Простой Рефал. Руководство пользователя

Оглавление

1 История диалекта

РЕФАЛ — РЕкурсивный Функциональный АЛгоритмический язык[1], язык функционального программирования, ориентированный на символьные вычисления, обработку и преобразование текстов. У данного языка есть много несовместимых диалектов, Простой Рефал — один из них.

Примечание. Устоявшегося способа написания названия языка РЕФАЛ нет, встречается запись как кириллицей, так и латиницей, целиком в верхнем регистре, с большой буквы или с маленькой. В этом документе, когда речь идёт о семействе языков, я буду использовать запись в верхнем регистре (РЕФАЛ), когда будет идти речь о конкретном диалекте, буду использовать запись, принятую в руководстве конкретного языка. Если слово «рефал» будет употребляться в роли прилагательного, то оно будет писаться с маленькой буквы: рефал-выражение, рефал-программа. Поскольку я автор Простого Рефала, то я выбрал ему имя и поэтому утверждаю: оба слова пишутся с большой буквы, название диалекта можно дословно переводить на другие языки, но при этом оба слова тоже должны быть с большой буквы. Например, по-английски он называется Simple Refal (не Easy Refal, см. далее, почему).

Язык создавался мною как исследовательский проект — хотел для себя понять, как осуществляется компиляция кода на РЕФАЛе в императивный код. Целью было написание минимального, но при этом алгоритмически полного компилятора диалекта Базисного РЕФАЛа, простота транслятора (например, однопроходность) была важнее удобства программирования на языке.[2] Несмотря на это ограничение (а, возможно, благодаря ему), язык получился довольно целостным и согласованным. Компилятор оказался достаточно простым, благодаря чему в МГТУ имени Н. Э. Баумана на кафедре ИУ9 он использовался (и используется) как тестовый полигон для нескольких курсовых проектов. В частности, в настоящем дистрибутиве можно видеть результат одного из таких исследований — генерация интерпретируемого кода (о чём подробно будет сказано позже).

В процессе эволюции компилятор незначительно отошёл от своей первоначальной простоты. Во-первых, он стал использоваться как back-end для Модульного Рефала, для чего в него были добавлены такие возможности, как идентификаторы, абстрактные типы данных и статические ящики. Позже на нём исследовалась реализация вложенных функций — соответственно, язык пополнился и этим удобным средством. Однако, в обоих случаях концептуальная целостность и согласованность языка не пострадали (как мне кажется).

[1] Так расшифровывает эту аббревиатуру Немытых А. П. (Немытых, 2014), я с ним согласен.
[2] Подробнее о целях проекта и достигнутых результатах можно прочитать в файле doc/historical/note001.txt настоящего дистрибутива.

2 Начинающим

Писать хороший учебник, объясняющий язык с нуля, содержащий примеры, задачи и контрольные вопросы, я здесь не буду. Во-первых, написание учебника, в отличие от справочника, требует гораздо большего времени и сил, а во-вторых, всё уже написано до нас. Есть хороший учебник (Турчин, 1989) по похожему диалекту РЕФАЛ-5, написанный самим Валентином Фёдоровичем Турчиным, автором РЕФАЛа. Поэтому рекомендуем сначала изучить именно его, поскольку дальнейшее изложение будет вестись в предположении, что читатель владеет диалектом РЕФАЛ-5. В частности, такие понятия, как объектное выражение, абстрактная рефал-машина и др., введённые в том учебнике, здесь определяться не будут.

Прочитали учебник? Напишите программу преобразующую арифметические выражения в инфиксной нотации в постфиксную (польскую инверсную запись). Получилось? Переходите к следующему разделу.

3 Язык Простой Рефал: синтаксис и семантика

3.1 Структура программы

Программа на Простом Рефале (далее для краткости — на Рефале) состоит из набора единиц трансляции — исходных файлов, написанных на Рефале (с расширением .sref) и C++ (с расширением .cpp), последние включают в себя библиотеки первичных функций и среду поддержки времени выполнения (run-time support library, далее для краткости — рантайм). Компилятор Простого Рефала транслирует исходные тексты на Рефале в исходные тексты на C++, получившийся набор файлов транслируется компилятором C++ (совместимым со стандартом ANSI/ISO C++ 1998 года).

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

Синтаксически это выглядит так:

```
Program = { ProgramElement }.
```

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 (далее, пробелы) 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».

В любом месте, где допусти́м пробел, можно вставить и **комментарий**. Комментарии могут быть двух видов: однострочные (в стиле C) и многострочные (в стиле C++). Многострочные комментарии начинаются с `/*` и заканчиваются на `*/`, могут пересекать границы текста. Последовательность символов `/*` внутри многострочного комментария запрещена. Однострочные комментарии начинаются на `//` и заканчиваются в конце строки.

Каждая функция имеет своё уникальное **имя**. Имя представляет собой последовательность латинских букв, цифр и знаков `-` (минус) и `_` (прочерк), начинающуюся с заглавной латинской буквы. Примеры имён: `GN-Local`, `FuncArguments`, `Example12345`, `A-b_c`. Имена чувствительны к регистру. Символы `-` и `_` взаимозаменяемы, поэтому имена `A-b_c` и `A_b-c` эквивалентны. При трансляции в C++ имена программных конструкций в целевом коде получаются из имён в исходном коде путём замены знаков `-`

на _ (т.е. предыдущее имя после трансляции преобразится в `A_b_c`). Длина имён не ограничена, однако с очень длинными именами в целевом коде может отказаться работать нижележащий компилятор C++ (либо считать имена идентичными, если у них совпадают первые N символов, где N зависит от выбранного компилятора).

3.2 Программные элементы (объявления и определения)

3.2.1 Основные понятия

Программные элементы делятся на две категории: определения функций и ссылки на функции, объявленные в других единицах трансляции — объявления внешних функций (компилятор также понимает несколько других программных элементов, которые в текущей версии смысла не несут и существуют лишь по историческим причинам — см. раздел **Ошибка! Источник ссылки не найден.**).

Определение функции определяет функцию с некоторым именем, с конкретной логикой и конкретной областью видимости. Логика для регулярных функций (см. 3.2.5) описывается программистом в теле функции, для пустых функций (см. 3.2.3) и статических ящиков (см. 3.2.4) создаётся компилятором по специальному шаблону.

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.

Функцию, определённую в локальной области видимости, будем называть **локальной функцией**. Такие функции доступны по имени только в том файле, где они определены, соответственно, в различных единицах трансляции могут находиться локальные функции с одинаковыми именами.

Функция, определённая в глобальной области видимости — так называемая **entry-функция** — доступна по имени как в текущей единице трансляции (т. е. она также присутствует также в локальной области видимости того файла, где она определена), так и в других единицах, где она объявлена как внешняя (см. 3.2.2). Когда единица трансляции содержит entry-функции, говорят, что она *экспортирует* эти функции. Когда единица трансляции содержит объявления внешних функций — ссылки на entry-функции, определённые в других единицах трансляции, говорят, что она *импортирует* эти функции.

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

В целевом коде на C++ локальные функции соответствуют функциям (с определённой сигнатурой), определённым с модификатором static, entry-функции — функциям без этого модификатора. Таким образом, единица трансляции, написанная на C++, экспортирует те рефал-функции, которые имеют соответствующую сигнатуру и объявлены без модификатора static.

Синтаксис программного элемента:

```
ProgramElement =
    ExternalDeclaration    // см. 3.2.2
  | EnumDefinition         // см. 3.2.3
  | SwapDefinition         // см. 3.2.4
  | FunctionDefinition     // см. 3.2.5
  | ForwardDeclaration     // см. 3.2.6
  | IdentifierDefinition   // см. 3.2.6
  | ";".
```

Примечание. Таким образом, синтаксис Простого Рефала допускает на верхнем уровне, помимо объявлений и определений, любое количество точек с запятой. Программисты на Рефале-6 при желании могут после каждого блока функции ставить точку с запятой.

