Федеральное государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «ИНФОРМАТИКА И СИСТЕМЫ УПРАВЛЕНИЯ»
КАФЕДРА «ТЕОРЕТИЧЕСКАЯ ИНФОРМАТИКА И КОМПЬЮТЕРНЫЕ ТЕХНОЛОГИИ»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОМУ ПРОЕКТУ

Добавление в компилятор Простого Рефала условий из Рефала-5.

Руководитель курсового проекта _____ (А. В. Коновалов)
(подпись, дата)

Исполнитель курсового проекта,
студент группы ИУ9-71 _____ (Д. Р. Габбасов)
(подпись, дата)

Москва, 2017

Содержание.

ВВЕДЕНИЕ.

На текущий момент синтаксис и частично семантика Простого Рефала представляет собой подмножество Базисного Рефала диалекта РЕФАЛ-5, пополненное, однако, функциями высших порядков (косвенный вызов функции, указатели на глобальные функции, вложенные безымянные функции).

Программировать на подмножестве Базисного Рефала бывает затруднительно, поскольку для каждой функции — отдельной смысловой единицы, приходится писать порой несколько отдельных вспомогательных функций. В результате код становится громоздким.

РЕФАЛ-5 же поддерживает два синтаксических расширения Базисного Рефала: условия и блоки. [1] Блоки — те же безымянные функции, которые, однако, можно только вызвать одновременно с созданием. Предложение с блоком РЕФАЛа-5 эквивалентно предложению Простого Рефала, результатная часть которого состоит из вызова функции Fetch со вложенной функцией. Поэтому синтаксис блоков в Простом Рефале избыточен.

В данном курсовом в компилятор Простого Рефала будет добавлена конструкция, идентичная условиям из РЕФАЛа-5, а также конструкция безусловных присваиваний.

1 ОБЗОР ЯЗЫКА И КОМПИЛЯТОРА ПРОСТОГО РЕФАЛА.

РЕФАЛ — РЕкурсивный Функциональный АЛгоритмический язык, язык функционального программирования, ориентированный на символьные вычисления, обработку и преобразование текстов. [2] У данного языка есть много несовместимых диалектов, Простой Рефал — один из них.

Особенности языка, отличающие его от других диалектов РЕФАЛа:

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

Файл исходного текста на Рефале состоит из последовательности программных элементов — определений функций и ссылок на функции, определённые в других единицах трансляции.

Программа пишется в свободном формате, то есть переводы строк приравнены к обычным пробельным символам, там, где допустим пробельный символ (пробел или табуляция), допустима вставка перевода строки. Пробельные символы можно вставлять между двумя любыми лексемами языка. Пробелы обязательны в том случае, когда две лексемы, записываемые подряд, могут интерпретироваться как одна сплошная лексема (например, два числа, записанные слитно, будут интерпретироваться как одно число и т.д.). Пробелы недопустимы внутри идентификаторов, чисел, директив. Пробелы внутри цепочек литер интерпретируются как образы литер со значением «пробел».

Глобальная регулярная функция (та, которая не вложенная) представляет собой именованный блок, который может предваряться ключевым словом $ENTRY. Блок, в свою очередь, содержит последовательность нуля или более предложений (в отличие от РЕФАЛа-5, где в любой функции должно быть как минимум одно предложение).

Предложение состоит из двух частей: образцовой части и результатной части, разделённых знаком «=». Каждое предложение заканчивается точкой с запятой (в отличие от РЕФАЛа-5, где после последнего предложения точка с запятой не обязательно).

Образцовая часть (синонимы: левая часть, образцовое выражение, образец) состоит из последовательности образцовых термов, среди которых могут быть литералы атомарных термов, скобочные термы двух видов (структурные круглые скобки и именованные

квадратные скобки, т. н. абстрактные типы данных) и переменные, некоторые из которых могут быть помечены знаком переопределения ^.

