектными выражениями. Объектное выражение является последовательностью объектных термов, каждый из которых является либо атомарным символом (литерой, числом, идентификатором и т.д.), либо выражением в квадратных или круглых скобках. Выражение в квадратных скобках является объявлением АТД — абстрактного типа данных, а выражение в круглых скобках является составным скобочным термом.

В результатном выражении и образце могут содержаться переменные. Всего в Рефале существует 3 типа переменных:

- s-переменные — символы;
- t-переменные — термы;
- e-переменные — произвольные выражения, в том числе пустые.

Переменные записываются как ⟨тип⟩.⟨индекс⟩, где ⟨тип⟩ — один из символов s, t или e, ⟨индекс⟩ — последовательность латинских букв, цифр, знаков прочерка и дефиса, не может начинаться на дефис. Переменные в образце в процессе сопоставления с аргументом принимают свои значения, равные определенным частям объектного выражения аргумента функции. В случае удачного сопоставления, после присваивания значений переменным они могут быть использованы в правой части предложения при построении результата.

Левая часть предложения в Рефале-5λ после образца может содержать условия. Условие описывает дополнительные ограничения. Оно начинается со знака «,» и состоит из пары — расширенного результатного выражения (левой части условия) и образца (правой части условия), разделённых знаком «:». Для одного образца может быть несколько условий. Пример предложения с

условиями представлен на Листинге 2. Условие выполняется, если удалось сопоставить значение левой части с образцом.

```
CheckAB {
  s.1 s.2, s.1 : 'a', s.2 : 'b' = True;
  e.Another = False;
}
```

Листинг 2. Функция с несколькими условиями.

Правая часть выражения в Рефале-5λ может содержать присваивания и классические блоки. Присваивания служат для вычисления дополнительных значений и присваивания их переменным. Присваивание выглядит как расширенное результатное выражение, за которым следует левая часть. При этом они разделены знаком «:». При неуспешном сопоставлении с левой частью программа аварийно завершается. Присваиваний, как и условий может быть несколько. Таким образом, несколько присваиваний могут являться непрерывной цепочкой для обработки данных (Листинг 3). Расширенное результатное выражение — результатное выражение, после которого следует несколько тел функций через знак «:». Вышеупомянутый классический блок состоит из результатного выражени и единственного тела функции.

```
ChainProcessing {
  e.Arg = <SomeProcessing e.Arg> : e.Arg2
        = <AnotherProcessing e.Arg2> : e.Arg3
        = <FinalProcessing e.Arg3>;
}
```

Листинг 3. Цепочка присваиваний в предложении.

Рефал-5λ также поддерживает вложенные функции, которые записываются как тело функции в фигурных скобках. Помимо этого в данном языке были реализованы замыкания, которые представляют собой либо ссылку на глобальную именованную функцию, либо объект безымянной функции. Такие объекты часто применяется, например, при использовании функций высшего порядка из библиотеки LibraryEx [4].

1.2. Проблема связанных переменных в образцах

В приведённом на Листинге 3 примере конструкция выглядит избыточно из-за создания новой нумеруемой переменной с новым именем в каждом присваивании. В отличие от данного примера, если бы название переменной было осмысленным, то данная нумерация выглядела бы ещё более избыточно. В Рефал-5λ была добавлена синтаксическая возможность избежать подобной нумерации. Если в выражении в образце после имени переменной указать знак «^», то эта переменная сокроет одноимённую, связанную раньше. В данном образце она будет считаться новой, и в оставшейся части предложения переменная с этим именем будет связана уже с новым значением [1]. Позже данная переменная может быть скрыта неограниченное количество раз. Переписанный с использованием этой возможности пример представлен на Листинге 4.

```
ChainProcessing {
  e.Arg = <SomeProcessing e.Arg> : e.Arg^
        = <AnotherProcessing e.Arg> : e.Arg^
        = <FinalProcessing e.Arg>;
}
```

Листинг 4. Использование конструкции с символом «^».