3.2.2 Объявления внешних функций ($EXTERN)

Синтаксис:

```
ExternalDeclaration = "$EXTERN" NameList.
NameList = NAME { "," NAME } ";".
```

Объявление функции добавляет в текущую область видимости соответствующие имена функций — ссылки на entry-функции, определённые в других единицах трансляции. Если в единице трансляции одна и та же функция определена и объявлена как внешняя, то определение функции имеет приоритет, но такая ситуация ошибкой не является.

Простой Рефал единицы трансляции компилирует независимо и потому не проверяет, есть ли для каждого объявления внешней функции соответствующее определение entry-функции в какой-то другой единице трансляции. В случае, когда внешняя функция объявлена, но нигде не определена, возможны два варианта. Если к этой функции есть обращения в коде, то произойдёт ошибка компоновки нижележащим компилятором C++, если же обращений нет — ошибки не будет.

Для внешних функций, объявленных в текущей единице трансляции, в начале сгенерированного файла создаются объявления внешних функций языка C++:

```
$EXTERN A, B, C;
```

Компилируется в:

```
extern refalrts::FnResult A(refalrts::Iter arg_begin, refalrts::Iter arg_end);
extern refalrts::FnResult B(refalrts::Iter arg_begin, refalrts::Iter arg_end);
extern refalrts::FnResult C(refalrts::Iter arg_begin, refalrts::Iter arg_end);
```

3.2.3 Пустые функции ($ENUM и $EENUM)

Синтаксис:

```
EnumDefinition = ( "$ENUM" | "$EENUM" ) NameList.
```

Определение пустой функции создаёт функцию с телом, не содержащим ни одного предложения. Вызов такой функции всегда приводит к аварийному остановку (авосту) программы — невозможности сопоставления. Функции, определённые при помощи ключевого слова $ENUM, имеют локальную область видимости, при помощи $EENUM (entry enum) — глобальную.

Конструкция является синтаксическим сахаром. Код

```
$ENUM L;
$EENUM E;
```

идентичен следующему (см. далее синтаксис функций):

```
L { /* пусто */ }
$ENTRY E { /* пусто */ }
```

Пустые функции существуют в языке по историческим причинам. В первых версиях языка отсутствовали идентификаторы (см. раздел 3.3.3 «Типы данных Простого Рефала»), поэтому в роли символических имён использовались имена функций. Поскольку такие функции никогда не вызываются, их тело можно сделать пустым, а поскольку их нужно много (например, для каждого домена лексем в лексическом анализаторе, для каждого узла в синтаксическом дереве, для каждой команды в промежуточном коде нужно имя), имело смысл ввести синтаксический сахар. Их использование выглядело так: если требовалось имя, используемое только в одной единице трансляции, то оно определялось как $ENUM, если в нескольких — в одной из них как $EENUM, в остальных как $EXTERN. После чего указа-

тели на эти глобальные функции можно было использовать внутри выражений в качестве символических имён. Сейчас такой необходимости нет: идентификатор (символическое имя) просто записывается внутри выражения как некоторое имя, предварённое знаком #.

Впрочем, по сравнению с идентификаторами у пустых функций есть преимущество — поддержка инкапсуляции, т. е. пустая функция, объявленный как $ENUM, будет локальна в рамках данной единицы трансляции — создать её можно либо внутри неё, либо путём копирования переменной (и аналогично проанализировать в образце). Данное свойство применяется для создания абстрактных типов данных — см. раздел 3.3.5.2.

Примечание. Идея пустых функций заимствована из РЕФАЛа-2 (Алешин, Красовский, Романенко, & Шерстнев, 1991).

3.2.4 Статические ящики ($SWAP и $ESWAP)

Синтаксис:

```
SwapDefinition = ( "$SWAP" | "$ESWAP" ) NameList.
```

Ключевое слово $SWAP определяет функцию-статический ящик с локальной областью видимости, $ESWAP (entry swap) — с глобальной областью видимости.

Статические ящики представляют собой функции с состоянием. Функция-статический ящик при вызове с любым аргументом возвращает аргумент своего предыдущего вызова, при первом вызове возвращает пустое выражение.

Статические ящики используются для моделирования глобальных переменных. Запись некоторого выражения в глобальную переменную осуществляется путём вызова статического ящика с этим выражением в качестве аргумента и игнорированием возвращаемого значения. В результате, статический ящик будет хранить данное выражение до следующего своего вызова. Но, поскольку любой вызов статического ящика изменяет его состояние, чтение из ящика становится уже несколько сложнее: необходимо сначала вызвать ящик с любым аргументом (обычно используется пустой аргумент), а затем возвращённое значение записать в статический ящик (как записать — см. выше). В коде это может выглядеть примерно так (о синтаксисе функций и библиотечной функции Fetch см. в последующих разделах):

```
$SWAP SomeGlobalValue;

//FROM LibraryEx
$EXTERN Fetch;

GetSomeGlobalValue {
  =
    <Fetch
      <SomeGlobalValue>                  // Вызов вернёт ранее установленное значение.
      {
        e.Value =
          e.Value                        // Старое значение возвращаем …
          <SetSomeGlobalValue e.Value>;  // …и снова устанавливаем.
      }
    >;
}

Nil { e.AnyExpr = /* пусто */; }  // Функция возвращает пустое значение при любом аргументе.

SetSomeGlobalValue {
  e.NewValue =
    <Nil <SomeGlobalValue e.NewValue>>;  // Игнорируем предудущее значение при помощи Nil.
}
```

Примечание. Знатоки РЕФАЛа-5 заметят, что статический ящик ведёт себя также, как и функция Rp.

Примечание 2. Если есть статические ящики, значит должны быть и нестатические? Они есть, но не здесь. В диалекте РЕФАЛ-2 (Алешин, Красовский, Романенко, & Шерстнев, 1991) существуют как статические ящики (аналогичные описанным здесь), так и динамические, которые создаются в куче во время выполнения программы. Простой Рефал не поддерживает динамические ящики, однако, за неимением лучшего заимствует данную терминологию.

3.2.5 Регулярные функции

Синтаксис:

```
FunctionDefinition = [ "$ENTRY" ] NAME Block.
```

Регулярные функции — это функции, тело которых пишет программист. Функции, определённые без ключевого слова $ENTRY, имеют локальную область видимости, с ключевым словом $ENTRY — глобальную (поэтому их так и называют — entry-функции).

Все разновидности функций (пустые функции, статические ящики, регулярные функции) компилируются в функции C++ со следующей сигнатурой:

```
static refalrts::FnResult ЛокальнаяФункция(
        refalrts::Iter arg_begin, refalrts::Iter arg_end
) {
        …
}

refalrts::FnResult EntryФункция(refalrts::Iter arg_begin, refalrts::Iter arg_end) {
        …
}
```

Смысл возвращаемого значения и параметров `arg_begin` и `arg_end` будет рассмотрен в разделе, посвящённом интерфейсу с языком C++.

3.2.6 Исторические конструкции — $FORWARD и $LABEL

Синтаксис:

```
ForwardDefinition = "$LABEL" NameList.
IdentifierDefinition = "$LABEL" NameList.
```

В текущей версии языка они игнорируются — их появление эквивалентно пустому месту.

В ранних версиях языка (до 2015 года включительно) все имена (включая функции и идентификаторы), которые упоминаются в теле функции, должны были быть выше по тексту программы объявлены или определены. Для того, чтобы сослаться на локальную функцию, определённую ниже по тексту, требовалось указать её имя (выше по тексту) после ключевого слова $FORWARD. А чтобы записать внутри выражения идентификатор, его надо было предварительно определить при помощи ключевого слова $LABEL. В текущей версии языка таких требований нет, но из соображений совместимости с унаследованным кодом компилятор такие конструкции принимает, но игнорирует.

3.3 Функции, семантика и сравнение с РЕФАЛом-5

Простой Рефал можно охарактеризовать как расширенное подмножество Базисного РЕФАЛА диалекта РЕФАЛ-5 (главы 1 и 2 руководства (Турчин, 1989)). Основными отличиями от РЕФАЛа-5 являются используемые типы данных, совсем другая стандартная библиотека, отсутствие средств Расширенного РЕФАЛа диалекта РЕФАЛ-5 (главы 4 и 6 в том же руководстве) и расширения языка, отсутствующие в РЕФАЛе-5 (самое важное из таких расширений — вложенные функции).