Результатная часть (синонимы: правая часть, результатное выражение, результат) состоит из последовательности результатных термов, среди которых могут быть литералы атомарных термов, пассивные скобочные термы тех же двух видов (круглые и абстрактные скобки), скобки активации и блоки — литералы вложенных функций. Одно из отличий от РЕФАЛа-5 заключается в том, что после открывающей скобки активации синтаксически не обязано находиться имя функции, проверка того, что за ним находится экземпляр функции (указатель на функцию или замыкание) осуществляется во время выполнения программы.

Переменные, как и в РЕФАЛе-5, могут быть трёх видов: s-переменные — могут сопоставляться только с атомами, t-переменные — могут сопоставляться с любым термом и e-переменные — могут сопоставляться с любым объектным выражением. Область видимости переменной, объявленной в некотором образцовом выражении — образцовое и результатное выражение данного предложения, а также все вложенные функции, определённые внутри результатного выражения. Однако, внутри вложенных функций переменные могут скрывать другие одноимённые переменные из внешней области видимости, если они помечены знаком переопределения ^. Переменные записываются как вид.индекс, где вид может быть одной из букв s, t или e, индекс — 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. В одной области видимости не может быть двух переменных с одинаковым индексом, но разного вида. [3]

Общие черты языков Простой Рефал и РЕФАЛ-5 (подмножество Базисного РЕФАЛа):

- Функции внешне имеют сходный синтаксис.
- Функции оперируют объектными выражениями: принимают в качестве аргумента одно объектное выражение и возвращают его в качестве результата.
- Семантика выполнения программы тоже определяется в терминах абстрактной Рефал-машины.
- Идентичная семантика сопоставления с образцом. Структура результатного выражения незначительно отличается.

Чего нет в РЕФАЛе-5, но есть в Простом Рефале:

- Функции как атомы и как объекты первого класса.
- Абстрактные типы данных.
- Вложенные функции.

Что есть в РЕФАЛе-5, но нет в Простом Рефале:

- Условия. Их нет. Данная работа ставит своей целью это изменить.
- Блоки. Их нет, и они не нужны. Функциональность блоков реализуется при помощи вложенных функций и библиотечной функции Fetch.
- Копилка. Есть близкое по функциональности средство — статические ящики. Несложно на основе статических ящиков реализовать библиотеку, реализующую возможности копилки.
- Метавычисления. Вместо метафункции Mu можно использовать косвенный вызов функций функциональность, связанная с метакодом, пока не реализована в библиотеке Простого Рефала.

Компилятор Простого Рефала самоприменимый, то есть написан на Простом Рефале. Он осуществляет генерацию кода на C++.

Компиляция многопроходная, имеет следующий проходы:

1) Загрузка исходного текста программы.
2) Лексический анализ.
3) Синтаксический анализ.
4) Проверка контекстных зависимостей.
5) Редуктор до подмножества (обессахариватель).
6) Генерация высокоуровневого RASL’а.
7) Генерация низкоуровневого RASL’а.
8) Генерация целевого кода на C++.

2 ОБЗОР ПОСТАВЛЕННОЙ ЗАДАЧИ.

2.1 Условия, синтаксис и семантика.

Условие — мощная синтаксическая конструкция, которую в большинстве случаев нельзя лаконично эмулировать с помощью имеющихся синтаксических средств Простого Рефала. Синтаксис условий следующий (см. Листинг 1).

```
Функция {
  …
  Образец, Результат1 : Образец1, Результат2 : Образец2, … РезультатN : ОбразецN =
Результат;
  …
}
```

Листинг 1. Синтаксис условий.

Здесь на каждый из образцов (кроме ОбразецN) наложено условие.

- Образец имеет условие , Результат1 : Образец1, Результат2 : Образец2, … РезультатN : ОбразецN.
- Образец1 имеет условие , Результат2 : Образец2, … РезультатN : ОбразецN.
- …
- ОбразецN−1 имеет условие , … РезультатN : ОбразецN.

