Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«**Московский Государственный Технический Университет
имени Н. Э. Баумана**»

Курсовой проект.

по дисциплине: «Конструирование компиляторов»

на тему:

«Условия РЕФАЛа-5 в Простом Рефале»

Выполнила:
студентка 4 курса,
группы ИУ 9-72
Козлова А.А.

Руководитель:
Коновалов А.В.

Москва 2018

Оглавление

Введение

Преподаватель МГТУ им. Баумана, Коновалов Александр Владимирович создал язык программирования Простой Рефал.

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

При программировании на подмножестве Базисного Рефала бывают ситуации, когда для каждой функции — отдельной смысловой единицы, приходится писать несколько вспомогательных функций. Функции высших порядков, особенно вложенные функции, помогают облегчить задачу, но в ряде случаев код всё равно остаётся громоздким.

Чтобы сделать программирование на Простом Рефале более удобным, следует добавить в компилятор Простого Рефала на стадии анализа и синтеза эффективную поддержку условий .

1. Обзор предметной области.

1.1. Описание языка Простой Рефал.

Файл исходного текста на Простом Рефале состоит из последовательности программных элементов. Программные элементы делятся на две категории:

1. определения функций
2. ссылки на функции, объявленные в других единицах трансляции — объявления внешних функций

Область видимости может быть либо локальной — только текущая единица трансляции, либо глобальной — вся программа. В одной области видимости не может быть определено двух разных функций с одинаковыми именами.

Объявления внешних функций добавляют в локальную область видимости имена entry-функций, объявленных в других единицах трансляции.

Синтаксис функции представлен в Листинге 1.

```
1. FunctionDefinition = [ "$ENTRY" ] NAME Block.
2. Block = "{" { Sentence } "}".
3. Sentence = Pattern { Assign } "=" BlockedResult.
4. BlockedResult = Result { ":" Block }.
5. Assign = "=" BlockedResult ":" Pattern.
6. Pattern = { PatternTerm }.
7. Result = { ResultTerm }.
8. PatternTerm =
9. CommonTerm | "(" Pattern ")" | "[" NAME Pattern "]" | RedefinitionVariable.
10. RedefinitionVariable = VARIABLE "^".
11. ResultTerm =
12. CommonTerm | "(" Result ")" | "[" NAME Result "]" | "<" Result ">" | Block.
13. CommonTerm = CHAR | NUMBER | NAME | VARIABLE | "#" NAME.
```

Листинг 1. Синтаксис функции.

Глобальная регулярная функция представляет собой именованный блок, который может предваряться ключевым словом $ENTRY . Блок, в свою очередь, содержит последовательность нуля или более предложений.

Предложение состоит из двух частей: образцовой части и результатной части, разделённых знаком «=». Каждое предложение заканчивается точкой с запятой.

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
- 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

type.*index* , где

type может быть одной из букв s , t или e ,

index — 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 [1].

1.2. Описание компилятора Простого Рефала и Рефала-5λ . Проходы компиляции.

Компилятор Рефала-5λ на стадии анализа поддерживает два языка: Простой Рефал и Рефал-5λ, которые являются расширением Рефала-5. Он может порождать как интерпретируемый код, так и код на C++.

Компилятор создает исполнимые файлы операционной системы, состоящие из префикса, интерпретатора и интерпретируемого кода. Префикс представляет собой программу на языке C++, скомпилированную в исполнимый файл.

Компилятор Рефала-5λ может использоваться как готовый (ранее скомпилированный) префикс, так и «подготавливать» префикс в процессе компиляции (вызывая компилятор C++). Последний режим используется в случае, если исходные файлы содержат вставки кода C++, либо если отключена оптимизация прямой кодогенерации.

На данный момент синтаксис Рефала-5λ поддерживает условия, в Простом Рефале такого нет, на более ранней стадии функции с условиями преобразуются в эквивалентные функции без условий, что не оптимально. На стадии синтеза условия не поддерживаются, поэтому их следует добавить.

Компиляция является многопроходной, это означает, что абстрактное синтаксическое дерево программы обрабатывается несколько раз. Проходы компилятора представлены на Рис. 1.

Рассмотрим основные проходы компиляции и функции, отвечающие за их выполнение.

Один из основных проходов, на который стоит обратить внимание, — это синтаксический анализ.

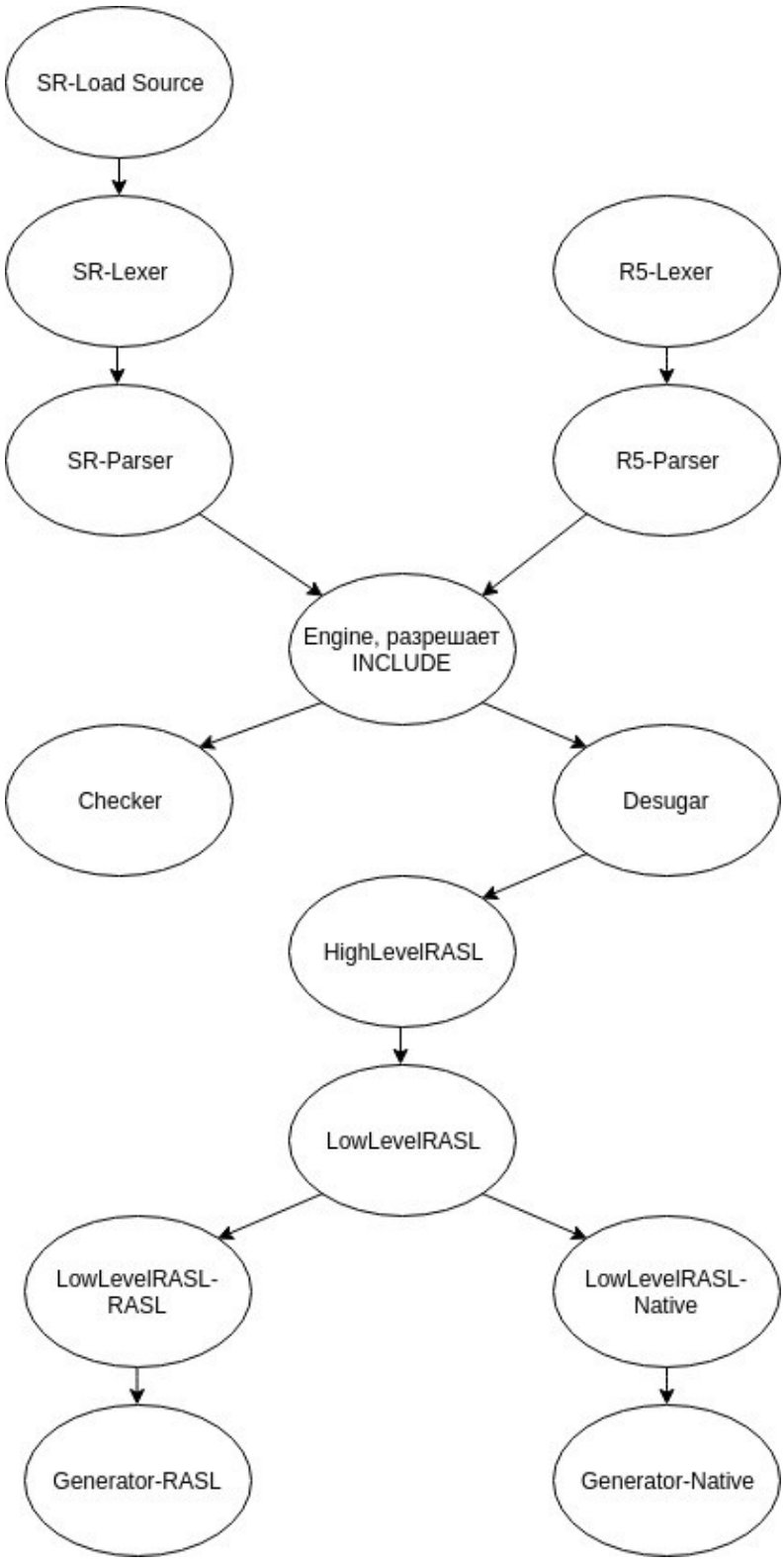

Рис.1. Проходы компилятора.