Использование «^» при написании программ на Рефале-5λ позволяет избежать излишней нумерации переменных и делать программы более понятными и читабельными. Однако в результате добавления данной конструкции возникла проблема — при написании программы легко забыть дописать символ «^» рядом с переопределяемой переменной. В одних случаях данная ошибка приведёт к снижению быстродействия, в других — может привести к серьёзной логической ошибке в коде, которую достаточно сложно будет обнаружить в дальнейшем. Рассмотрим случай совершения данной

ошибки в контексте рассмотренной выше функции *ChainProcessing*. На Листинге 5 представлена «имитация» забытого символа «^» при написании данной функции.

```
ChainProcessing {
  e.Arg = <SomeProcessing e.Arg> : e.Arg
        = <AnotherProcessing e.Arg> : e.Arg^
        = <FinalProcessing e.Arg>;
}
```

Листинг 5. Функция с «забытым» символом «^».

В данном случае в первом присваивании производится сопоставление результата работы функции *SomeProcessing* с изначальным аргументом функции *ChainProcessing*. Если функция *SomeProcessing* во всех случаях возвращает результат, отличный от переменной *e.Arg*, то данная ошибка будет обнаружена при первой попытке запустить программу, вызывающую функцию *ChainProcessing*. Однако, если функция *SomeProcessing* может вернуть исходный аргумент, то данная ошибка может быть обнаружена не сразу и дальнейшее её обнаружение будет более проблематичным.

2. Разработка

Для решения данной проблемы необходимо было разработать механизм, который помогал бы разработчику увидеть ошибку в виде забытого символа «^». При этом важно, чтобы обнаруженная потенциальная ошибка не останавливала компиляцию. Для данного случая достаточно удобно было оформить это в виде предупреждений компилятора — сообщений, выдаваемых в процессе компиляции. Было решено реализовать 2 типа таких предупреждений:

- предупреждения типа *repeated*, выдача которых производится в соответствии с некоторой оптимальной эвристикой;
- предупреждения типа *repeated-maybe*, которые выдаются для всех связанных переменных в образцах, которые не были обозначены как *repeated*.

Очевидно, что предупреждения типа *repeated-maybe* чаще всего будут выдаваться на безошибочные участки кода, где связанные переменные используются повторно по задумке программиста. Однако в случае неидеальной эвристики, выбранной для предупреждений типа *repeated*, среди всех предупреждений *repeated-maybe* могут попадаться те, которые действительно укажут на ошибки в коде.

Приведённые в предыдущем разделе конструкции языка Рефал-5λ показывают, что образцы могут встречаться не только в предложениях в изначальной функции, но ещё в условиях, присваиваниях, блоках, вложенных функциях внутри других предложений. Поэтому эвристика должна строиться на том, в какой именно конструкции используется какой-либо образец. Для выбора оптимальной эвристики были изучены коммиты из репозитория компилятора Рефала-5λ [1], которые добавляли пропущенные ранее символы «^». Примеры таких изменений в коммитах приведены на Листингах 6-8 с

необходимым контекстом. Добавленные в данных коммитах символы «^» дополнительно подчёркнуты.

```
...
    = <CreateMetaFunctions e.AST> : e.AST^
    = e.AST
    : {
        e.AST-B (NativeBlock t.SrcPos e.Name) e.AST-E
          = Success WithNative
            e.AST-B (NativeBlock t.SrcPos e.Name) e.AST-E;

        e.AST-B
        (Function t.SrcPos s.ScopeClass (e.Name)
        NativeBody e.Body)
        e.AST-E
          = Success WithNative
            e.AST-B
            (Function t.SrcPos s.ScopeClass (e.Name)
            NativeBody e.Body)
            e.AST-E;

        e.AST^ = Success NoNative e.AST;
    };
...
```

Листинг 6. Пример из коммита №1.

```
...
= <Map &MakeGraphNode e.AST> : e.Graph
= <Map &RemoveReferenceTagFromChildren e.Graph> : e.Graph^
= <MapAccum &UnsuffixedFunctions () e.Graph>
: (e.Roots) e.Graph^
```

Листинг 7. Пример из коммита №2.

```
RemoveAssigns-WindBlocks {
  (e.Result) e.Blocks
    = <Reduce
        {
          (e.Result^) ((e.BlockName) e.Body)
```

```
            = ((#CallBrackets (#Closure (e.BlockName) e.Body)
               e.Result));
        }
        (e.Result) e.Blocks
    >;
}
```

Листинг 8. Пример из коммита №3.