3.3.1 Общее с РЕФАЛом-5

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

- Функции внешне имеют сходный синтаксис (есть небольшие различия, позже на них остановимся).
- 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 (см. 1.5 в (Турчин, 1989)).
- Идентичная семантика сопоставления с образцом. Структура результатного выражения незначительно отличается.
- *(Относится не к языку, а к реализации)* Затраты времени на операции. Реализация Простого Рефала использует классическое списковое представление, поэтому особенности стоимости отдельных операций (сопоставление с открытыми и повторными переменными, стоимость конкатенации и копирования) те же, что и в реализации РЕФАЛа-5 (версия PZ Oct 29 2004).

3.3.2 Синтаксис функций

Синтаксис функции:

```
FunctionDefinition = [ "$ENTRY" ] NAME Block.
Block = "{" { Sentence } "}".
Sentence = Pattern "=" Result ";".
Pattern = { PatternTerm }.
Result = { ResultTerm }.
PatternTerm =
  CommonTerm | "(" Pattern ")" | "[" NAME Pattern "]" | RedefinitionVariable.
RedefinitionVariable = VARIABLE "^".
ResultTerm =
  CommonTerm | "(" Result ")" | "[" NAME Result "]" | "<" Result ">" | Block.
CommonTerm = CHAR | NUMBER | NAME | VARIABLE | "#" NAME.
```

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 (см. далее), 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 ^ (о них будет рассказано при описании вложенных функций).

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 ^ (подробнее в разделе 3.3.5.3). 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.

3.3.3 Типы данных Простого Рефала

Объектное выражение в Простом Рефале представляет собой последовательность **объектных термов**. Объектные термы относятся к одной из двух категорий: атомарные термы и составные термы.

Атомарные термы (атомы) представляют собой элементы данных, которые невозможно разбить на составные части механизмом сопоставления с образцом. Традиционно они называются символами (symbols), но, чтобы не возникало путаницы с литеральными символами (characters), я решил отступить от сложившейся терминологии. Виды атомов:

- **Литеральные символы** (ASCII-символы, characters) Один атом представляет собой 8-битный символ, хранимый в переменной типа char языка C++. Поддержки Уникода и кодовых страниц на уровне рантайма и стандартной библиотеки языка на данный момент нет — ввод-вывод в текстовые файлы и на консоль осуществляется побайтово средствами стандартной библиотеки C++ (`stdio.h`).
  Синтаксис. Литеральные символы записываются в одинарных кавычках, последовательность нескольких литеральных символов подряд можно записывать внутри одной пары кавычек. Для записи одинарной кавычки она удваивается. Также можно использовать escape-последовательности: `\n`, `\r`, `\t`, `\\`, `\'`, `'\NNN'` (символ с восьмеричным кодом NNN), `'\xNN'` (символ с шестнадцатеричным кодом NN) с тем же смыслом, что и в языке C++, `\dNNN` — символ *с десятичным* кодом NNN.
  Примеры:

В тексте программы	Что из себя представляет	Примечание
`'A' 'B' 'C'`	ABC	
`'ABC'`	ABC	несколько символов можно записывать слитно
`' '`		пробел
`''`	'	одинарная кавычка
`'\''`	'	тоже самое
`''''''`	'''	три кавычки
`'''A''B'`	'A'B	
`'\r\n'`	CR+LF	перевод строки в dos/windows
`'\d013\d010'`	CR+LF	запись символов в десятичном коде
`'\015\012'`	CR+LF	запись символов в восьмеричном коде
`'\x0D\x0A'`	CR+LF	запись символов в шестнадцатеричном коде

- **Целые числа.** Представляют собой десятичную запись беззнаковых целых чисел в диапазоне от 0 до 2^N-1, где N=32 при компиляции (компилятором C++) в 32-разрядный код и N=64 при компиляции в 64-разрядный код. При синтаксическом анализе целочисленное переполнение

не проверяется, поэтому в программе можно записать число хоть из 100 десятичных цифр, программа успешно скомпилируется, но это число будет проинтерпретировано как некоторое другое десятичное число, входящее в допустимый диапазон. Следствие: если вы планируете выполнять вычисления с большими числами на 64-разрядной платформе, убедитесь, что компилятор также собран под 64-разрядную платформу.
Примеры: `10`, `00010` (лидирующие нули игнорируются), `4294967295` (максимально допустимое число на 32-разрядной платформе), `18446744073709551615` (максимально допустимое число на 64-разрядной платформе).

- **Экземпляры функций,** которые делятся на два подвида. Экземпляр глобальной функции — **указатель на функцию** — записывается просто как имя функции, может присутствовать как в левой, так и в правой части. Экземпляр вложенной функции — **замыкание** — создаётся из блока — литерала вложенной функции, записанной в результатной части. Подробнее о них будет рассказано в разделах 3.3.5.1 и 3.3.5.3.
  Имя, используемое в указателе на глобальную или локальную функцию, должно быть объявлено или определено в текущей единице трансляции.
  Примеры: `Fab` (указатель на функцию `Fab`), `{ t.B = (t.A t.B); }` (замыкание из вложенной функции).
- **Идентификаторы.** Запись атома-идентификатора представляет собой имя идентификатора, предварённое символом #. Между # и именем идентификатора допустим пробел.
  Примеры: `# True`, `#False`, `# NotFound`, `# /* комментарий */ TokenString` — в последнем случае между знаком # и именем находится комментарий, запись синтаксически корректна, но лучше так не пишите.
- **Дескрипторы файлов** (указатели). Данные атомы служат для представления некоторого низкоуровневого ресурса, представляемого указателем C++ на некие данные. Записываться как литералы в тексте программы не могут, они могут порождаться только во время выполнения функциями, реализованными на C++. В стандартной библиотеке (Library) атомы этого типа используются для хранения указателей `FILE*`, за что и получили своё название (см. функцию `FOpen`). Сторонние библиотеки могут хранить в них любые другие типы ресурсов (например, сокеты из winsock). На печать выводятся (средствами Library, либо в дампе поля зрения) как `*XXX…X`, где `XXX…X` — шестнадцатеричная запись указателя.

Составные термы бывают двух видов:

- **Структурные** (круглые) **скобки.** Полностью аналогичны структурным скобкам РЕФАЛа-5.
- **Абстрактные типы данных,** они же АТД-скобки (АТД-термы), они же ADT-скобки (ADT-термы), они же квадратные скобки, они же именованные скобки. Записываются как `[ИМЯ выражение]`, где `ИМЯ` — имя некоторой функции из пространства имён в данной точке, `выражение` — объектное выражение. При отсутствии имени функции после открывающей скобки сигнализируется синтаксическая ошибка. Подробнее об абстрактных типах данных написано в разделе 3.3.5.2. Между открывающей квадратной скобкой и именем допустим пробел.

3.3.4 Семантика. Как выполняется программа

Определённое выражение (синоним: **активное выражение**) — последовательность **определённых термов**, которые, как и в случае с объектным выражением, могут быть атомарными и составными. Атомарные термы ничем не отличаются от атомов объектного выражения. Составные термы могут быть, как и в случае с объектными выражениями, структурными и именованными скобками (но внутри себя содержат уже определённое выражение), так и **скобками активации** (синоним: **скобки конкретизации**) — определёнными выражениями, заключёнными в угловые скобки.

Семантика Рефала определяется через понятие **рефал-машины** — абстрактного устройства, выполняющего программу на Рефале. Это устройство содержит две области памяти: **поле программы**, хранящее определения всех функций программы, **поле зрения**, которое хранит определённое выражение и **поле переменных** — набор ячеек памяти, каждая из которых связана со статическим ящиком и хранит объектное выражение. Поле программы фиксировано во время выполнения, поле зрения меняется на каждом шаге, ячейки статических ящиков меняются только при вызове соответствующих статических ящиков.

До начала выполнения программы поле зрения содержит определённое выражение `<Go>`, где `Go` — указатель на соответствующую глобальную функцию, все ячейки поля переменных хранят пустое выражение. Функция `Go` должна быть определена в одном из модулей программы как entry-функция.