На этом этапе грамматика для предложения функции имеет вид, представленный на Листинге 2.

```
Block = "{" {Sentence} "}".
Sentence = Pattern { Assign } "=" BlockedResult.
BlockedResult = Result { ":" Block }.
Assign = "=" BlockedResult ":" Pattern.
```

Листинг 2. Грамматика предложения функции Простого Рефала без поддержки условий.

Синтаксический анализ выполняется с помощью функции SR-ParseProgram из файла «SR-Parser.sref». Она принимает на вход список ошибок и последовательность токенов, после чего вызывается функция ParseElements, которая выполняет синтаксический анализ методом рекурсивного спуска и возвращает расширенный список ошибок и абстрактное синтаксическое дерево, представляющее собой древовидную структуру, в которой хранится информация о функциях и предложениях, входящих в эти функции.

Грамматика абстрактного синтаксического дерева отражена на Листинге 3.

```
e.AST ::= t.ProgramElement*
t.ProgramElement ::=
    (#Function t.SrcPos s.ScopeClass (e.Name) e.Body)
  | (#Enum t.SrcPos s.ScopeClass e.Name)
  | (#Swap t.SrcPos s.ScopeClass e.Name)
  | (#Declaration t.SrcPos s.ScopeClass e.Name)
  | (#Ident t.SrcPos e.Name)
  | (#Include t.SrcPos e.Name)
  | (#NativeBlock t.SrcPos e.Code)
```

Листинг 3. Грамматика абстрактного синтаксического дерева.

Структура узла абстрактоного синтаксического дерева, который связан с предложением функции, написанной на Простом Рефале отражена на Листинге 4. Под тэгом #Condition подразумевается символ “, ”, под #Assign - символ “ = “.

```
t.FunctionBody ::= (#Sentences ((e.Pattern) t.Assignment* (e.Result))*)
t.Assignment ::= (#Assign (e.Result) (e.Blocks) (e.Pattern))
```

Листинг 4. Формат узлов AST для функции без поддержки условий.

Следующий проход представляет собой разрешение подключаемых файлов и не выделен в отдельную функцию. Он выполняется внутри Engine.sref, обнаруживает в исходных текстах директивы $INCLUDE (элемент дерева (#Include t.SrcPos e.Name)) и подгружает содержимое указанных файлов.

После получения всех синтаксических деревьев строится результирующее дерево, равное их конкатенации, при этом в нём изменяются t.SrcPos и удаляются узлы #Include за ненадобностью. [3]

«Обессахариватель» - фаза компиляции, упрощающая синтаксическое дерево до более компактного подмножества (устраняющая «синтаксический сахар»). В частности, такие конструкции, как блоки и присваивания преобразуются в вызовы вспомогательной функции, переопределения переменных сводятся к их переименованию (использованию нового имени).

Вложенные функции также являются «синтаксическим сахаром» — для каждой из них неявно создаётся глобальная функция, на место вложенной функции помещается операция построения замыкания — связывания глобальной функции с контекстом.

На данным момент «обессахариватель» превращает условие во вспомогательную функцию, которая проверяет выполнение условия, и, если оно не выполняется, передаёт управление предложениям, представленном на Листинге 5.

```
F {
    . . .
  Pat1, ResC1: PatC1 = Res1;
  Pat2 = Res2;
  . . .
}
↓ ↓ ↓
F {
  . . .
  Pat1 = <F_check [перем] ResC1>;
  e.X = <F_cont e.X>;
}
```

```
F_check {
  [перем] PatC1 = Res1;
  [перем] e.Other = <F_cont Pat1>;
}
F_cont {
  Pat2 = Res2;
  . . .
}
```

Листинг 5.

Здесь F_check — функция проверки условия, F_cont — функция-«континуация», образованная из предложений, следующих за условием. Континуация вводится только для упрощения и избежания дублирования кода. Как [перем] обозначен набор переменных, присутствовавших в образце.

Если образец перед условием содержит открытые e-переменные, добавляются также функции F_forward , которая осуществляет продвижение открытой e-переменной на один терм вперёд (если это не удаётся, то восстанавливается аргумент функции и управление передаётся на континуацию) и F_next, которая осуществляет сопоставление с уже «продвинутой» открытой переменной [4].

После проверки контекстных зависимостей и «обессахаривателя» по абстрактному синтаксическому дереву строится промежуточное представление, называемое RASL (Refal Assembly Language). RASL представляет собой последовательность команд различного типа.

В компиляторе Простого Рефала промежуточное представлние формируется в два прохода, в первом из которых генерируется высокоуровневый RASL, а во втором — низкоуровневый RASL.

Выскоуровневый RASL представляет собой промежуточный императивный язык, содержащий вложенные конструкции — предложения и циклы удлинения переменных. Низкоуровневый RASL можно считать некоторым аналогом ассемблера.

За генерацию высокоуровневого RASLa отвечает функция HighLevelRASL, расположенная в файле «HighLevelRASL.sref». В ней для каждой функции из дерева вызывается HighLevelRASL-Function, для каждого предложения в функции HighLevelRASL-Sentence, для левой и правой части предложения GenPattern и GenResult соответственно. Эти функции отвечают за генерацию команд RASL для функций, предложений, образца и результата соответственно. В полученном промежуточном представлении каждой функции соответствует блок команд RASL.

Модель команд RASL можно описать следующим образом. Последовательность команд внутри функций, обозначаемых #Function, должна начинаться с #CmdIssueMem, которая резервирует массив context. Далее располагаются команды сопоставления с образцом (например, #CmdChar) и вложенные предложения #CmdSentence (содержит информацию о допустимых альтернативах в случае ошибки сопоставления). Затем либо должна следовать составная команда #CmdOpenELoop — цикл по открытой e-переменной, за которой следует #CmdFail (возврат из функции значения «сопоставление невозможно»), либо команды создания новых узлов и команды сборки результатного выражения, завершающиеся #CmdReturnResult. Составные команды внутри себя должны иметь точно такую же структуру.

Низкоуровневый RASL создается функцией LowLevelRASL из файла «LowLevelRASL.sref», задачей которой является развёртывание составных инструкций промежуточного кода (#CmdSentence и #CmdOpenELoop) в более примитивные команды и формирование интерпретируемого кода. LowLevelRASL, в зависимости от переданной опции, осуществляет прямую кодогенерацию, генерирует интерпретируемый код с вызовом интерпретатора, или выполняет и то, и другое. Процедура развертывания команд напрямую зависит от выбранной опции.