В примере на Листинге 6 демонстрируется пример с наименьшим отрицательным влиянием данной ошибки - потеря быстродействий, так как переменная *e.AST* сначала копируется, а потом проверяется равенство.

В примере на Листинге 7 в последнем присваивании демонстрируется потенциальная ошибка из-за забытого символа «^». Но в данном случае это не является фактической ошибкой, так как функция *UnsuffixedFunctions* не меняет содержимое переменной *e.Graph*, поэтому данный случай тоже сводится к потере быстродействия из-за лишнего копирования и сравнения.

В примере на Листинге 8 демонстрируется ошибочное поведение при пропущенном символе «^», так как переменная *e.Result* внутри вызова *Reduce* совпадает с одноименной переменной снаружи вызова. Но данную ошибку достаточно просто обнаружить в случае нескольких элементов внутри *e.Blocks*, так как аккумулятор внутри вызова функции *Reduce* будет совпадать с переменной e.Result только при обработке первого элемента.

Изучив эти и другие участки кода с данной проблемой, можно выдвинуть гипотезу о том, что пропущенный символ «^» чаще всего встречается в присваиваниях и последних предложениях внутри блоков и вложенных функций. Практически никогда данная ошибка не встречается в образцах в условиях, так как само логическое предназначение условий подразумевает использование уже определённых переменных.

3. Реализация

Первоначально было необходимо добавить сущности предупреждений типа *repeated* и *repeated-maybe*. Так как компилятор уже поддерживал активацию выдачи предупреждений с помощью аргументов командной строки, то было необходимо добавить два новых предупреждения к уже существующим. В модуль *Config.ref* была добавлена информация о существовании новых предупреждений (Листинг 9).

```
WarningForName {
  ...
  'repeated' = True repeated;
  'repeated-maybe' = True repeated-maybe;
  ...
}

$ENTRY Config-AllWarningIds {
    /* пусто */ = (screening nul-in-compound repeated
repeated-maybe)
}
```

Листинг 9. Изменения в Config.ref.

В модуль *Checker.ref* была добавлена информация, печатающаяся при выдаче предупреждений, содержащая указание повторной переменной (Листинг 10).

```
PrepareMessage {
  ...

  RepeatedVariable s.Mode e.Index
          = 'Repeated variable ' s.Mode '.' e.Index ' in
assignment/last sentence';

  RepeatedVariableMaybe s.Mode e.Index
      = 'Maybe repeated variable ' s.Mode '.' e.Index;
  ...
```

```
}
```

Листинг 10. Изменения функции PrepareMessage в Checker.ref.

Также в функцию, печатающую вспомогательную информацию компилятора была добавлена информация о новых предупреждениях.

В компиляторе Рефала-5λ за семантическую проверку отвечает функция *CheckProgram*, определённая в модуле *Checker.ref*. Так как вводимая новая проверка является семантической, то она должна располагаться в функции *CheckProgram*. Поэтому работа по проверке образцов на присутствие повторных переменных производилась в модуле *Checker.ref*. Новая проверка была добавлена в виде вызова отдельной функции внутри функции *CheckPattern*, осуществляющей проверку образца.