Примечание. В РЕФАЛе-5 стартовая функция может носить либо имя `Go`, либо имя `GO`. В Простом Рефале допустим только первый вариант.

Рефал-машина работает в пошаговом режиме. На каждом шаге исполнитель находит в поле зрения **первичное активное подвыражение** — самую левую пару скобок активации, не содержащую внутри себя других скобок активации. Справа от открывающей угловой скобки должен располагаться атом-экземпляр-функции. Рефал-машина исполняет данный экземпляр функции, передавая ему в качестве **аргумента** часть первичного активного подвыражения, находящуюся между экземпляром функции и закрывающей угловой скобкой. Если справа от открывающей угловой скобки экземпляр функции отсутствует, то происходит аварийный останов рефал-машины (**авост**).

Если в поле зрения не удалось найти первичное активное подвыражение, то происходит успешный останов рефал-машины. Оставшееся на этот момент содержимое поля зрения отбрасывается, поскольку программа, как правило, вызывается ради побочных эффектов[3].

Процесс **исполнения функции** (**вызова функции**) зависит от вида функции. Если функция является регулярной, то рефал-машина последовательно перебирает предложения данной функции, сопоставляя аргумент с левыми частями до первого успешного сопоставления. **Сопоставление** образца с аргументом считается успешным, если можно с каждой переменной образца связать такое значение (атом для s-переменных, объектный терм для t-переменных и объектное выражение для e-переменных), что замена переменных на эти значения превращает образец в аргумент. При этом разные вхождения переменной с одним и тем же индексом должны заменяться на одинаковые значения. Затем, рефал-машина берёт правую часть того предложения, с левой частью которого сопоставление было успешным, и заменяет в ней все вхождения переменных на те значения, с которыми они стали связаны в процессе сопоставления. Получившееся активное выражение будет **результатом выполнения функции**. В частности, если в правой части присутствовали вложенные функции, то в результате функции на их месте будут сформированы замыкания — экземпляры этих вложенных функций, в контекстах (см. 3.3.5.3) которых будут присутствовать переменные, связанные при сопоставлении с образцом.

Примечание. Процесс сопоставления с образцом точно такой же, как и в РЕФАЛе-5 (Турчин, 1989).

Если не нашлось ни одного предложения, с левой частью которого удалось сопоставить аргумент, то происходит авост рефал-машины. Поскольку пустые функции (3.2.3) являются частным случаем регулярных, тело которых не имеет ни одного предложения, то при их вызове авост происходит всегда.

Для внешних функций, написанных не на Рефале, рефал-машина передаёт выполнение низкоуровневому коду, который анализирует аргумент, выполняет какие-то действия и формирует результат. Как

[3] Исключением может быть либо бенчмарк, либо некий юнит-тест, когда анализируется либо общая продолжительность вычисления, либо корректность работы (отсутствие авоста), которые являются внешними по отношению к рефал-машине свойствами.

правило, результат выполнения внешней функции является объектным выражением, т.е. не содержит скобок активации (в стандартной библиотеке Простого Рефала Library таких функций нет). Внешние функции используются либо для чистых вычислений, которые невозможно либо нерационально записывать на Рефале (например, арифметика или манипуляции с типами атомов), либо недетерминированные функции или функции с побочными эффектами, как правило — ввод-вывод.

При исполнении статического ящика рефал-машина в качестве результата выполнения извлекает объектное выражение из связанной с ящиком ячейки памяти и на его место кладёт аргумент вызова.

После вызова функции рефал-машина удаляет из поля зрения первичное активное подвыражение, помещая на его место результат выполнения функции. После чего переходит к следующему шагу.

3.3.4.1 Некоторые особенности реализации

Здесь будут описаны две особенности, которые отличают исполнение программы в текущей реализации от идеализированного описания выше. На семантику выполнения они никак не влияют, однако о них полезно знать при чтении дампов поля зрения при отладке программы.

Если правая часть предложения содержит вложенные функции, то при построении результата функции на месте каждой вложенной функции на верхнем уровне вместо атома-замыкания порождается терм вызова функции рантайма `refalrts::create_closure`, аргументом которого является указатель на неявно сгенерированную функцию для вложенной функции и контекст замыкания. Контекст замыкания представляет собой обычное объектное выражение, каждый терм которого соответствует переменной контекста (e-переменные записываются как скобочные термы). Эта функция в результате порождает атом замыкания. Таким образом, замыкания с контекстом требуют для своего создания один шаг рефал-машины.

Вызов замыкания с контекстом тоже осуществляется за два шага. На первом шаге рефал-машина разбирает атом замыкания на пару «указатель на глобальную функцию + контекст», заменяя замыкание на указатель и конкатенируя контекст замыкания с фактическим аргументом. На втором шаге вызывается глобальная функция обычным образом. Следовательно, при авосте внутри вложенной функции в дампе поля зрения после левой угловой скобки будет располагаться не замыкание, а глобальная функция, в начале аргумента которой будет располагаться контекст этой функции.

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

3.3.5.1 Функции как атомы и как объекты первого класса

В РЕФАЛе-5 атомы могут быть только трёх разных видов[4]: идентификаторы, литеральные символы и целые числа (там они называются макроцифрами). Простой Рефал к этому набору добавляет ещё и указатели на функции (а также и вложенные функции, но о них позже). Кроме того, если РЕФАЛ-5 требует, чтобы после `<` всегда следовало имя функции, определённой в текущей единице трансляции, то Простой Рефал снимает это ограничение. Наличие атома-функции за открывающей скобкой вызова проверяется лишь на этапе выполнения.

Что нам это даёт? Чуть меньше надёжности и гораздо больше гибкости! Теперь можно записывать и такие конструкции:

```
<s.Func некий аргумент>
<<GetFunc аргумент1> аргумент2>
```

Здесь предполагается, что `s.Func` содержит корректный указатель на функцию, а `<GetFunc…>` указатель на функцию возвращает. Знатоки РЕФАЛа-5 могут меня спросить: а чем отличается эта конструк-

[4] В руководстве (Турчин, 1989) утверждается, что язык поддерживает и вещественные числа, но по факту в реализации версии PZ Oct 29 2004 вещественные числа отсутствуют.

ция от функции Mu? Ответ: функция Mu работает в статической области видимости, т.е. вызывает функцию, имя которой совпадает с идентификатором, являющимся первым термом результата, т.е. функция Mu не способна вызвать функции, находящиеся в других единицах трансляции, если, конечно, эти функции не были импортированы директивой $EXTERN. Это, как можно догадаться, затрудняет написание библиотеки функций высшего порядка, поскольку, к примеру, функция Map, определённая в файле MyLib.ref следующим образом:

```
* файл MyLib.ref
$ENTRY Map {
  s.Func t.Next e.Tail =
    <Mu s.Func t.Next> <Map s.Func e.Tail>;

  s.Func /* пусто */ = /* всё */;
}
```

сможет преобразовывать выражение только встроенными функциями, либо функциями определёнными (либо импортированными) в MyLib.ref. А это снижает полезность библиотеки: чтобы выполнить некое новое преобразование, необходимо либо добавить соответствующую функцию в MyLib.ref, либо в ней же её импортировать. В обоих случаях приходится модифицировать файл библиотеки.

Более того, функция Map, в том виде, в каком она описана, имеет уязвимость. В качестве s.Func можно передать любой идентификатор, и, если он совпадёт с именем одной из локальных функций библиотеки, эта функция будет вызвана. Таким образом, вызов функции Mu с аргументом, полученным от вызывающего кода, может нарушить инкапсуляцию.

Аналогичная библиотека на Простом Рефале (описанная в файле, к примеру, MyLib.sref) с аналогичной функцией Map:

```
// файл MyLib.sref
$ENTRY Map {
  s.Func t.Next e.Tail =
    <s.Func t.Next> <Map s.Func e.Tail>;
  s.Func /* пусто */ = /* всё */;
}
```

сможет преобразовывать объектное выражение любыми пользовательскими функциями, доступными в других единицах трансляции, поскольку косвенный вызов функции выполняется не по имени, а по указателю. Также устраняется уязвимость, связанная с вызовом по имени локальных функций — локальные функции, определённые в некоей единице трансляции, в Простом Рефале не доступны извне, только если сам код явно не вернёт указатель на такую функцию.