Последний проход— генерация целевого кода на C++ на основе RASL. Это осуществляется с помощью функции GenProgram из файла «Generator.sref» .

Функция GenCommand генерирует код для отдельной команды. Функции, используемые в сгенерированном коде, объявляются в файлах «refalrts.h» и «refalrts.cpp».

Таким образом, компиляция выполняется за несколько проходов. Текст программы преобразуется в поток токенов с помощью лексического анализатора. Из потока токенов, в результате синтаксического анализа, получается абстрактное синтаксическое дерево. Абстрактное синтаксическое дерево преобразуется в промежуточное представление, называемое RASL. RASL модифицируется в соответствии с режимом кодогенерации. Далее по модифицированному RASL генерируется целевой код на языке C++.

Программа, написанная на Простом Рефале, может быть скомпилирована в интерпретируемый код или как «нативная» функция в C++. Также есть несколько режимов оптимизации:

1. оптимизация прямой кодогенерации (ключ «-Od»)
2. оптимизация совместного сопоставления с образцом (ключ «-OP»)
3. оптимизация построения результатного выражения (ключ «-OR»)

При оптимизации совместного сопоставления с образцом генерация команд промежуточного представления для каждого предложения в функции заменяется на генерацию команд сопоставления для общих частей предложений как одного блока и генерацию команд для оставшихся частей предложений.

Оптимизация результатных выражений строит последовательность команд преобразования вызова функции в поле зрения в результатное выражение функции, минимизируя вычислительные затраты на этапе выполнения. Однако, компиляция с использованием этой опции проходит дольше. Оптимизация выполняется путём поиска похожих фрагментов выражения вызова функции (образцовое выражение предложения, окружённое скобками вызова функции и именем функции) и результатного выражения функции и генерации команд, учитывающих это сходство.

1. 3. Условия в Рефал-5.

Условия в Рефал-5 записывается после образца и имеют вид:

, ResultCond : PattrenCond

где

- ResultCond - произвольное результатное выражение, может содержать переменные, которые уже встречались в этом же предложении слева от запятой.
- PatternCond - произвольный образец, может содержать как старые, так и новые переменные

В общем виде предложение с условиями выглядит так:

Pattern, ResultCond1 : PattrenCond1, ResultCond2 : PattrenCond2, … = Result

Условия "работают" последовательно:

1) сопоставление переданных функцию аргументов в и образца предложения Pettern

2) значения старых переменных подставляются в ResultCond1 и PatternCond1 , в результате получаются рабочее выражение ResultCond1' и новый образец PatternCond1'

3) ResultCond1' вычисляется, результат вычислений - объектное выражение ResultCond1"

4) ResultCond1" - сопоставляется с образцом PatternCond1'

В случае успеха новые переменные с их значениями добавляются к списку старых переменных, и продолжается вычисление предложения: следующих условий или правой части. В случае неуспеха работа переходит на предыдущий образец: последний вариант сопоставления отвергается, и продолжается поиск других вариантов[1].

Цель данной курсовой работы - добавить поддержку условий в Простой Рефал.

2. Реализация.

Для реализации поддержки условий на этапе лексического анализа в компилятор Простого Рефала не понадобилось ничего добавлять, так как лексема для запятой - #TkComma уже была добавлена ранее и предложения с условиями уже правильно обрабатывались на этом этапе.

Для добавления поддержки условий потребовалось добавить в грамматику нетерминальный символ `ConditionalPattern` .

В итоге грамматика для предложения функции стала иметь вид, представленный на Листинге 6.

```
Block = "{" {Sentence} "}".
Sentence = ConditionalPattern { Assign } "=" BlockedResult.
BlockedResult = Result { ":" Block }.
Assign = "=" BlockedResult ":" Pattern.
ConditionalPattern = Pattern { "," BlockedResult ":" Pattern }.
```

Листинг 6. Грамматика предложения функции Простого Рефала с поддержкой условий.

Семантика ConditionalPattern (образца с условиями) совпадает с семантикой условий Рефала-5, описанной в пункте 1.3 расчётно-пояснительной записки.

Далее для корректной обработки условий на этапе синтаксического анализа потребовалось изменить структуру узла абстрактоного синтаксического дерева, который связан с предложением функции, написанной на Простом Рефале. Изменения отражены на Листинге 7.

```
t.Sentence ::= ((e.Pattern) t.AssignOrCondition* (e.Result) (e.Blocks))
t.AssignOrCondition ::= (#Assign (e.Result) (e.Blocks) (e.Pattern)) |
                        (#Condition (e.Result) (e.Blocks) (e.Pattern))
```

Листинг 7. Формат узлов AST для функции с поддержкой условий.

Чтобы построить AST с преобразованной структурой, понадобилось внести изменения в парсер, который находится в файле «SR-Parser.sref». Были модифицированы функции ParseSentence и ParseSentencePart.

В функции ParseSentence до внесенных изменений сначала вызвалась функция ParseSentencePart, в которой обрабатывался фрагмент, представляющий собой следующую конструкцию «Pattern = Result {Blocks}». Далее проверялся следующий токен, если он является символом:

1) «:», то рекурсивно вызывается функция ParseSentence, которой передается следующие образец и результат присваивания, после этого объектные выражения, возвращаемые функцией, вызванной рекурсивно, добавлялись в «список» присваиваний e.Assignments, каждый элемент которого представлял собой конструкцию вида (#Assign (e.Result) (e.Blocks) (e.InnerPattern)), после чего функция возвращала объектное выражение для образца и результата предложения, «список» присваиваний, объектное выражения для блоков, список ошибок и последовательность ещё не обработанных токенов ((e.StartPattern) e.Assignments (e.EndResult) (e.EndBlocks)) t.ErrorList e.Tokens.

2) «}» , символом конца файла «EOF» или неизвестным токеном, сообщение о соответсвующей ошибке добавляется в список ошибок, после чего функция ParseSentence завершалась и возвращала ((e.StartPattern) (e.Result) (e.Blocks)) t.ErrorList e.Tokens.

3) Если следующий токен является «;», то функция возвращала объектное выражение для образца и результата предложения, объектное выражения для блоков, список ошибок и последовательность ещё не обработанных токенов ((e.StartPattern) (e.Result) (e.Blocks)) t.ErrorList e.Tokens;

Предназначение функции ParseSentencePart заключалось в обработке части предложения функции вида «Pattern = Result {Blocks}». В этой функции сначала вызывался функция ParsePattern, которая разбирает образец присваивания, затем проверялось наличие символа «=» между образцом и результатом, в случае его отсутствия осуществлялось добавление новой ошибки в список ошибок и изменялся список токенов, после этого вызывалась функция ParseResult, которая анализирует результат присваивания, затем

функция ParseSentencePart завершалась и возвращала объектные выражения вида ((e.Pattern) (e.Result) (e.Blocks)) t.ErrorList e.Tokens;

При реализации поддержки условий понадобилось изменить функцию ParseSentencePart таким образом, чтобы она также обрабатывала части предложения функции вида «Pattern , Result {Blocks}». Для этого та часть функции, где проверялось наличие символа «=» между образцом и результатом, была заменена на функцию Fetch, в которой выполняется проверяется какой символ стоит между образцом и результатом. Если этот символ «=» или «,», то функция Fetch возвращает соответствующий тег, список ошибок и ещё не обработанных токенов. Если символ после образца является недопустимым, то считается, что пропущен символ «=» и соответсвующая ошибка добавляется в список ошибок и функция Fetch возвращает тег #Assign, обновленнный список ошибок и список ещё не обработанных токенов.