Для того, чтобы проверять образец согласно предложенной выше эвристике, необходимо в функцию проверки образца передавать аргумент, сигнализирующий о том, что данный образец находится в присваивании или в последнем предложении функции. Поэтому для различия последнего предложения от остальных была добавлена функция *CheckSentenceRec*, передающая в функцию *CheckSentence* специальное значение, которое потом передаётся в функцию проверки образца. Также в функцию проверки присваиваний и условий при проверке образца была добавлена передача информации о конструкции, в которой данный образец находится. Вышеперечисленные изменения представлены на Листинге 11.

```
CheckSentenceRec {
  (e.ScopeVars) t.LastSentence
     = <CheckSentence e.ScopeVars t.LastSentence LastSentence>
;
  (e.ScopeVars) t.Sentence e.Sentences
    = <CheckSentence e.ScopeVars t.Sentence NotLastSentence>
      <CheckSentenceRec (e.ScopeVars) e.Sentences>;
}
```

```
CheckSentence {
      e.ScopeVars ((e.Pattern) e.Assignments (e.Result)
(e.Blocks)) e.Flag

    = <CheckPattern (e.ScopeVars) (e.Pattern) e.Flag>
    : (e.ScopeVars^) e.PatternFunctionsAndErrors
    ...
}

DoCheckAssignments {
  ...
  ((s.ChainType (e.Result) (e.Blocks) (e.Pattern))
   e.Assignments)
  (e.ScopeVars) e.Errors

    = ...

    = <CheckPattern (e.ScopeVars) (e.Pattern) s.ChainType>
    : (e.ScopeVars^) e.PatternFunctionsAndErrors

    ...
}
```

Листинг 11. Передача контекста в функцию проверки образца.

После реализации передачи контекста в функцию *CheckPattern* в неё был добавлен вызов *CheckPattern-WarnRepeated* — новой функции, осуществляющей проверку связанных переменных в образце (Листинг 12). В неё передаются переменные области видимости, образец и результат работы функции *WarningIdByFlag*. Эта функция возвращает тип предупреждение, которое нужно выдавать в зависимости от контекста образца. Также, для гибкости изменения выдаваемых предупреждений в зависимости от контекста, данная функция может ничего не вернуть, а функция *CheckPattern-WarnRepeated* в этом случае не будет производить проверку. В

случае, если функция *WarningIdByFlag* вернула не пустой результат, то производится проверка с помощью функции *CheckPatter-WarnRepeatedRec*.

```
CheckPattern {
  (e.ScopeVars) (e.Pattern) e.Flag
          =  <CheckPattern-Flatten  (e.ScopeVars)  <FlatExpr
e.Pattern>>
      <CheckPattern-WarnRepeated
       (e.ScopeVars)
       <WarningIdByFlag e.Flag>
       (e.Pattern)
      >;
}

CheckPattern-WarnRepeated {
 (e.ScopeVars) s.WarningId (e.Pattern)
   = <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
      (e.Pattern)>;
 (e._) (e._) = /* не нужно проверять */;
}

CheckPattern-WarnRepeatedRec {
...
}

WarningIdByFlag {
  Assign = RepeatedVariable;
  Condition = RepeatedVariableMaybe;
  LastSentence = RepeatedVariable;
  NotLastSentence = RepeatedVariableMaybe;
  /* пусто */ = /* пусто */;
}
```

Листинг 12. Инициализация проверки.

При проверке образца недостаточно сравнивать все переменные в образце со всеми переменными в области видимости, так как в этом случае будет

большое количество ложных срабатываний. Это связано с тем, что в образцах с открытыми переменными повторные переменные используются для того, чтобы разрешить неоднозначность сопоставления. Поэтому необходимо анализировать однозначно сопоставляемые части образца.