В расширенной библиотеке LibraryEx уже имеется функция Map, причём она более мощная, нежели определена здесь.

Примечание. Конструкции вида <10 …>, <'A' …> или <> являются синтаксически корректными (компилятор даже не выдаёт на них предупреждения), однако, при передаче управления на них, они приводят к ошибке времени выполнения.

3.3.5.2 Абстрактные типы данных

Язык РЕФАЛ-5 реализует инкапсуляцию на уровне функций — функции, не помеченные как $ENTRY, недоступны из других единиц трансляции (однако, смотри оговорку про функцию Mu в предыдущем разделе). Данные же никак не защищены. Если функции модуля оперируют с неким термом, как абстрактным типом данных, то этот терм может быть либо скобочным термом, либо атомом — в любом случае реализация доступна вызывающему коду. Копилка также доступна для модификации любому коду в программе: функция Dgall извлекает всё содержимое копилки.

Простой Рефал реализует инкапсуляцию на уровне данных. Во-первых, как говорилось выше, можно объявить функцию-статический ящик локальной, а это значит, что внешний код не будет иметь данным в ящике никакого доступа (в противоположность доступности копилки). Во-вторых…

Во-вторых, в язык встроены абстрактные типы данных! Можно описать составной терм, доступ к содержимому которого есть только в той единице трансляции, где он создаётся. Для пользователей библиотеки он будет похож на атом — невозможно будет написать образец, который разобьёт этот терм на составные части. (но атомом не будет: сопоставляться с s-переменными он не сможет, копирование будет выполняться за время пропорциональное размеру терма).

Для создания такого терма необходимо сначала описать некоторую локальную функцию (проще всего это сделать с помощью директивы $ENUM), она будет служить тегом АТД-терма, а затем создать терм вида `[ИмяФункции некое-закрытое-содержимое]`, где `ИмяФункции` — тег АТД, `некое-закрытое-содержимое` — данные, подлежащие защите. Поскольку функция локальная, то сослаться из внешнего кода на неё невозможно, а значит и невозможно написать образец (вне единицы трансляции, где локальная функция описана), который бы получал доступ ко внутреннему содержимому.

В качестве тега можно использовать и имя entry-функции, тогда код из любой единицы трансляции сможет получить доступ к содержимому терма (импортировав функцию директивой $EXTERN и указав её имя после открывающей квадратной скобки). Такой терм не будет обеспечивать инкапсуляцию, однако может повысить надёжность программы (для обращения к таким данным надо будет писать образец вида `[Имя …]`, попытка сопоставления с `(e.Any)` сразу выявит ошибку в программе).

Примечание. Код на C++ имеет полный доступ к содержимому абстрактных типов данных, созданных в других единицах трансляции, например, базовая библиотека (Library) может выводить их содержимое в консоль или в текстовый файл.

Примечание 2. Абстрактные типы данных не полностью закрыты для кода на Рефале из других единиц трансляции — их можно сравнивать на равенство путём сопоставления с повторными t-переменными. Истинные абстрактные типы данных должны были бы сами определять операцию сравнения на равенство.

3.3.5.3 Вложенные функции

В работе (Скоробогатов & Чеповский, 2006) был предложен диалект Refal-7, первый диалект РЕФАЛа, в котором поддерживались вложенные функции. Данная идея меня заинтересовала, и я тоже решил добавить вложенные функции себе в язык (см. doc/historical/note003.txt в каталоге дистрибутива). Однако, в отличие от Refal-7, где вложенные функции могут быть как именованные, так и безымянные, в Простом Рефале поддерживаются только безымянные функции.

Функции в программе на Простом Рефале существуют в двух ипостасях. Во-первых, функция — это некий исполнимый блок кода, выполняющий некоторые действия, при передаче управления на него. Блок кода порождается из соответствующего фрагмента текста программы. Во-вторых, функция — это некий объект, присутствующий в поле зрения, с которым программа может производить некие манипуляции: копировать, сравнивать на равенство, вызывать. Функция во второй ипостаси в разделе 3.3.5.1 была названа указателем на функцию. Можно сказать, что отношения между функцией и указателем на функцию такие же, как между классом и объектом в ООП: функции существуют только на стадии компиляции, по ним порождаются **экземпляры функций**, существующие во время выполнения.

Функция состоит из имени функции и блока (неявно создаются блоки у статических ящиков и пустых функций). Блок функции содержит исполнимый код, имя служит для ссылки на этот код из других мест программы. Однако, часто пишутся функции, на которые ссылка осуществляется только один раз — почему бы для них вместо отдельного определения функции и однократного использования

имени не записать сразу блок в месте использования? При трансляции такую конструкцию можно трактовать как некоторую глобальную функцию с уникальным именем и ссылку на неё в правой части предложения. При выполнении соответствующего предложения на месте текстуальной записи такой **вложенной функции** будет помещаться указатель на эту функцию. Получается как-то так.

Вместо

```
UnBracket { (e.X) = e.X; }

Plain {
  e.Brackets =
    <Map UnBracket e.Brackets>;
}
```

Можно записать

```
Plain {
  e.Brackets =
    <Map { (e.X) = e.X; } e.Brackets>;
}
```

У внимательного читателя должен возникнуть вопрос: а что делать, если внутри вложенной функции используются переменные с теми же именами, что и во внешней функции? Ответ: в Простом Рефале такие переменные получают те значения, которые были с ними связаны во внешней функции. Таким образом, возможен случай, когда в результатной части предложения вложенной функции присутствует переменная, которой нет в образцовой части — такая переменная была связана уровнем выше.

Пример. Рассмотрим функцию, вычисляющую декартово произведение двух множеств.

```
// Функция CardProd вычисляет декартово произведение двух множеств
// <CartProd ('a' 'b' 'c') (1 2)>[5]
//   == ('a' 1) ('a' 2)  ('b' 1) ('b' 2)  ('c' 1) ('c' 2)
$ENTRY CartProd {
  (e.SetA) (e.SetB) =
    <Map
      { // № 1
        t.A =
          <Map
            { t.B = (t.A t.B); } // № 2
            e.SetB
          >;
      }
      e.SetA
    >;
}
```

В этом примере вложенная функция, помеченная как № 1, содержит ссылку на переменную `e.SetB`, связанную внутри функции `CartProd`. Функция, помеченная как № 2, ссылается на переменную `t.A`, связанную в функции № 1.

Очевидно, что объект, порождаемый из такой вложенной функции во время выполнения, будет гораздо сложнее, чем просто указатель на функцию в смысле языка C++ — он, помимо указателя на код, выполняющий вычисления, должен хранить и значения связанных переменных. Такой объект в дальнейшем будем называть **замыканием** (а указатели на глобальные функции будем рассматривать как частный случай замыкания).

[5] Нотация, используемая в комментарии, объясняется в разделе 4.1.

Введём определение. **Контекстом** замыкания вложенной функции называется множество переменных, связанных снаружи и используемых внутри функционального блока. Для примера выше контекст первой функции — переменная `e.SetB`, второй — `t.A`.

Экземпляры функций являются атомами, а значит и объектными выражениями. Их, как и другие объектные выражения, можно копировать и сравнивать на равенство (путём использования одноимённых переменных). Опишем характерные свойства экземпляров функций.

1. Экземпляры функций, как и другие атомы, копируются за константное время.
2. Передача управления на указатель на функцию выполняется за константное время. Передача управления на замыкание может выполняться как за константное время, так и за время пропорциональное размеру контекста. Последнее возможно, если в поле зрения присутствует несколько копий данного замыкания, поэтому для вызова каждого из экземпляров требуется своя копия контекста — такова списковая реализация.
   Примечание. Продолжительность всего шага рефал-машины может зависеть от структуры и размеров аргумента, а также от алгоритма, реализованного в функции: в образцах могут присутствовать открытые и повторные переменные, построение результата может требовать копирования переменных. В этом свойстве идёт речь об интервале времени между обнаружением первичного активного подвыражения и началом сопоставления с первым образцом функции.
3. Два экземпляра функции, полученные путём копирования одного атома, равны.
4. Два замыкания, построенные из текстуально разных функциональных блоков, не равны.
5. Два замыкания, имеющие разное содержимое элементов контекста, не равны.
6. Указатели на функцию равны тогда и только тогда, когда они указывают на одну и ту же функцию.