По причине того, что между образцом и результатом могут быть разные символы потребовалось создать новую переменную s.Type, в которой хранится тег, позволяющий отличать присваивания и условия.

Модифицированная функция ParseSentencePart представлена на Листинге 9.

```
1.ParseSentencePart {
2.  t.ErrorList e.Tokens
3.    = <ParsePattern t.ErrorList e.Tokens> : t.ErrorList^ (e.Pattern) e.Tokens^
4.    = <Fetch
5.       e.Tokens
6.       {
7.         (#TkReplace t.SrcPos) e.Tokens^ = #Assign t.ErrorList e.Tokens;
8.         (#TkComma t.SrcPos)  e.Tokens^ = #Condition t.ErrorList e.Tokens;
9.         t.Unexpected e.Tokens^
10.          = #Assign <AddUnexpected t.ErrorList t.Unexpected '"," or "="'>
11.            t.Unexpected e.Tokens;
12.       }
13.     >
14.    : s.Type t.ErrorList^ e.Tokens^
15.    = <ParseResult t.ErrorList e.Tokens>
16.    : t.ErrorList^ (e.Result) (e.Blocks) e.Tokens^
17.    = ((e.Pattern) s.Type (e.Result) (e.Blocks)) t.ErrorList e.Tokens;
18.}
```

Листинг 8. Функция ParseSentencePart после добавления обработки условий.

В функции ParseSentence «список» присваиваний e.Assignments был заменен на «список» присваиваний и условий — e.AssignmentsANDCond, каждый элемент которого представляет собой конструкцию вида (s.Type (e.Result) (e.Blocks) (e.InnerPattern)). Кроме того была добавлена обработка ошибки, когда условие является последним в предложении и отсутствует символ «=» перед результатом предложения.

```
1. ParseSentence {
2.   t.ErrorList e.Tokens
3.     = <ParseSentencePart t.ErrorList e.Tokens>
4.     : ((e.StartPattern) s.Type (e.Result) (e.Blocks))
5.       t.ErrorList^ t.NextToken e.Tokens^
6.     = t.NextToken
7.     : {
8.         (#TkSemicolon t.SrcPos)
9.           = <Fetch
10.              s.Type
11.              {
12.                #Assign
13.                  = t.ErrorList;
14.                #Condition
15.                  = <EL-AddErrorAt
16.                      t.ErrorList t.SrcPos 'Before the last result expr must be "="'
17.                    >;
18.              }
19.            >
20.           : t.ErrorList^
21.           = ((e.StartPattern) (e.Result) (e.Blocks)) t.ErrorList e.Tokens;
22.         (#TkColon t.SrcPos)
23.           = <ParseSentence t.ErrorList e.Tokens>
24.           : ((e.InnerPattern) e.AssignmentsANDCond (e.EndResult) (e.EndBlocks))
25.             t.ErrorList^ e.Tokens^
26.           = (s.Type (e.Result) (e.Blocks) (e.InnerPattern)) e.AssignmentsANDCond
27.           : e.AssignmentsANDCond^
28.           = ((e.StartPattern) e.AssignmentsANDCond (e.EndResult) (e.EndBlocks))
29.             t.ErrorList e.Tokens;
30.         (#TkCloseBlock t.SrcPos)
```

```
31.          = ((e.StartPattern) (e.Result) (e.Blocks))
32.            <EL-AddErrorAt t.ErrorList t.SrcPos 'Missed semicolon'>
33.            (#TkCloseBlock t.SrcPos) e.Tokens;
34.        (#TkEOF t.SrcPos)
35.          = e.Tokens : /* пусто */
36.          = ((e.StartPattern) (e.Result) (e.Blocks))
37.            <EL-AddErrorAt t.ErrorList t.SrcPos 'Unexpected EOF in function'>
38.            (#TkEOF t.SrcPos);
39.        t.Unexpected
40.          = ((e.StartPattern) (e.Result) (e.Blocks))
41.            <AddUnexpected t.ErrorList t.Unexpected 'semicolon'>
42.            e.Tokens;
43.    };
44.}
```

Листинг 9. Функция ParseSentence после добавления обработки условий.

При реализации присваиваний уже были сделаны правки в обессахариватель и модификации проверки контекстно-зависимых ошибок, таким образом, что на этих проходах уже осуществляется поддержка условий.

Функции, содержащие предложения с условиями, конвертировались в базисный Рефал при помощи модуля Desugaring-UnCondition.ref, т.е. стадия синтеза потребляла только базисный Рефал с конструктором замыкания. После реализации поддержки условий при указании ключа «-OC» этот модуль будет отключаться.

Следующим шагом после внесения изменений на стадии синтаксического анализа была добавлена обработка условий при генерации высокоуровневого RASL’a.

Сначала были добавлены условия при компиляции без оптимизаций в случае, когда программа компилируется в интерпретируемый код. Для этого был выбрана итеративная реализация условий. Этот режим копиляции на этапе генерации высокоуровневого RASLa описывается в файле «HighLevelRASL-DisjointFunc.sref».

Так как в обессахаривателе программа преобразовывалась в базисный Рефал с конструктором замыкания и условиями, функции имели вид, представленный на Листинге 10.

```
1.  F{
2.   Pat1 = Res1;
3.   Pat2 = Res2;
4.  …
5.}
```

Листинг 10. Структура функции после обессахаривателя до добавления поддержки условий.

Всвязи с этим функция HighLevelRASL-Function-Disjoint, которая возвращала код высокоуровневого RASLa программы, обрабатывала только конструкции вида «Pattern = Result» для каждого предложения функции. Это происходило следующим образом:

1) Построение начальной подстановки с помощью функции GenInitSubst-Save или GenInitSubst-Simple и генерация последовательности команд для инициализации этой начальной подстановки.

2) Преобразование подстановки e.0 → Pattern в последовательность команд сопоставления и «размеченный образец», используемый при построении результата (в общем случае).

3) Генерация команд построения результатного выражения с помощью функций GenResult-Opt или GenResult-Simple

4) Если в образце есть команды-заголовки циклов удлинения открытых переменных, команды за ними группируются в циклы.

Таким образом каждое предложение функции превращается в отдельную цепочку команд.

Если существует функция, представленная на Листинге 10, то поле зрения преобразуется следующим образом:

<F Pat> → <F Pat <F$n?1 Res1>>,

где Res1 — подчинённое поле зрения, n — номер предложения с условием.

```
1. F {
2.   ...
3.   Pat , Res1 : Pat1, Res2 : Pat2 = Res;
4.   ...
5. }
```

Листинг 11. Функция F.

После завершения вычисления Res1 внутри <F$n?1 …> окажется результат выполнения Res1, который надо сопоставить с Pat1. Если сопоставление возможно, то выполняется правая часть (Res). Если нет — или подбирается следующая подстановка переменных в Pat (удлиняются e-переменные), или осуществляется переход к следующему предложению. При этом из аргумента функции удаляется <F$n?1 …>.