Пусть имеется образец вида

$$P_1\ e.1\ P_2\ e.2\ \ldots\ P_n\ e.n\ P_{n+1}$$

где $P_i\,(i = 1..n+1)$ — части образца, не содержащие e-переменные (могут быть пустыми) на верхнем уровне и $n \in \mathbb{N}$. Тогда при $n = 1$ образец будет иметь вид

$$P_1\ e.1\ P_2\,.$$

В этом случае все части образца на верхнем уровне будут однозначно сопоставляемы и необходимо проверить все три части на наличие повторных переменных. Если $n > 1$, то весь интервал от $e.1$ до $e.n$ не будет однозначно сопоставляемым и его можно заменить на безымянную переменную $e.\_$ и проверять наличие связанных переменных только в частях P_1 и P_2. Если же образец не содержит e-переменных на верхнем уровне, то необходимо проверять его в исходном виде. Для изменения образцов с несколькими e-переменными на верхнем уровне была была реализована функция *ReducePattern* и вспомогательные функции — *FindLastE* для поиска последней e-переменной и *ReplaceInterval* для замены на безымянную переменную (Листинг 13).

```
ReducePattern {
  e.Pattern
    = <FindLastE e.Pattern> : t.LastE
    = <ReplaceInterval t.LastE e.Pattern>;
}

FindLastE {
  e.Pattern
```

```
    = <Reduce
        {
          t.LastElem (TkVariable t.Pos 'e' e.Index)
            = (TkVariable t.Pos 'e' e.Index);
          t.LastElem (e.Another) = t.LastElem;
        }
        ()
        e.Pattern
      >
}

ReplaceInterval {
  t.LastElem
  e.B (TkVariable t.Pos 'e' e.Index) e.M t.LastElem e.E
    = (e.B (TkVariable t.Pos 'e' '_') e.E);
}
```

Листинг 13. Функции для замены образца с несколькими e-переменными.

Алгоритм обхода образца в функции *CheckPattern-WarnRepeatedRec* является рекурсивным. В первую очередь проверяется верхний уровень. Если он содержит две и более e-переменные, то производится вышеописанная замена на безымянную переменную. Далее осуществляется последовательный проход по образцу слева направо. Если встречаются структурные или АТД скобки, то их внутреннее содержимое передаётся на проверку уровнем ниже. При встрече какого-либо символа (не переменной) или переменной с «^» они игнорируются. При встрече переменной без «^» для неё вызывается функция *CheckVariableRepeating*, проверяющая наличие переменной такого же типа с таким же индексом среди переменных области видимости и возвращающая предупреждение, если такая переменная нашлась (Листинг 14). Для безымянных переменных функция никогда не возвращает предупреждение. Исходный код фунции CheckPatter-WarnRepeatedRec представлен на Листинге 15.

```
CheckVariableRepeating {
```

```
  (e.ScopeVars) s.WarningId
  (TkVariable t.SrcPos s.Mode '_' e.Index) = ;

  (e.Vars-B (s.Mode e.Index) e.Vars-E) s.WarningId
  (TkVariable t.SrcPos s.Mode e.Index)
    = (Warning <WarningNameForId s.WarningId>
        t.SrcPos s.WarningId s.Mode e.Index);

  e._ = ;
}
```

Листинг 14. Функция CheckVariableRepeating.

```
CheckPattern-WarnRepeatedRec {
  (e.ScopeVars) s.WarningId
  (e.B (TkVariable t.Pos 'e' e.Index) e.M
       (TkVariable t.Pos2 'e' e.Index2) e.E)
    = <CheckPattern-WarnRepeatedRec
        (e.ScopeVars) s.WarningId
        <ReducePattern e.B (TkVariable t.Pos 'e' e.Index)
          e.M (TkVariable t.Pos2 'e' e.Index2) e.E>
      >;

  (e.ScopeVars) s.WarningId ((TkVariable e.Info) e.Tail)
    = <CheckVariableRepeating (e.ScopeVars) s.WarningId
        (TkVariable e.Info)>
      <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.Tail)>;

  (e.ScopeVars) s.WarningId ((TkNewVariable e.Info) e.Tail)
    = <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.Tail)>;

  (e.ScopeVars) s.WarningId ((Symbol e.Info) e.Tail)
    = <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.Tail)>;

  (e.ScopeVars) s.WarningId ((Brackets e.InBrackets) e.Tail)
```

```
    = <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.InBrackets)>
      <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.Tail)>;

  (e.ScopeVars) s.WarningId
  ((ADT-Brackets t.SrcPos (e.Name) e.InBrackets) e.Tail)
    = <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.InBrackets)>
      <CheckPattern-WarnRepeatedRec (e.ScopeVars) s.WarningId
        (e.Tail)>;

  (e.ScopeVars) s.WarningId () = /* предложения кончились */;
}
```

Листинг 15. Функция CheckPattern-WarnRepeatedRec.