Условие выполняется, когда значение, сформированное результатом (после знака , и перед знаком :), успешно сопоставилось с последующим образцом (после знака : и перед знаком , или =). В противном случае не выполняется.

Образец с условием сопоставлен успешно, если удаётся найти такую подстановку переменных образца в некоторые значения, что образец совпадает с сопоставляемым значением (аргументом функции, если он первый в предложении или результатом вычисления результатного выражения перед ним, если это образец условия) и условие, наложенное на образец, выполняется.

Если образец является частью условия, то в него могут входить переменные, которые уже получили свои значения при сопоставлении с другими образцами слева от него. Такие переменные называются связанными.

Как и в случае образца без условий, для образца с условиями может существовать несколько подстановок, которые превращают образец в требуемое значение. Как обычно, среди них выбирается та, при которой текстуально самая левая переменная принимает кратчайшую длину в термах. Если это не разрешает неоднозначности, то анализируется следующая переменная и т. д.

В рамках настоящей задачи предлагается пополнить Простой Рефал следующим синтаксисом (используется нотация Вирта) (см. Листинг 2).

```
Sentence = Pattern { Condition | Assignment } "=" Result.
Condition = "," Result ":" ConditionalPattern.
Assignment = "=" Result ":" ConditionalPattern.
```

Листинг 2.Пополнение синтаксиса условиями.

Здесь в левой части предложения за образцом может следовать ноль или более условий и присваиваний. О присваиваниях ниже. В образцовой части условий и присваиваний могут присутствовать переопределяемые переменные с тем же смыслом, что и во вложенных функциях.

2.1 Присваивания, мотивация, синтаксис и семантика.

Практика программирования на РЕФАЛе-5 показывает, что условия часто удобно использовать в роли присваиваний: в предложение добавляется условие, которое всегда выполняется, результатная часть которого вычисляет некоторое новое значение, а образцовая связывает его с одной или несколькими переменными. После чего эти переменные можно использовать в результатной части самого предложения.

В рамках Базисного Рефала в таком случае приходится писать промежуточную функцию (в которую передаётся контекст — переменные, сопоставленные в образце, которые нужны в результате) и то вычисление, которое находится в результатной части условия.

В рамках Простого Рефала с функциями высших порядков в этом случае в результатной части предложения помещается вызов функции Fetch, принимающий вычисляемое выражение и вложенную функцию из одного предложения, образец которой соответствовал бы образцу условия (по сути та же промежуточная функция, но только с гораздо более удобным синтаксисом).

Недостаток обоих подходов — громоздкость. Но есть и недостаток у условия в РЕФАЛе-5. Дело в том, что в силу их семантики, при возможной неудаче сопоставления, аргумент функции должен быть передан на анализ в левую часть следующего предложения. Таким образом, при вычислении результатных выражений в условии, переменные, передаваемые в них, копируются (а это может быть дорого на списковой реализации). Но часто так бывает, что некоторое крупное значение передаётся в условие-присваивание для преобразования и исходное значение в правой части предложения не используется. Компилятор не способен догадаться, что условие выполняется всегда и вместо копирования это значение можно перемещать за постоянное время.

Поэтому в предлагаемом синтаксисе наряду с классическими условиями предлагается конструкция «присваивание» имеющая вид = Результат : Образец. Её отличие от условия в том, что требуется, чтобы Образец (возможно, с наложенными на него условиями) всегда сопоставлялся успешно (то есть, при неуспешном сопоставлении будет происходить

аварийный останов программы).

Поскольку, в отличие от условий, присваивания никогда не производят отката, в их результатное выражение переменные можно не копировать, а перемещать. Это главная мотивация их внедрения в язык.

Фактически, присваивания являются синтаксическим сахаром. Запись в Листинге 3

```
Образец = Результат1 : Образец1 = Результат
```

Листинг 3. Пример присваиваний.