Функция F$n?1 в текущей реализации вызывается как обычная функция (когда Res1 полностью вычислится) и служит для возврата от подчинённого поля зрения к объемлющему — для возобновления выполнения функции F.

Вычислить вспомогательное поле зрения можно добавить 2 способами:

1) сохранять текущее состояние рефал-машины, выполнить вспомогательное поле зрения, восстановить состояние (итеративная реализация)

2) рекурсивно вызывать рефал-машину (рекурсивная реализация)

Для функции F, представленной на Листинге 11, формат выскоуровневого RASLa для предложения этой функции представлен на Листинге 12.

```
1.(#CmdSentence
2.  команды сопоставления Pat
3.  команды порождения <F$k?1 Res1>
4.  выполнение подчиненного поля зрения
5.    (#CmdSentence
6.      команды сопоставления Pat1
7.      команды порождения <F$k?2 Res2>
8.      выполнение подчиненного поля зрения
9.        (#CmdSentence
10.         команды сопоставления с <F$k?1 Pat2>
11.         команды порождения Res
12.         (#CmdNextStep)
13.       )
```

```
14.       команда удаления <F$k?2 …>
15.     )
16.     команда удаления <F$k?1 …>
17. )
```

Листинг 12. Формат высокоуровневого RASLa для предложения функции F.

Способ порождения команд сопоставления образца предложения остался прежним за исключением того, что начальная подстановка генерируется GenInitSubst-Save, так как необходимо хранить смещение на стеке и положение закрывающей скобки вызова «>» (чтобы перед ним разместить <F$k?1 Res1>).

Команды порождения <F$k?1 Res1> должны порождаться новой функцией GenResult-Condition. Вернее, в функцию GenResult-Condition должно передаваться только F$k?1 Pat2, угловые скобки надо размещать отдельно, чтобы самим контролировать их положение в стеке (контексте).

Порождение команд сопоставления с <F$k?1 Pat2> отличается от порождения команд сопоставления с образцом в случае присваивания тем, что начальная подстановка должна осуществляться с помощью новой функции GenInitSubst-Cond, так как обе функции GenInitSubst-Save и GenInitSubst-Simple добавляют команду инициализации нулевого диапазона, что в случае условий не нужно («нулевой» диапазон — это угловые скобки созданные вокруг F$k?1 Res2, построенные на предыдущем этапе).

Команды порождения Res2 должна генерироваться при помощи s.FnGenResult, но e.MarkedPattern должен состоять из размеченного образца с вызовом условия <F Pat1 <F$k?1 Pat2>>. Функции GenSubst-Save или GenSubst-Simple вызываются каждый раз при сопоставлении с образцом и от них получаются несколько e.MarkedPattern, которые нужно скомбинировать.

Для генерации команды удаления <F$k?1 …> нужно создать новую команду (#CmdSpliceToFreeList-Range s.Start s.End), которая переносит из поля зрения указанный диапазон и удаляет фрагмент условия.

Была создана функция GenInitSubst-Cond (Листинг 13) специально для условий, которая копирует все переменные для построения выражения в условии.

```
$ENTRY GenInitSubst-Cond {
  s.ContextTop (e.FuncName) e.Pattern
    = s.ContextTop : s.0
    = <Add s.ContextTop 1> : s.1
    = <Add s.ContextTop 2> : s.2
    = <Add s.ContextTop 4> : s.4
    = <Add s.ContextTop 5> : s.5
    = s.5
      (#Junk (#TkOpenCall s.0) (#TkName e.FuncName s.4))
       (#E s.2 e.Pattern)
       (#Junk (#TkCloseCall s.1))
       (
         (#CmdCallSave #AlgLeft s.0 s.2)
       );
}
```

Листинг 13. Функция GenInitSubst-Cond.

Также было замечено, что формирование результатного выражения в условии нельзя осуществлять при помощи уже существующих функций, генерирующих команды для результата, поскольку они по умолчанию при построении результата разрушает аргумент, поэтому для генерации команд при формировании результата условия потребовалось создать функцию GenResult-Condition (Листинг 14), которая вызывает функцию GenResult для получения последовательности команд построения выражения, после чего добавляет дополнительные операции для получения правой части.

```
$ENTRY GenResult-Condition {
  s.ContextOffset (e.PatternVars) e.CondResult
      = <GenResult s.ContextOffset (e.PatternVars) e.CondResult>
      : {
           s.ContextOffset^ (e.CommonVars) e.ResultCommands
```

```
6.              = s.ContextOffset
7.                <MakeVariableCommentTable e.CommonVars>
8.                (#CmdResetAllocator)
9.                (#CmdSetRes 1)
10.               e.ResultCommands
11.               (#CmdUseRes);
12.         };
13.}
```

Листинг 14. Функция GenResult-Condition.

В свою очередь функция GenResult подготавливает таблицу переменных, которая устроена таким образом, что сначала для переменных, пока их экземпляры есть в образцовом выражении, формирует команды переноса, а когда экземпляры кончились, формирует команды копирования переменных. Функция GenResult для условий (Листинг 15) похожа на функцию GenResult, которая используется для обработки результата предложения, только таблица переменных инициализируется иначе.

```
1.GenResult{
2.  s.ContextOffset (e.PatternVars) e.CondResult
3.    = <Map
4.        {
5.          (s.Mode (e.Index) e.Offsets)
6.            = (s.Mode (e.Index) (e.Offsets));
7.        }
8.        e.PatternVars
9.      >
10.    : e.PatternVars^
11.    = <DoGenResult
12.        (e.PatternVars)
13.        (/* alloc commands */) (/* other commands */)
14.        s.ContextOffset // счётчик новых элементов
15.        <CollectStrings e.CondResult>
16.      >;
17.}
```

Листинг 15. Функция GenResult для условий.

После того как таблица переменных сформирована вызывается циклическая функция DoGenResult, которая для каждого элемента результата формирует команду сопоставления. Для поддержки обработки условий в функцию DoGenResult, которая используется при обработке результата предложения, была добавлена обработка для открывающейся и закрывающейся скобки (Листинг 16).

```
DoGenResult {
 (e.Vars) (e.AllocCommands) (e.Commands)
 s.Counter (#TkOpenCallCond s.Offset) e.Result =
   <DoGenResult
     (e.Vars)
     (e.AllocCommands (#CmdCreateElem #Allocate s.Offset #ElOpenCall))
     ((#CmdInsertElem s.Offset) e.Commands)
     s.Counter e.Result
   >;
 (e.Vars) (e.AllocCommands) (e.Commands)
 s.Counter (#TkCloseCallCond s.Offset) e.Result =
   <DoGenResult
     (e.Vars)
     (e.AllocCommands (#CmdCreateElem #Allocate s.Offset #ElCloseCall))
     (
       (#CmdPushStack s.Offset)
       (#CmdPushStack <Dec s.Offset>)
       (#CmdInsertElem s.Offset)
       e.Commands
     )
     s.Counter e.Result
   >;
 ......
}
```

Листинг 16. Функция DoGenResult для условий.

Далее потребовалось модифицировать функцию HighLevelRASL-Function-Disjoint (Приложение 1) и создать функцию SentenseTail (Приложение 2).