Для указателей на глобальную функцию пункт 3 является следствием пункта 6. Пункты 4—5 определяют понятие равенства двух замыканий, которое можно обобщить и на указатели на глобальные функции, считая, что контекст у них всегда пустой. Отношение равенства двух замыканий, не покрываемое пунктами 3—5, не определено и зависит от реализации. Так, например, если вложенная функция имеет пустой контекст (т.е. не ссылается на переменные, объявленные вне её), то два экземпляра данной функции, построенные независимо, в текущей реализации окажутся равными, т.к. будут представлять собой один и тот же указатель на неявно сгенерированную глобальную функцию (см. пример с функцией Plain ранее по тексту).

Внутри вложенных функций некоторые переменные в образце можно пометить знаком ^, что будет обозначать, что они не включаются в контекст, а скрывают одноимённую переменную во внешней области видимости. Если переменная, помечаемая знаком ^, присутствует в образце несколько раз, то помечаться знаком ^ должно только её первое вхождение. Применение знака ^ к переменной, которой нет во внешней области видимости (в частности, у глобальных функций) ошибкой не является.

3.3.5.4 Другие отличия

Просто кратко перечислим:

- Пустые функции. В отличие от РЕФАЛа-5, функция на Простом Рефале может иметь пустое тело — такая функция приводит к аварийному останову программы при любом аргументе. Для пустых функций есть специальный синтаксис (см. 3.2.3).
- Статические ящики — см. 3.2.4.

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

- **Условия.** Их нет. Будут в следующих версиях.

- **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 (см. 3.3.5.1), функциональность, связанная с метакодом, пока не реализована в библиотеке Простого Рефала.

3.3.7 Отличия от РЕФАЛа-5 на уровне библиотеки

Встроенных в язык функций в Простом Рефале нет. Есть лишь внешние первичные функции, которые пишутся вручную на C++. Одна из библиотек таких первичных функций входит в дистрибутив Простого Рефала и располагается в файле `Library.cpp`. Можно сравнить библиотеку `Library` с библиотекой встроенных функций РЕФАЛа-5:

- Обе библиотеки предоставляют самые базовые средства программирования: преобразование атомов (литера ↔ её ASCII-код, строка цифр ↔ число), арифметические действия, ввод-вывод на консоль и в файлы.
- Преобразование идентификатора в строку литер Простым Рефалом пока не поддерживается, преобразование строки литер в идентификатор не поддерживается принципиально (все идентификаторы литерально задаются во время компиляции).
- Функция сравнения `Compare` возвращает не знаки арифметических действий, а знаки отношений (`'<'`, `'='`, `'>'`), определена для любых двух термов.
- В РЕФАЛе-5 есть ограниченное число дескрипторов файлов, задаваемое целыми числами от 1 до 20. Функции открытия файлов связывают конкретные файлы с этими дескрипторами. В Простом Рефале функции открытия файлов создают новые дескрипторы, их можно создать столько, сколько позволит операционная система.
- Простой Рефал не поддерживает ввод с перфокарт (`Card`) и печать (`Print`, `Prout`) на АЦПУ[6]. Для чтения со стандартного ввода и вывода на стандартный вывод используются функции с нейтральными именами `ReadLine` и `WriteLine`.
- Сокращения `<+ …>`, `<- …>`, `</ …>` и т.д. для `<Add …>`, `<Sub …>`, `<Div …>` и т.д. не поддерживаются — синтаксис языка регулярнее РЕФАЛа-5.
- Арифметические функции не поддерживают неограниченный размер аргументов, атомы-числа не являются макроцифрами. При арифметическом переполнении (как и при делении на 0) результат не определён.

4 Библиотека функций

В дистрибутив Простого Рефала входит стандартная библиотека языка, представленная двумя единицами трансляции — `Library.cpp` (далее на неё будем ссылаться как Library) и `LibraryEx.sref` (далее — LibraryEx). Первая содержит набор первичных функций, т. е. функций, которые невозможно выразить на Рефале: преобразования атомов, арифметические функции, ввод-вывод, вторая — набор удобных функций, написанных на Рефале, и использующих библиотеку Library. Прежде чем переходить к рассмотрению функций, входящих в библиотеку, сначала введём нотацию для описания форматов функций (про форматы функций см. (Турчин, 1989)).

4.1 Используемые обозначения

Формат функции будем описывать следующим образом:

```
<ИмяФункции аргументы>
  == вариант результата 1
```

[6] Алфавитно-цифровое печатающее устройство

```
  == вариант результата 2
```

Где ИмяФункции — имя функции, аргументы — запись формата аргумента в виде некоего выражения с переменными, но без вызовов других функций, вариант результата — один из возможных форматов результата выполнения функции. Переменные в формате имеют имена, описывающие смысл данной части аргумента/результата.

В записи результата может использоваться знак |, обозначающий альтернативу, знак * обозначающий повторение конструкции любое количество раз, знак + — один и более раз, ? — опциональный элемент. Кроме того, для пояснения структуры отдельных переменных, входящих в формат аргумента или результата функции, может использоваться следующий вариант БНФ:

```
переменная ::= некоторое выражение | некоторое выражение 2 | …
```

4.2 Базовая библиотека (Library)

4.2.1 Add, Sub, Mul, Div, Mod

```
<Add s.X s.Y>
<Sub s.X s.Y>
<Mul s.X s.Y>
<Div s.X s.Y>
<Mod s.X s.Y>
  == s.Result
s.X, s.Y, s.Result ::= number
```

Арифметические функции. Выполняют, соответственно, сложение, вычитание, умножение, деление, вычисление остатка двух чисел. Возвращаемый результат — атом-число. При арифметическом переполнении или делении на ноль результат не определён.

4.2.2 WriteLine

```
<WriteLine e.Expr> == пусто
```

Функция осуществляет запись в стандартный вывод некоторого результатного выражения, после которого выводит символ перевода на новую строку. Внутри `e.Expr` допустимы термы любого типа. Литеральные символы выводятся как есть, числа выводятся в десятичной системе счисления, для указателей на функции и идентификаторов выводится имя данной сущности, причём после последних трёх принудительно ставится пробел, скобки (круглые и квадратные) выводятся как есть, файл выводится как *XXXX…X, где XXXX…X — шестнадцатеричное представление указателя, для замыканий в фигурных скобках выводится указатель на сгенерированную глобальную функцию + контекст данной функции.

4.2.3 ReadLine

```
<ReadLine> == e.Chars 0ʔ
```

Функция считывает символы из стандартного ввода, пока не обнаружит перевод строки, либо конец файла. Перевод строки функция не возвращает. Если обнаружился конец файла, функция завершает строку числом `0`.

4.2.4 FOpen

```
<FOpen s.Mode e.FileName> == s.FileHandle
s.Mode ::= 'r' | 'w'
```

Данная функция пытается открыть файл с данным именем в указанном режиме (соответственно, чтение либо запись). Если `e.FileName` пустое, либо содержит что-либо отличное от литеральных символов, происходит аварийный останов. Если файл открыть не удалось (нижележащий `fopen()` вернул `NULL`), то тоже происходит аварийный останов.

4.2.5 FClose

<FClose s.FileHandle> == пусто

Закрывает указанный дескриптор файла. Если нижележащий fclose() вернул EOF, то происходит аварийный останов.

4.2.6 FWriteLine

```
<FWriteLine s.FileHandle e.Expr> == s.FileHandle
```

Аналогична функции `WriteLine`, только запись производит в `s.FileHandle`.

4.2.7 FReadLine

```
<FReadLine s.FileHandle> == s.FileHandle e.Chars 0?
```

Аналогична функции `ReadLine`, только чтение производит из `s.FileHandle`.

4.2.8 Arg

```
<Arg s.Number> == пусто | char+
```

Возвращает n-й аргумент командной строки. Если номер аргумента превышает число аргументов, то возвращается пустое выражение.