эквивалентна записи в Листинге 4.

```
Образец =
    <
      {
        Образец1 = Результат;
      }
      Результат1
    >;
```

Листинг 4. Эквивалентная запись присваиваний.

или, если использовать библиотечную функцию Fetch (см Листинг 5).

```
Образец =
    <Fetch
      Результат1
      {
        Образец1 = Результат;
      }
    >;
```

Листинг 5. Эквивалентная запись присваиваний с использованием функции Fetch.

Соответственно, в процессе компиляции их можно (и даже нужно) будет преобразовывать в соответствующие безымянные функции. Они не намного усложнят синтаксический анализ (по сравнению с синтаксическим анализом, усложнённым одними лишь условиями), их довольно просто компилировать, а польза от них будет огромная (уже в коде Простого Рефала довольно часто встречаются вызовы Fetch с функцией из одного предложения).

В качестве мотивации встраивания присваиваний можно принять тот факт, что даже в коде компилятора Простого Рефала очень часто встречаются конструкции, представленные в Листинге 5. И целесообразно заменить их более понятной конструкцией из Листинга 3.

Ещё одна мотивация – борьба с вложенными функциями. Например, конструкцию, представленную в Листинге 6, можно заменить на конструкцию, представленную в Листинге 7. При этом можно дать имена результатам промежуточных функций, что улучшит документированность кода.

```
Образец =
    <F …
      <G …
        <H … >
      >
    >;
```

Листинг 6. Пример вложенности.

```
Образец =
    <H … > : e.ResultOfH =
    <G … > : e.ResultOfG =
    <F … > : e.ResultOfF;
```

Листинг 7. Пример раскрытия вложенности с помощью присваиваний.

3 РЕАЛИЗАЦИЯ.

3.1 Расширение синтаксиса. Модификация синтаксического анализатора.

Первым делом необходимо модифицировать лексический анализатор (проход 2), добавив в него лексему для двоеточия - #TkColon – в раздел пунктуации. Таким образом, на этапе синтаксического анализа мы будем работать уже с соответствующей лексемой в предложениях.

Следующий шаг – модификация синтаксического анализа под расширенную грамматику, учитывающую синтаксис условий и присваиваний.

Для этого необходимо изменить тэг для предложения в синтаксическом дереве и доопределить новые конструкции (см. Листинг 8 и Листинг 9).

```
t.Sentence ::= ((e.Pattern) (e.Result))
```

Листинг 8. Тэг предложения до изменения.

```
t.Sentence ::= ((e.Pattern) t.Condition* (e.Result))
t.Condition ::= (s.Type (e.Result) (e.Pattern))
s.Type ::= #Cond | #Assign
```

Листинг 9. Измененный тег предложения.

При этом под #Cond подразумевается символ “ , ”, а под #Assign символ “ = “.

В качестве реализации поддержки данных конструкций необходимо модифицировать функцию ParseSentence таким образом, чтобы она могла правильно обрабатывать предложения, содержащие условия и присваивания.

Для этого есть несколько способов.

Первый: построить автомат, который будет по очереди обрабатывать образец и результат, где пары разделены знаком “ = “ или “ , “ , а между парами стоит “ : “ (при этом известно, что заканчивается предложение символом “ ; “). Состояния автомата будет, соответственно, два: сканирование образца и сканирование результата. После этого собрать получившиеся пары в предложение, удовлетворяющее Листингу 9.

Второй способ похож на первый: использовать существующий алгоритм для обработки пары образец-результат рекурсивным образом, а потом сформировать правильное предложение.

Третий способ: после обработки первого образца вести обработку следующим образом: ждать после образца символ “ = “ или “ , “, обрабатывать идущий за ними результат. Далее, в зависимости от символа, идущего после результата, либо заканчивать обработку,

интерпретируя последний результат как тот самый результат, что не участвует в условиях и присваиваниях, либо обрабатывать идущий после этого результата образец, который является частью присваивания или условия, связанного с данным результатом. После сканирования этого условия дополнить дерево соответствующей записью условия или присваивания.