Добавленная обработка условий реализована таким образом, что оптимизация построения результатного выражения уже поддерживается.

Следующим этапом являлось добавление поддержки условий при оптимизации совместного сопоставления с образцом (ключ «-OP»), для этого потребовалось добавить аналогичные изменения в функцию HighLevelRASL-Function-Conjoint (Приложение 3), которая находится в файле «HighLevelRASL-ConjointFunc.sref».

При итеративном выполнении условий нужно сохранять состояние Рефал-машины, а потом восстанавливать его.

Для этого потребовалось создать структуру, описывающую состояние Рефал-машины, представленную на Листинге 17.

```
1. struct StateRefalMachine{
2. refalrts::RefalFunction *callee;
3. refalrts::Iter begin; /* нужно для icSetResArgBegin в startup_rasl */
4. refalrts::Iter end;
5. const refalrts::RASLCommand *rasl;
6. refalrts::FunctionTableItem *functions;
7. const refalrts::RefalIdentifier *idents;
8. const refalrts::RefalNumber *numbers;
9. const refalrts::StringItem *strings;
10.
11. refalrts::vm::Stack<const refalrts::RASLCommand*> open_e_stack;
12. refalrts::vm::Stack<refalrts::Iter> context;
13.
14. refalrts::Iter res;
15. refalrts::Iter trash_prev;
16. int stack_top;
17.};
```

Листинг 17. Структура, описывающая состояние Рефал-машины.

Для восстановления и сохранения состояния Рефал-машины также потребовалось добавить 2 новые операции PopState и PushState.

После этого нужно было добавить поддержку условий в режиме прямой кодогенерации, когда по командам промежуточного представления генерируется целевой код на языке C++. Для этого была выбрана рекурсивная реализация условий.

Для этого потребовалось, начиная с этапа генерации LowLevelRASL, разделить условия на #CmdConditionFunc-ToRasl и #CmdConditionFunc-ToNative.

Кроме того для рекурсивного вызова функции main_loop пришлось создать функцию recursive_call_main_loop, которая представлена на Листинге 18.

```
1.refalrts::FnResult refalrts::recursive_call_main_loop() {
2.  const  refalrts::RASLCommand rasl[] = {
3.    { refalrts::icNextStep,0,0,0},
4.  };
5.  return refalrts::vm::main_loop(rasl);
6.}
```

Листинг 18.

Затем был добавлен тег #CmdCallCondition, который в режиме интерпретиции раскрывается как представлено в Листинге 19, а в режиме прямой кодогенерации преобразовывается в код, представленный на Листинге 20.

```
1.  (#CmdCallCondition) =
2.    (#CmdPushState)
3.    (#CmdNextStep);
```

Листинг 19.

```
1. refalrts::FnResult rec_res = refalrts::recursive_call_main_loop();
2. if (rec_res != refalrts::cSuccess)
3.    return rec_res;
```

Листинг 20.

Таким образом в Простой Рефал была добавлена поддержка условий.

3. Тестирование.

Для того, чтобы убедиться, что программы с условиями обрабатываются без ошибок были добавлены специальные автотесты.

Кроме того в скрипты автотестов были добавлены все вариации флагов оптимизации, включающие в себя флаг -OC. Эти же флаги были добавлены в скрипты тестов, которые представляют собой случайно сгенерированную программу. Последние создаются при помощи генератора случайных программ Рефала-5, автором которого является Немытых А.П.

Компилятор, после добавления в него обработки условий, успешно прошел все автотесты и 300 случайно сгенерированных теста.

Также было осуществлено тестирование производительности с помощью нескольких программ.

Тестирование проводилось на Microsoft Windows [Version 10.0.16299.309] на компьютере с процессором Intel® Core™ i5-2430M CPU @ 2.40GHz, памятью 8 Гбайт при помощи компилятора Borland C++ 5.5.1 и на Ubuntu 16.04 на компьютере с процессором Intel(R) Core(TM) i7-4510U CPU @ 2.00GHz, с памятью 8 Гбайт при помощи компилятора GCC 5.4.0.

Тестирование осуществлялось с помощью следующих программ:

1) Генератор случайных программ на Рефале, который использовался для тестирования корректности работы компилятора.

Исходная версия использовала встроенные функции Рефала-5 Random и RandomDigit, но чтобы сделать его поведение детерминированным, были вручную написаны на Рефале функции Random_ и RandomDigit_, использующие генератор случайных чисел Фибоначчи с запаздываниями.

Результаты тестирования на генераторе случайных программ представлены в таблице 1 и 2.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	10.603	10.250	+3%
-OR/-ORC	9.517	9.617	-1%
-Od/-OdC	9.039	8.671	+4%
-OdR/-OdRC	8.640	8.928	-3%

Таблица 1. Тестирование на компиляторе GCC 5.4.0 на Ubuntu 16.04.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	9.285	9.118	+2%
-OR/-ORC	7.982	8.188	-3%
-Od/-OdC	7.967	7.626	+4%
-OdR/-OdRC	6.702	6.794	-1%

Таблица 2. Тестирование на компиляторе Borland C++ 5.5.1 на Microsoft Windows 10.0.

2) Генератор кода конечного автомата на Java, разбирающего данный набор строк, автором которого является Скоробогатов С.Ю. Функция Go в тесте вызывается 100 раз.

Результаты тестирования на генераторе кода конечного автомата представлены в таблице 3 .

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	8.670	7.420	+14%
-OR/-ORC	7.727	7.361	+5%
-Od/-OdC	7.718	6.947	+10%
-OdR/-OdRC	6.830	6.744	+1%

Таблица 3. Тестирование на компиляторе Borland C++ 5.5.1 на Microsoft Windows 10.0.

3) RMCC — генератор таблиц LL(1)-разбора по грамматике, автором которого является Скоробогатов С.Ю. Функция Go выполняется дважды.

Резальтаты тестирования представлены в таблице генератор таблиц представлены в таблице 4.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	11.079	10.953	+1%
-OR/-ORC	11.436	11.236	+1.7%
-Od/-OdC	9.814	9.657	+1.5%
-OdR/-OdRC	9.578	10.330	-7%

Таблица 4. Тестирование на компиляторе Borland C++ 5.5.1 на Microsoft Windows 10.0.

4) Проход суперкомпиляции суперкомпилятора SCP4, применяемый к интерпретатору Рефала, интерпретирующему функцию факториала.

Результаты тестирования компилятора на суперкомпиляторе SCP4 представлены в таблицах 5 и 6.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	33.456	33.280	+0.5%
-OR/-ORC	33.341	33.359	0%
-Od/-OdC	31.088	30.622	+10%
-OdR/-OdRC	31.494	30.669	+2%

Таблица 5. Тестирование на компиляторе GCC 5.4.0 на Ubuntu 16.04.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	21.503	20.300	+5%
-OR/-ORC	20.185	21.654	-7%
-Od/-OdC	19.138	18.978	+1%
-OdR/-OdRC	20.110	20.337	0%

Таблица 6. Тестирование на компиляторе Borland C++ 5.5.1 на Microsoft Windows 10.0.

5) Парсер арифметических выражений, выполняющий разбор путём подбора возможных разбиений исходной строки.