Примечание. Библиотека Library не даёт возможность определить число аргументов командной строки, т. к. пустой аргумент — вполне законная конструкция как на Windows, так и на Linux. Можно вызвать программу с командной строкой `prog 1 "" 3` — три аргумента, второй из них пустой.

4.2.9 ExistFile

```
$EENUM True, False;
<ExistFile e.FileName> == True | False
```

Проверяет существование данного файла. Проверка на существование осуществляется путём попытки вызова `fopen(filename, "r")`, если функция вернула не `NULL`, то возвращается `True`, в противном случае — `False`.

4.2.10 GetEnv

```
<GetEnv e.VarName> == e.VarValue
```

Функция возвращает значение переменной среды с именем `e.VarName`. Если такая переменная не определена, возвращается пустое выражение.

4.2.11 Exit

```
<Exit s.RetCode>
```

Функция завершает программу с кодом возврата `s.RetCode`. Очевидно, ничего не возвращает.

4.2.12 System

```
<System e.Command> == пусто
```

Функция выполняет указанную команду оболочки.

4.2.13 IntFromStr

```
$EENUM Success, Fails;
```

```
<IntFromStr e.Text>
  == Success s.Number e.Rest
  == Fails e.Text
```

Функция пытается прочитать от начала текста `e.Text` последовательность литер, представляющих собой десятичную запись числа. Если `e.Text` начинается с последовательности десятичных цифр, то функция возвращает `Success`, целое число, соответствующее данной последовательности и остаток строки. Если же `e.Text` не начинается с последовательности десятичных цифр, то функция возвращает `Fails` и свой аргумент.

4.2.14 StrFromInt

```
<StrFromInt s.Number> == e.Text
```

Функция преобразует число в последовательность десятичных цифр.

4.2.15 Chr, Ord

```
<Chr s.Number> == s.Char
<Ord s.Char> == s.Number
```

Две взаимно-обратные функции. Первая возвращает литеральный символ с ASCII-кодом `s.Number`, вторая — для данного символа возвращает его ASCII-код.

4.2.16 SymbCompare

```
<SymbCompare s.Left s.Right> == '<' | '=' | '>'
```

Функция сравнивает два атома и возвращает результат сравнения. Атомы различных типов ранжируются следующим образом: число > литеральный символ > функция > идентификатор > файл. Замыкания с контекстом не поддерживаются. Числа сравниваются естественным образом, для символов сравниваются их ASCII-коды (в зависимости от компилятора C++, где тип `char` может быть как `signed`, так и `unsigned`, результат сравнения этой функцией может быть разным), функции и идентификаторы сравниваются по именам и упорядочиваются в лексикографическом порядке, файлы сравниваются как указатели в C++ (сравнение для двух указателей типа void стандартом C++ разрешается, но зависит от реализации).

4.2.17 SymbType

```
$EENUM TypeNumber, TypeCharacter, TypeFunction, TypeFile, TypeIdentifier;
<SymbType s.Atom>
  == TypeNumber | TypeCharacter | TypeFunction | TypeIdentifier | TypeFile
```

Возвращает тип атома. Попытка вызвать функцию с замыканием приведёт к аварийному останову.

4.3 Расширенная библиотека (LibraryEx)

Эта библиотека содержит набор функций, написанных на Рефале и предназначенных для упрощения процесса программирования. В неё входит ряд функций высшего порядка (типа Map или Reduce), а также удобные обёртки над функциями библиотеки Library (такие как Inc или LoadFile).

Для пояснения семантики ряда функций помимо формата будет приводиться и исходный код самих функций.

4.4 LibraryEx: надстройки над функциями базовой библиотеки и другие полезные функции

4.4.1 LoadFile

```
<LoadFile e.FileName> == (e.Line)*
```

Загружает файл в поле зрения, каждая строка файла заворачивается в отдельный скобочный терм. Если файл заканчивается на пустую строку, она игнорируется. Для чтения файла используется функция FReadLine.

4.4.2 SaveFile

```
<SaveFile (e.FileName) (e.Line)*> == пусто
```

Сохраняет последовательность строк (e.Line)* в файле с заданным именем. Для записи строк используется функция FWriteLine.

4.4.3 Inc, Dec

```
<Inc s.Number> == s.Number
<Dec s.Number> == s.Number

$ENTRY Inc {
  s.Num = <Add s.Num 1>;
}

$ENTRY Dec {
  s.Num = <Sub s.Num 1>;
}
```

Функции увеличивают и уменьшают число на единицу. Запись <Inc s.Number> компактнее и выразительнее, чем <Add s.Number 1>.

4.4.4 FastIntFromStr

```
<FastIntFromStr e.Text> == s.Number
```

Преобразует последовательность десятичных цифр в число. Если e.Text не является последовательностью десятичных цифр, происходит аварийный останов.

4.4.5 ArgList

```
<ArgList> == (e.Arg)+
```

Возвращает аргументы командной строки <Arg 0> <Arg 1> … до первого пустого значения, возвращённого функцией Arg. Если среди аргументов программы присутствует пустой аргумент, аргументы после него будут проигнорированы.

4.4.6 Compare

```
<Compare t.Left t.Right> == '<' | '=' | '>'
```

Сравнивает два терма. Термы ранжируются следующим образом: скобочный терм > число > литеральный символ > функция > идентификатор > файл. Замыкания с контекстом и абстрактные типы данных не поддерживаются. Скобочные термы сравниваются рекурсивно в лексикографическом порядке. Атомы между собой сравниваются также, как и для функции SymbCompare.

4.4.7 Compare-T

```
<Compare-T t.Left t.Right>
  == s.Relation t.Left t.Right
s.Relation ::= '<' | '=' | '>'
```

Сравнивает два терма и возвращает помимо отношения термов, также сам аргумент (суффикс -T означает, что функция прозрачная (transparent)). Сравнение выполняет аналогично Compare.

4.4.8 Type, Type-T

```
$EENUM TypeBracket;
```

```
<Type t.Term> == s.Type
<Type-T e.Expr> == s.Type e.Expr

s.Type ::= TypeNumber | TypeCharacter | TypeFunction | TypeIdentifier | TypeFile
  | TypeBracket | '*'
```

Функция Type определяет тип терма. Замыкания с контекстом и абстрактные типы данных не поддерживаются. Функция Type-T определяет тип первого терма выражения e.Expr, если выражение пустое, то в качестве типа возвращает '*'.

4.4.9 Trim

```
<Trim e.String> == e.String’
```

Функция удаляет пробельные символы (пробелы, табуляции, переводы строк CR и LF) в начале и в конце строки.

4.5 LibraryEx: функции высших порядков

4.5.1 Понятие функтора, Apply

Большинство функций высших порядков, представленных в библиотеке, оперирует таким объектом, как функтор:

```
t.Functor ::= s.Closure | (t.Functor e.BoundedArgs)
```

Функтор может быть либо экземпляром функции, либо скобочным термом, первым элементом которого является функтор. Вызов функтора осуществляется функцией Apply, которая разворачивает скобочные термы до тех пор, пока первым термом не будет атом, который затем и вызывает:

```
<Apply t.Function e.Args> ≈≈ <t.Function e.Args>

t.Function ::=
  s.Closure | (t.Function e.BoundedArgs)


$ENTRY Apply {
  s.Fn e.Argument = <s.Fn e.Argument>;

  (t.Closure e.Bounded) e.Argument =
    <Apply t.Closure e.Bounded e.Argument>;
}
```

Функции высших порядков (кроме функции Y), определённые ниже, используют функцию Apply для вызова функторов.