4. Тестирование

Для проверки предупреждений были написаны автоматические тесты (Листинги 16-18).

```
* NO-WARNINGS


$ENTRY Go {
  = <TestAssignment ('a') 'a'>
    <TestAssignment2 ('a') 'a'>
    <TestAssignment3 ('a') 'a'>
    <TestAssignment4 ('a') 'a'>
    <TestAssignment5 ('a') 'a'>
}

TestAssignment {
  t.X s.Y = <SomeProcessing t.X s.Y> : t.X^ s.Y^ = /* пусто */
}

TestAssignment2 {
  t.X s.Y = <SomeProcessing t.X s.Y> : t.X^ e.B s.Y e.E
          = /* пусто */ ;
}

TestAssignment3 {
  t.X s.Y = <SomeProcessing t.X s.Y> : e.B t.X e.E s.Y^
          = /* пусто */ ;
}

TestAssignment4 {
  t.X s.Y = <SomeProcessing t.X s.Y> : e.B s.Y e.E
          = /* пусто */ ;
}

TestAssignment5 {
  t.X s.Y = <SomeProcessing t.X s.Y> : e.B t.X s.Y e.E
          = /* пусто */ ;
}

SomeProcessing {
  e.Any = e.Any
```

```
}
```

Листинг 16. Автотест repeated.ref.

```
* WARNING repeated

*$FROM LibraryEx
$EXTERN Map;

$ENTRY Go {
  = <TestAssignment ('a') 'a'>
    <TestAssignment2 ('a') 'a'>
    <TestAssignment3 ('a') 'a'>
    <TestLastSentence (1) 'a'>
}

TestAssignment {
  t.X s.Y = <SomeProcessing t.X s.Y> : t.X s.Y = /* пусто */ ;
}

TestAssignment2 {
  t.X s.Y = <SomeProcessing t.X s.Y> : t.X e.B s.Y e.E
          = /* пусто */ ;
}

TestAssignment3 {
  t.X s.Y = <SomeProcessing t.X s.Y> : e.B t.X e.E s.Y
          = /* пусто */ ;
}

TestLastSentence {
  e.List s.Symbol
    = <Map
        {
         (s.Num s.Symbol) = (s.Num 'default symbol');
         (s.Num s.Symbol)
         = (s.Num 'not default symbol (should be)') ;
        }
        e.List
      >
}

SomeProcessing {
  e.Any = e.Any
```

```
}
```

Листинг 17. Автотест repeated.WARNING.ref.

```
* WARNING repeated-maybe

*$FROM LibraryEx
$EXTERN Map;

$ENTRY Go {
  = <TestCondition 'a' 'a'>
    <TestNotLastSentence (1) 'a'>
}

TestCondition {
  t.X s.Y, <SomeProcessing t.X s.Y> : t.X s.Y = /* пусто */ ;
}

TestNotLastSentence {
  e.List s.Symbol
    = <Map
        {
          (s.Num s.Symbol) = 'not last sentence';
          (s.Num s.Symbol^) = 'last sentence';
        }
        e.List
      >
}

SomeProcessing {
  e.Any = e.Any
}
```

Листинг 18. Автотест repeated-maybe.WARNING.ref.

На Листинге 16 представлен автотест *repeated.ref*, проверяющий, что предупреждение *repeated* не выдаётся в различных присваиваниях, в которых образцы не удовлетворяют вышеописанной эвристике.

На Листингах 17 и 18 представлены автотесты *repeated.WARNING.ref* и *repeated-maybe.WARNING.ref*, которые проверяют случаи, когда должны выдаваться предупреждения типа *repeated* и *repeated-maybe* соответственно.