Тест написан специально для проверки условий. Пример функции из теста:

```
1.ParseE {
2.  e.Expr s.Op e.Term
3.    , <ParseE e.Expr> : t.Expr
4.    , <OneOf s.Op '+-'> : True
5.    , <ParseT e.Term> : t.Term
6.    = (t.Expr s.Op t.Term);
7.
8.  e.Term , <ParseT e.Term> : t.Term = t.Term;
9.
10.  e.Other = /* пусто */;
11.}
```

Листинг 21. Пример функции из специального теста.

Полный листинг находится в приложении 4.

Результаты тестирования компилятора на парсере арифметических выражений представлены в таблицах 7 и 8.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	8.979	6.330	+30%
-OR/-ORC	9.062	6.492	+28%
-Od/-OdC	8.457	6.041	+29%
-OdR/-OdRC	8.510	6.481	+24%

Таблица 7. Тестирование на компиляторе GCC 5.4.0 на Ubuntu 16.04.

Сравниваемые режимы	Время выполнения программы без оптимизации условий	Время выполнения программы с оптимизацией условий	Прирост производительности
-no opt / -OC	6.706	3.986	+40%
-OR/-ORC	5.028	4.218	+16%
-Od/-OdC	4.860	3.076	+37%
-OdR/-OdRC	3.033	2.938	+3%

Таблица 8. Тестирование на компиляторе Borland C++ 5.5.1 на Microsoft Windows 10.0.

Заключение.

В данной работе была полностью реализованная поддержка условий на всех проходах компиляции и во всех режимах компиляции. На основе результатов полученных при тестировании можно сделать вывод, что оптимизация с условиями не на всех программах дает значительный прирост производительности, а в случае, когда включены оптимизация с результатом и оптимизации с условием скорость работы программы может даже снижаться. Однако на программах, в которых в предложениях функций условия находятся в подавляющем количестве, увеличение производительности очень существенно и может достигать 30-40%.

Используемая литература.

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) Коновалов А.В. Пользовательская документация для языка Простой Рефал [Электронный ресурс]: GitHub .- Режим доступа: https://github.com/bmstu-iu9/simple-refal .

3) Коновалов А.В. Интерфейсы между отдельными проходами компиляции [Электронный ресурс]: GitHub .- Режим доступа: https://github.com/bmstu-iu9/simple-refal/blob/master/src/compiler/interfaces.md.

4) Коновалов А.В. Подход к преобразованию условий [Электронный ресурс]: GitHub .- Режим доступа: https://github.com/bmstu-iu9/simple-refal/blob/master/doc/5-to-basis/Подход к преобразованию условий.

Приложение.

Приложение 1. Функция HighLevelRASL-Function-Disjoint после добавления поддержки условий.

```
1.$ENTRY HighLevelRASL-Function-Disjoint {
2.  s.FnGenInitSubst s.FnGenSubst s.FnGenResult
3.  s.ScopeClass (e.Name) e.Sentences
4.    = <Map
5.        {
6.          ((e.Pattern) e.Conditions (e.Result))
7.            = <GenInitSubst-Save 0 (e.Name) e.Pattern>
8.            : s.ContextTop e.Substitutes (e.FirstPatternCommands)
9.            = <s.FnGenSubst
10.                 s.ContextTop e.Substitutes (e.FirstPatternCommands)
11.               >
12.             : s.StartOffsetCond
13.               (e.PatternVars) (e.MarkedPattern)
14.               e.PatternCommands
15.             = <SentenceTail
16.                 (s.StartOffsetCond)
17.                 (s.FnGenResult e.Result)
18.                 (e.PatternVars) ((e.MarkedPattern))
19.                 e.Conditions
20.               >
21.             : s.MaxOffset e.SentenceTailCommands
22.             = <FoldOpenELoops
23.                 e.PatternCommands
24.                 <PutVariableDebugTable e.PatternVars>
```

```
            e.SentenceTailCommands
          >
          : e.OuterSentence
        = ((#CmdIssueMemory  s.MaxOffset) e.OuterSentence);
    }
    e.Sentences
  >
: e.Sentences^ ((#CmdIssueMemory s.LastMemory) e.LastSentence)
= <MapReduce
    {
      s.MaxMemory ((#CmdIssueMemory s.Memory) e.Sentence) =
        <Max s.MaxMemory s.Memory>
        (#CmdSentence e.Sentence);
    }
    s.LastMemory
    e.Sentences
  >
: s.MaxMemory e.Sentences^
= (#Function
    s.ScopeClass (e.Name)
    (#CmdIssueMemory s.MaxMemory)
    e.Sentences
    <Fetch
      e.LastSentence
      {
        e.Commands (#CmdOpenELoop e.OpenELoop) =
          e.Commands (#CmdOpenELoop e.OpenELoop) (#CmdFail);
        e.LastSentence^ = e.LastSentence;
      }
    >
  );
}
```

Приложение 2. Функция SentenceTail.

```
1. $ENTRY SentenceTail{
2.  (s.StartOffsetCond)
3.  (e.ResultSentence)
4.  (e.PatternVars)(e.MarkedPatternCur)
5.  (#Condition (e.CondName) (e.CondResult) (e.CondPattern)) e.ConditionsTail
6.    = <GenResult-Condition
7.        <Inc2 s.StartOffsetCond> (e.PatternVars)
8.    (#TkOpenCallCond s.StartOffsetCond)
9.        (#TkName e.CondName) e.CondResult
10.       (#TkCloseCallCond <Inc s.StartOffsetCond>)
11.     >
12.     : s.EndOffsetCond-Res e.CondResultCommands
13.   = <GenInitSubst-Cond s.StartOffsetCond (e.CondName) e.CondPattern>
14.     : s.OffsetCond
15.       e.CondSubstitutes (e.FirstCondPatternCommands)
16.   = <GenSubst-Save-Cond
17.       s.OffsetCond e.CondSubstitutes (e.PatternVars)
18.       (e.FirstCondPatternCommands)
19.     >
20.     : s.StartOffsetCondNext
21.       (e.PatternVarsAfterCond) (e.MarkedPatternCond)
22.       e.CondPatternCommands
23.   = e.MarkedPatternCur (e.MarkedPatternCond) : e.MarkedPatternCur^
24.   = <SentenceTail
25.       (s.StartOffsetCondNext)
26.       (e.ResultSentence)
27.       (e.PatternVarsAfterCond) (e.MarkedPatternCur)
28.       e.ConditionsTail
29.     >
30.     : s.NestedStackTop e.NestedSentence
31.    = <Max s.EndOffsetCond-Res s.NestedStackTop> : s.StackTop
32.    = (#CmdSentence <FoldOpenELoops
33.                    e.CondPatternCommands
34.                    <PutVariableDebugTable e.PatternVars>
```

```
                    e.NestedSentence
                >
      )
      (#CmdSpliceToFreeList-Range
        s.StartOffsetCond
        <Inc s.StartOffsetCond>
      )
    : e.CmdSentenceCommands
  = e.CondResultCommands
    (#CmdCallCondition)
    (#CmdProfileFunction)
    e.CmdSentenceCommands
    (#CmdFail)
    : e.SentenceCommands
  = s.StackTop e.SentenceCommands;
(s.ContextOffset)
(s.FnGenResult e.Result)
(e.PatternVars)
(e.MarkedPatternCur)
/*пусто*/
  = e.MarkedPatternCur : (e.MarkedPattern t.CloseCall)  e.MarkedPatternCur^
  = <Map
      {
        (e.MarkedPattern^) = e.MarkedPattern;
      }
      e.MarkedPatternCur
    >
    : e.MarkedPatternCur^
  = <s.FnGenResult
      s.ContextOffset (e.PatternVars)
      (e.MarkedPattern e.MarkedPatternCur t.CloseCall)
      e.Result
    >
    : s.ContextCount e.ResultCommands
```

```
69.    = s.ContextCount e.ResultCommands;
70.}
```