Был выбран третий вариант, так как он показался наиболее удобным для обработки «на лету». Получившиеся изменения функции ParseSentence представлены в Листинге 10.

```
ParseSentence {
  t.ErrorList (e.Sentences) e.Tokens =
    <Fetch
      <ParsePattern
        t.ErrorList ( /* мультискобки */ )
        ( /* просканировано */ ) e.Tokens
      >
      <Seq
        {
          t.ErrorList^ (e.Pattern) (#TkReplace s.LnNum) e.Tokens^ =
            t.ErrorList (e.Pattern) e.Tokens;

          t.ErrorList^ (e.Pattern) t.NextResultToken e.Tokens^ =
            <EL-AddUnexpected
              t.ErrorList t.NextResultToken '='
            >
            (e.Pattern) t.NextResultToken e.Tokens;
        }
        {
          t.ErrorList^ (e.Pattern) e.Tokens^ =
            (e.Pattern)
            <ParseAssignmentPart
              t.ErrorList e.Tokens
            >;
        }
        {
          (e.Pattern) (e.Result) e.Assignments t.ErrorList^ e.Tokens^ =
            t.ErrorList
            (e.Sentences ((e.Pattern) e.Assignments (e.Result))) e.Tokens;
        }
      >
    >;
}
```

Листинг 10. Измененная функция ParseSentence.

Можно заметить, что в Листинге 10 используется функция ParseAssignmentPart, которая осуществляет обработку предложения, начиная с той части, где могут быть присваивания или условия. На Листинге 11 представлена её реализация для присваиваний (для условия реализация аналогична).

```
ParseAssignmentPart {
  t.ErrorList (e.Assignments) e.Tokens =
    <Fetch
      <ParseResult
        t.ErrorList ( /* мультискобки */ )
        ( /* просканировано */ ) e.Tokens
      >
      {
        t.ErrorList^ (e.Result) (#TkSemicolon s.LnNum) e.Tokens^ =
          (e.Result) (e.Assignments) t.ErrorList e.Tokens;
        t.ErrorList^(e.Result) (#TkColon s.LnNum) e.Tokens^ =
          <Fetch
            <ParsePattern
              t.ErrorList ( /* мультискобки */ )
              ( /* просканировано */ ) e.Tokens
            >
            <Seq
              {
                t.ErrorList^ (e.Pattern) e.Tokens^ =
                  (e.Assignments (#Assign (e.Result) (e.Pattern)))
                  t.ErrorList e.Tokens;
              }
              {
                (e.Assignments^)
                t.ErrorList^ (#TkReplace s.LnNum^) e.Tokens^ =
                  <ParseAssignmentPart
                    t.ErrorList (e.Assignments) e.Tokens
                  >;
              }
            >
          >;
      }
    >;
}
```

Листинг 11. Реализация функции ParseAssignmentPart.

3.2 Модификация проверки контекстно-зависимых ошибок.

Следующий проход компилятора после синтаксического анализа – проверка контекстных зависимостей.

В него так же надо внести модификации, так как при обработке условий и присваиваний появляются новые контекстные зависимости.

Как и в случае с модификацией синтаксического анализатора, необходимо изменить функцию CheckSentence, чтобы она обрабатывала ту часть предложения, где содержатся условия и присваивания.

Идея модификации следующая: вначале обрабатывается первый образец, формируется начальное множество контекстов. Затем обрабатываются присваивания следующим образом: при обработке результата проверяется контекст, а при проверке образца ещё и пополняется множество контекстов. Так до тех пор, пока не закончатся все присваивания.

Затем проверяются ошибки контекста в последнем результате.