Оба файла начинаются с комментария вида `* WARNING <warning-type>`, где `warning-type` — тип проверяемого предупреждения. Автотест *repeated.WARNING.ref* проверяет, что предупреждение типа *repeated* возникает при повторной переменной в образце последнего предложения и в образцах в присваиваниях, которые удовлетворяют эвристике. Автотест *repeated-maybe.WARNING.ref* проверяет, что предупреждение типа *repeated-maybe* возникает при повторных переменных в образцах условий и в образцах не последних предложений. На написанных тестовых программах выдача реализованных предупреждений отработала корректно.

Эффективность выдаваемых предупреждений была оценена при помощи компиляции исходного кода компилятора Рефала-5λ. Предупреждение считалось эффективным, если оно указывало на часть кода, в которой в связи с отсутствующим символом «^» возникала ошибка или потеря производительности. Результаты предоставлены в Таблице 1. Одно из предупреждений типа *repeated* указало на полноценную ошибку в коде. При этом практически все предупреждения типа *repeated-maybe* являлись ссылками на корректный код, только иногда указывая на потерю производительности.

Тип предупреждения	Кол-во предупреждений	Кол-во верных срабатываний	Эффективность, %
repeated	10	8	80
repeated-maybe	30	4	13.3

Таблица 1. Эффективность предупреждений.

Также выдача новых предупреждений была произведена при компиляции суперкомпилятора MSCP-A [5]. В данном случае эффективность выдаваемых предупреждений была значительно ниже. Вероятно это связано с тем, что данный суперкомпилятор написан на классическом Рефале-5, в котором отсутствуют некоторые синтаксические конструкции, добавленные в Рефале-5λ (в том числе и сам символ «^»). Было выдано 2 предупреждения типа *repeated* и

53 предупреждения типа *repeated-maybe*, из которых только пара указывала на потерю производительности.

Также было произведено тестирование времени компиляции программ с выдачей новых предупреждений. Результаты тестирования представлены в Таблице 2.

Программа	Время компиляции без предупреждений, с	Время компиляции с предупреждениями, с	Прирост времени компиляции, %
Компилятор Рефала-5λ	38.330	38.639	0.81
Суперкомпилятор MSCP-A	22.657	22.855	0.87

Таблица 2. Результаты тестирования времени компиляции.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы была реализована выдача предупреждений о связанных переменных в образцах. Было добавлено 2 вида предупреждений компилятора — *repeated* и *repeated-maybe*. Первый тип предупреждений, выдающихся при повторных переменных в последних предложениях функции и присваиваниях, оказался достаточно эффективным — выдаваемые предупреждения часто сигнализируют о реальных ошибках и потерях производительности. Подавляющее большинство выдаваемых предупреждений второго типа являются ложными сигналами об ошибке, что является логичным следствием частого использования повторяющихся переменных в условиях и предложениях, не являющимися последними, при программировании на классическом Рефале-5 и Рефале-5λ.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий компилятора Рефал-5λ: [Электронный ресурс]. URL: https://github.com/bmstu-iu9/refal-5-lambda (дата обращения: 06.02.2021).
2. Рефал 5. Руководство по программированию и справочник: [Электронный ресурс]. URL: http://refal.ru/rf5_frm.htm (дата обращения: 06.02.2021).
3. Рефал-5λ. Приложение B. Краткий справочник: [Электронный ресурс]. URL: https://bmstu-iu9.github.io/refal-5-lambda/B-reference.html (дата обращения: 06.02.2021).
4. Рефал-5λ. Библиотека LibraryEx: [Электронный ресурс]. URL: https://mazdaywik.github.io/refal-5-framework/LibraryEx.html (дата обращения: 06.02.2021)
5. Модельный суперкомпилятор MSCP-A: [Электронный ресурс]. URL: http://refal.botik.ru/mscp/ (дата обращения: 28.02.2021).