Приложение 3. Функция HighLevelRASL-Function-Conjoint после добавления поддержки условий.

```
1. $ENTRY HighLevelRASL-Function-Conjoint {
2.   s.FnGenInitSubst s.FnGenSubst s.FnGenResult
3.   s.ScopeClass (e.Name) e.Sentences
4.    = <MapReduce
5.       {
6.         (e.HardPatterns) ((e.Pattern) e.Conditions (e.Result))
7.           = <CreateHardPattern e.Pattern> : e.HardPattern
8.           = (e.HardPatterns (e.HardPattern))
9.             ((<PatternComment e.HardPattern>) e.Conditions (e.Result));
10.       }
11.       (/* hard patterns */)
12.       e.Sentences
13.     >
14.   : (e.HardPatterns) e.SentenceTails
15.   = <CreateGlobalGen e.HardPatterns> : (e.FastGen) (e.GlobalGen)
16.   = <SplitGen (e.GlobalGen) e.SentenceTails> : (e.CommonPattern) e.SentSubsts
17.   = <GenPattern GenInitSubst-Save s.FnGenSubst (e.Name) e.CommonPattern>
18.   : s.ContextSize (e.Vars) (e.MarkedPattern) e.CommonMatchCommands
19.   = <MapReduce
20.       {
21.         (
22.           (e.Substitute) e.Substs)
23.           ((e.HardGenComment) e.Conditions (e.Result)
24.         )
25.         =
26.           (e.Substs)
27.           ((e.Substitute) (e.HardGenComment) e.Conditions (e.Result));
```

```
    }
    (e.SentSubsts) e.SentenceTails
  >
: () e.SentencesWithSubst
= <MapReduce
    {
      s.MaxMemory ((e.Substitute) (e.HardGenComment) e.Conditions (e.Result))
        = <ComposeVars (e.MarkedPattern) (e.Substitute) (e.Vars)>
        : e.SubstitutesAnsJunks
        = <MakeSavers s.ContextSize e.SubstitutesAnsJunks>
        : s.ContextSize^ (e.CmdSaves) e.SubstitutesAnsJunks^
        = <s.FnGenSubst
            s.ContextSize
            e.SubstitutesAnsJunks
            (e.CmdSaves)
          >
        : s.StartOffsetCond
          (e.PatternVars) (e.MarkedPattern^)
          e.PatternCommands
        = <SentenceTail
            (s.StartOffsetCond)
            (s.FnGenResult e.Result)
            (e.PatternVars) ((e.MarkedPattern))
            e.Conditions
          >
        : s.MaxOffset e.SentenceTailCommands
        = <FoldOpenELoops
            e.PatternCommands
            <PutVariableDebugTable e.PatternVars>
            e.SentenceTailCommands
          >
          : e.Commands
        = <Max s.MaxMemory s.MaxOffset>
          (#CmdSentence
```

```
            (#CmdComment e.HardGenComment)
            e.Commands
          );
      }
      s.ContextSize e.SentencesWithSubst
    >
  : s.MaxMemory e.Sentences^ (#CmdSentence e.LastSentence)
  = (#Function
      s.ScopeClass (e.Name)
      (#CmdIssueMemory s.MaxMemory)
      (#CmdComment 'FAST GEN:' <PatternComment e.FastGen>)
      (#CmdComment 'GLOBAL GEN:' <PatternComment e.GlobalGen>)
      e.CommonMatchCommands
      e.Sentences
      <Fetch
        e.LastSentence
        {
          e.Commands (#CmdOpenELoop e.OpenELoop) =
            e.Commands (#CmdOpenELoop e.OpenELoop) (#CmdFail);
          e.LastSentence^ = e.LastSentence;
        }
      >
    );
}
```

Приложение 4. Тест - парсер арифметических выражений

```
$ENTRY Go {
= <Check '12*(X-45/Q)-2'>
  <Check '12*(X-45/Q)-2)'>
  <Check '(1-2)/(3-4)/(K)'>
}

Check {
 e.Expr
  , <ParseE e.Expr> : t.Expr
   = <Prout '[' e.Expr '] -- valid, ' t.Expr>;
```

```
12.  e.Expr = <Prout '[' e.Expr '] -- invalid'>;
13.}
14.
15.* E → E + T | T
16.ParseE {
17.  e.Expr s.Op e.Term
18.    , <ParseE e.Expr> : t.Expr
19.    , <OneOf s.Op '+-'> : True
20.    , <ParseT e.Term> : t.Term
21.    = (t.Expr s.Op t.Term);
22.
23.  e.Term , <ParseT e.Term> : t.Term = t.Term;
24.
25.  e.Other = /* пусто */;
26.}
27.
28.OneOf {
29.  t.Term e.Samples-B t.Term e.Samples-E = True;
30.  t.Term e.Samples = False;
31.}
32.
33.* T → T * F | F
34.ParseT {
35.  e.Term s.Op e.Factor
36.    , <ParseT e.Term> : t.Term
37.    , <OneOf s.Op '*/'> : True
38.    , <ParseF e.Factor> : t.Factor
39.    = (t.Term s.Op t.Factor);
40.
41.  e.Factor , <ParseF e.Factor> : t.Factor = t.Factor;
42.
43.  e.Other = /* пусто */;
44.}
45.
46.* F → (E) | NUM | VAR
47.ParseF {
48.  '(' e.Expr ')' , <ParseE e.Expr> : t.Expr = t.Expr;
49.  e.Number , <ParseNUM e.Number> : s.Number = s.Number;
50.  e.Variable , <ParseVAR e.Variable> : s.Variable = s.Variable;
51.
52.  e.Other = /* пусто */;
53.}
54.
55.* NUM: DIGIT DIGIT*
56.ParseNUM {
```

```
  s.Digit e.Digits
    , <IsDigit s.Digit> : True
    , <IsDigits e.Digits> : True
    = <Numb s.Digit e.Digits>;

  e.Other = /* пусто */;
}

IsDigit {
  s.Digit , <Type s.Digit> : 'D' s.0 s.Digit = True;
  s.Other = False;
}

IsDigits {
  s.Digit e.Digits
    , <IsDigit s.Digit> : True
    , <IsDigits e.Digits> : True
    = True;

  /* пусто */ = True;

  e.Other = False;
}

* VAR: identifier
ParseVAR {
  e.Var-B '-' e.Var-E = /* пусто, в имени переменной не допустим дефис */;

  e.Variable
    , <Implode e.Variable>
    : {
        0 e.Tail = /* пусто */;
        s.Var /* без остатка */ = s.Var;
        s.Var e.Rest = /* пусто */;
      };
}
```