Код модифицированной функции CheckSentence и связанной с ней функции CheckAssignments приветен в Листинге 12.

```
CheckSentence {
  e.ScopeVars ((e.Pattern) e.Assignments (e.Result)) =
    <Fetch
      <CheckPattern (e.ScopeVars) <FlatExpr e.Pattern>>
      <Seq
        {
          (e.ScopeVars^) e.PatternFunctionsAndErrors =
          (e.PatternFunctionsAndErrors)
          <CheckAssignments (e.Assignments) (e.ScopeVars)>;
        }
        {
          (e.PatternFunctionsAndErrors^) (e.ScopeVars^)
          e.AssignmentsFunctionsAndErrors =
            (e.ScopeVars) e.PatternFunctionsAndErrors
            e.AssignmentsFunctionsAndErrors;
        }
        {
          (e.ScopeVars^) e.PatternFunctionsAndErrors^ =
            e.PatternFunctionsAndErrors
            <CheckResult (e.ScopeVars) <FlatExpr e.Result>>;
        }
      >
    >;
}

CheckAssignments {
  () (e.ScopeVars) e.Errors = (e.ScopeVars) e.Errors;

  ((#Assign (e.Result) (e.Pattern)) e.Assignments) (e.ScopeVars)
  e.Errors =
    <Fetch
      <CheckResult (e.ScopeVars) <FlatExpr e.Result>>
      <Seq
        {
          e.Errors^ =
            (e.Errors)
            <CheckPattern (e.ScopeVars) <FlatExpr e.Pattern>>;
        }
        {
          (e.Errors^) (e.ScopeVars^) e.PatternFunctionsAndErrors =
            <CheckAssignments (e.Assignments) (e.ScopeVars) e.Errors
            e.PatternFunctionsAndErrors>;
        }
      >
    >;
}
```

Листинг 12. Модификация проверки контекстно-зависимых ошибок

3.3 Модификация обессахаривателя: удаление переопределений переменных.

.Как известно, в теле функций в предложениях образцах можно переопределять переменный, добавляя к ним символ “ ^ “. Таким образом показывается, что данная переменная не связана с предыдущим её определением.

Так как в присваиваниях фигурируют образцы, там могут быть новые переопределения, которые надо учитывать и правильно обрабатывать. Поэтому необходимо модифицировать функцию EnumerateVars-Sentences в файле Desugaring.sref специальным образом.

Идея модификации похожа на ту идею, что использовалась выше для модификации проверки контекстных зависимостей. Вначале идет обработка первого образца. Затем, пока есть присваивания, идет поочередная обработка результатов и образцов из присваиваний с сохранением обработанных присваиваний. При этом при каждом новом образце глубина переопределений увеличивается на единицу. И в конце обрабатывается результат, не входящий в присваивания, снова формируется правильное предложение из образца, присваиваний и результата с удаленными переопределениями переменных.

Код модификации обессахаривателя представлен в Листинге 13 и Листинге 14.

```
EnumerateVars-Sentences {
  s.Depth (e.ScopeVars) #Sentences e.Sentences =
    #Sentences
    <Map
      {
        ((e.Pattern) e.Assigns (e.Result)) =
          <Fetch
            <EnumerateVars-Expr s.Depth (e.ScopeVars) e.Pattern>
            <Seq
              {
                (e.ScopeVars^) e.Pattern^ =
                  <EnumerateVars-Assignments () s.Depth (e.ScopeVars) e.Assigns>
                  (e.Pattern);
              }
              {
                (e.ScopeVars^) s.Depth^ e.Assigns^ (e.Pattern^) =
                  <Fetch
                    <EnumerateVars-Expr s.Depth (e.ScopeVars) e.Result>
                    {
                      (e.ScopeVars^) e.Result^ =
                        ((e.Pattern) e.Assigns (e.Result));
                    }
                  >;
              }
            >
          >;
      }
      e.Sentences
    >;

  s.Depth (e.ScopeVars) #NativeBody t.SrcPos e.Code =
    #NativeBody t.SrcPos e.Code;
```

Листинг 13. Код модифицированной функции EnumerateVars-Sentences.

```
EnumerateVars-Assignments {
  (e.RAssignments) s.Depth (e.ScopeVars) = (e.ScopeVars) s.Depth
  e.RAssignments ;

  (e.RAssignments) s.Depth (e.ScopeVars) (#Assign (e.Result)
  (e.Pattern)) e.Assignments =
    <Fetch
      <EnumerateVars-Expr s.Depth (e.ScopeVars) e.Result>
      <Inc s.Depth>
      <Seq
        {
          (e.ScopeVars^) e.Result^ s.Depth^  =
            <EnumerateVars-Expr s.Depth (e.ScopeVars) e.Pattern>
            (e.Result);
        }
        {
          (e.ScopeVars^) e.Pattern^ (e.Result^) =
            <EnumerateVars-Assignments
              (e.RAssignments (#Assign (e.Result) (e.Pattern)))
              s.Depth (e.ScopeVars) e.Assignments
            >;
        }
      >
    >;
}
```

Листинг 14. Код вспомогательной функции EnumerateVars-Assignments.

3.4 Модификация обессахаривателя: удаление присваиваний.

Как было описано выше, присваивания целесообразно заменять на вызов соответствующей безымянной функции, как описано в Листинге 4.

Целесообразно сделать это в процессе работы обессахаривателя дополнительным проходом Pass-RemoveAssigns.

В данный момент это проход осуществляется перед проходом Pass-NameNestedFunc, который именует вложенные безымянные функции. Это может приводить к проблемам с отладкой, так как будет не совсем ясно при каком сопоставлении произошла ошибка. В дальнейшем планируется реализовать дополнительный проход для именования условий и присваиваний для того, чтобы таких проблем не было.

Идея прохода следующая: вначале идем по всем предложениям всех функций. Для каждого предложения вызываем функцию Remove-Assignments-Sentence, которая вызывается рекурсивно до тех пор, пока в предложении есть присваивания. А обработку присваивания она осуществляет следующим образом: для первого присваивания формируется предложение, где образцом служит тот же самый образец, что подавался на вход, а в качестве результата будет вызов замыкания с одним предложением: сопоставлением образца присваивания с результатам входного предложения с оставшимися присваиваниями. Аргументом же замыкания будет результат присваивания. Таким образом

цепочка присваиваний будет преобразована в вызовы функций.

Код прохода Pass-RemoveAssigns и вспомогательных функция представлен в Листинге 15.

```
Pass-RemoveAssigns {
  e.AST =
    <Map
      {
        (#Function s.ScopeClass (e.Name) e.Body) =
          (#Function
            s.ScopeClass (e.Name)
            <RemoveAssigns-Sentences e.Body>
          );

        t.OtherProgramElement = t.OtherProgramElement;
      }
      e.AST
    >;
}

RemoveAssigns-Sentences {
  #Sentences e.Sentences =
    #Sentences
    <Map
      Remove-Assignments-Sentence
      e.Sentences
    >;

  #NativeBody t.SrcPos e.Code =
    #NativeBody t.SrcPos e.Code;
}

Remove-Assignments-Sentence {
  ((e.Pattern)(e.Result)) = ((e.Pattern)(e.Result));

  ((e.Pattern) (#Assign (e.AssignResult)(e.AssignPattern))
  e.Assignments (e.Result)) =
    (
      (e.Pattern)
      (
        (#CallBrackets
          (#Closure #Sentences
            <Remove-Assignments-Sentence
              ((e.AssignPattern)e.Assignments(e.Result))
            >
          )
          e.AssignResult
        )
      )
    );
}
```

Листинг 15.Реализация удаления присваиваний.

3.5 Способы реализации условий.

В отличии от присваиваний, условия не удастся реализовать как синтаксический сахар. Поэтому здесь нужен другой подход. Опишем два возможных подхода:

1) Рекурсивная реализация.

В этом случае для выполнения результатной части условия внутри выполняющейся функции рекурсивно вызывается Рефал-машина. При этом может быть удобно использовать специальную функцию, в которую передается условие и которая завершается вместе со внешней функции при успешном сопоставлении, или передает управление на следующее предложение иначе.

Достоинства такого подхода:

- Простота реализации.

Недостатки:

- Рекурсивные вызовы внутри условий будут расходовать системный стек, размер которого ограничен несколькими мегабайтами. А стиль функционального программирования предполагает активное использование рекурсии.

2) Итеративная реализация.

При переходе к условию в текущее поле зрения (скорее всего, куда-то внутрь вызова функции — первичного активного подвыражения) добавляется результатное выражение условия, а также информация о текущем состоянии вычисления функции (номер предложения, номер условия в предложении, значения переменных, связанных в образце), выполнение передаётся на результатное выражение. Когда оно завершается, управление возвращается к приостановленной функции, она восстанавливает своё состояние и анализирует вычисленное значение результатной части условия.

Достоинства такого подхода:

- Глубина рекурсии ограничена не системным стеком, а объёмом кучи.

Недостатки:

- Сложность реализации.

На данный момент условия ещё не реализованы. Но для их реализации подготовлены правки в синтаксический анализатор, семантический анализатор и обессахариватель, аналогичные правкам для присваиваний. В дальнейшем планируется добавление поддержки условий.

4 ТЕСТИРОВАНИЕ.

Тестирование производилось с помощью встроенных автотестов, встроенных в компилятор. При этом они были дополнены тестами, проверяющими производительность присваиваний, представленными в Листингах 16-18.

```
$ENTRY Go { = : =  ; }
```

Листинг 16. Тест 3-assign-dry-run.

```
$ENTRY Go { e.X e.Z = e.X : e.X^ = e.Z ; }
```

Листинг 17. Тест 4-assign-semantic.

```
$ENTRY Go { =
  <Num
    (#Binary
      (#Value 's')
      (#Unary
        (#Value 't')
      )
    )
  >;
}

Num {
  t.Tree
    = <NumRec 1 t.Tree> : s.Num t.Tree^
    = t.Tree;
}

NumRec {
  s.Num (#Value s.Value)
    = <I s.Num> (#Value s.Num s.Value);

  s.Num (#Unary t.Tree)
    = <NumRec s.Num t.Tree> : s.Num^ t.Tree^
    = s.Num (#Unary t.Tree);

  s.Num (#Binary t.Left t.Right)
    = <NumRec s.Num t.Left> : s.Num^ t.Left^
    = <NumRec s.Num t.Right> : s.Num^ t.Right^
    = s.Num (#Binary t.Left t.Right);
}

I {
  e.X = e.X;
}
```

Листинг 17. Тест 5-assign-hard.

ЗАКЛЮЧЕНИЕ.

В данной работе была полностью реализованная конструкция присваиваний и реализованы основные шаги для реализации условий.

При этом в дальнейшем планируется добавить именование условий и присваиваний для более удобной отладки, поработать над производительностью, чистотой кода и добавит поддержку условий.

Список использованной литературы.

1) Турчин В. Ф. Пользовательская документация для языка РЕФАЛ-5 [Электронный ресурс]: Содружество «РЕФАЛ/Суперкомпиляция» .- Режим доступа: http://www.refal.net/rf5_frm.htm .
2) Турчин В. Ф. Алгоритмический язык рекурсивных функций (РЕФАЛ). — М.: ИПМ АН СССР, 1968.
3) Коновалов А.В. Пользовательская документация для языка Простой Рефал [Электронный ресурс]: GitHub .- Режим доступа: https://github.com/bmstu-iu9/simple-refal .