Исторически функторы использовались для имитации функциональности замыкания с контекстом: писалась глобальная функция, указатель на неё «связывался» с некоторым значением, которое играло роль контекста. Функция CardProd, рассмотренная ранее, с использованием глобальных функций и функторов выглядела бы так:

```
// Функция CardProd вычисляет декартово произведение двух множеств
// <CartProd ('a' 'b' 'c') (1 2)>[7]
//   == ('a' 1) ('a' 2)  ('b' 1) ('b' 2)  ('c' 1) ('c' 2)

$FORWARD CartProd-By-tA, CartProd-By-tB;

$ENTRY CartProd {
  (e.SetA) (e.SetB) =
```

[7] Нотация, используемая в комментарии, объясняется в разделе 4.1.

```
    <Map (CardProd-By-tA e.SetB) e.SetA>;
}

CartProd-By-tA {
  e.SetB t.A =
    <Map (CartProd-By-tB t.A) e.SetB>;
}

CartProd-By-tB {
  t.A t.B = (t.A t.B);
}
```

Как правило, удобнее использовать вложенные функции вместо функторов, однако функторы могут оказаться полезными в тех случаях, когда действие сводится исключительно к вызову *имеющейся* функции с фиксированной частью аргумента. Например, вызов

```
<Map (Add 3) e.Numbers>
```

лаконичнее и читабельнее, чем

```
<Map { s.Number = <Add 3 s.Number>; } e.Numbers>
```

Другой пример будет приведён в примере с функцией Fetch.

4.5.2 Map

```
<Map t.Func t.Elem*> == <t.Func t.Elem>*

$ENTRY Map {
  t.Fn t.Next e.Tail = <Apply t.Fn t.Next> <Map t.Fn e.Tail>;

  t.Fn = ;
}
```

Функция Map последовательно применяет функтор к каждому терму своего аргумента.

4.5.3 Reduce

```
<Reduce t.Func t.Acc t.Elem*> == t.Acc",
<t.Func t.Acc t.Elem> == t.Acc'

$ENTRY Reduce {
  t.Fn t.Acc t.Next e.Tail =
    <Reduce
      t.Fn <Apply t.Fn t.Acc t.Next> e.Tail
    >;

  t.Fn t.Acc = t.Acc;
}
```

Функция Reduce сворачивает каждый элемент последовательности (в порядке слева-направо) с термом-аккумулятором с применением данного функтора. Пример:

```
<Reduce Add 0 e.Numbers>    // вычисляет сумму чисел в e.Numbers
<Reduce Max 0 e.Numbers>    // вычисляет максимальное число
                            // (определение Max см. в примере к Fetch)
```

4.5.4 Fetch

```
<Fetch e.Argument t.Func> == <t.Func e.Argument>

$ENTRY Fetch {
  e.Argument t.Function =
    <Apply t.Function e.Argument>;
}
```

Функция служит для повышения читабельности вызовов вложенных функций. Пример:

Вызов с Fetch

```
Max {
  t.X t.Y =
    <Fetch
      <Compare t.X t.Y> {
        '<' = t.Y;
        s.Other = t.X;
      }
    >;
}
```

Вызов без Fetch

```
Max {
  t.X t.Y =
    <
      {
        '<' = t.Y;
        s.Other = t.X;
      }
      <Compare t.X t.Y>
    >;
}
```

Видно, что применение функции Fetch упрощает восприятие кода. Сначала описываются данные, которые требуется обработать (результат сравнения двух термов), а затем обработка этих данных. Без функции Fetch сначала описывается некая функция, и уже потом, когда взгляд переходит к аргументу, удаётся понять, для чего она описывалась.

Функцию Fetch можно сравнить с операторами if и switch-case императивных языков программирования: сначала вычисляется «выбирающее» выражение, затем, в зависимости от результата, выполняется та или иная ветвь.

Знатоки РЕФАЛа-5 заметят, что функция Fetch эквивалентна блоку, with-конструкции, с той лишь разницей, что блок в РЕФАЛе-5 является частью синтаксиса предложения, а Fetch — идиома, построенная на основе других средств языка (вложенные функции, косвенный вызов) и находится внутри результатной части. Та же функция Max на РЕФАЛе-5 выглядеть будет вот так:

```
* РЕФАЛ-5
Max {
  s.X s.Y, <Compare s.X s.Y>: {
      '-' = s.Y;
      s.Other = s.X;
    }
}
```

Примечание. В РЕФАЛе-5 функция Compare может применяться только к числам и возвращает знак разности первого и второго аргумента: `'-'`, `'0'` или `'+'`.

Вторым аргументом Fetch является не экземпляр функции, а функтор, поэтому можно писать, например, такой код:

```
// Прочитать файл и удалить начальные и конечные пробелы.
<Fetch
  <LoadFile e.FileName>
  (Map {
    (e.Line) = (<Trim e.Line>);
  })
>
```

Особую мощь функция Fetch приобретает в сочетании с функцией Seq (см. далее).

4.5.5 Y-комбинатор

```
<Y { s.Loop = { рекурсивная функция с вызовом s.Loop }; }>
  == результат рекурсивной функции

$ENTRY Y {
  s.Func = { e.Arg = <<s.Func <Y s.Func>> e.Arg>; };
}
```

Прежде чем перейти к описанию функции Y, необходимо сделать небольшое лирическое отступление. Вложенные функции позволяют записать некоторую служебную функцию в том месте, где она используется. Однако, эта служебная функция не может быть рекурсивной — чтобы вызвать себя, надо сослаться на себя, а имени у неё нет. Как же быть? Если пополнить язык именованными вложенными функциями (по аналогии с Refal-7, (Скоробогатов & Чеповский, 2006)), тогда Простой Рефал перестанет быть Простым. Есть другой способ — передать внутрь замыкания само это замыкание. Функция Y именно это и делает.

Своё название функция получила из мира математики — из комбина́торной логики. В рамках этого раздела математики вводится набор особых функций высшего порядка — комбинаторов, с применением которых можно задавать любой вычислительный процесс. Один из таких комбинаторов — комбинатор неподвижной точки (Y-комбинатор) служит для превращения нерекурсивной функции в рекурсивную. Фактически функция Y и реализует данный комбинатор.[8] Далее по тексту функцию Y будем называть Y-комбинатором.

Y-комбинатор принимает в качестве аргумента функцию, принимающую единственный аргумент — s-переменную и возвращающую замыкание, которое следует сделать рекурсивным, возвращает функцию, которая является рекурсивным эквивалентом функции, возвращаемой аргументом. При вычислении Y-комбинатора функция-аргумент комбинатора вызывается и получает аргументом функцию, которую возвращает в результатном выражении, но уже сделанную рекурсивной — функционально то же самое, что и возвращает Y-комбинатор.

Поясню на примере, т. к. без примера здесь не разобраться. Допустим, мы хотим написать функцию, итеративно вычисляющую (т.е. с применением хвостовой рекурсии) факториал:

```
$FORWARD DoFact;

Fact {
  0 = 1;
  s.N = <DoFact s.N 1 1>;
}

DoFact {
  s.N s.Prod s.N = <Mul s.Prod s.N>;
  s.N s.Prod s.K = <DoFact s.N <Mul s.Prod s.K> <Inc s.K>>;
}
```

Здесь функция DoFact является служебной, однако просто так её записать внутри правой части функции Fact нельзя — функция является рекурсивной, а вложенные функции должны быть безымянными.

Если бы программа писалась на Refal-7, то можно было бы записать вот так:

```
Fact {
  0 = 1;

  s.N =
    <Fetch
      1 1
      $func Loop {
        s.Prod s.N = <Mul s.Prod s.N>;
        s.Prod s.K = <Loop <Mul s.Prod s.K> <Inc s.K>>;
      }
```

[8] Подробнее про Y-комбинатор можно прочитать по следующим ссылкам:

- https://ru.wikipedia.org/wiki/Комбинатор_неподвижной_точки
- http://www.wikiznanie.ru/ru-wz/index.php/Комбинатор_неподвижной_точки
- http://habrahabr.ru/post/79713/

```
    >;
}
```

Вложенная функция имеет имя `Loop`, поэтому может вызывать себя рекурсивно. На Простом Рефале на помощь приходит Y-комбинатор:

```
Fact {
  0 = 1;

  s.N =
    <Fetch
      1 1
      <Y
        {  // № 1
          s.Loop = {  // № 2
            s.Prod s.N = <Mul s.Prod s.N>;
            s.Prod s.K = <s.Loop <Mul s.Prod s.K> <Inc s.K>>;
          };
        }
      >
    >;
}
```

Y-комбинатор превращает функцию, помеченную как № 2 в рекурсивную и её же возвращает (эту функцию вызывает функция `Fetch`). Внутри функции № 2 к себе рекурсивной можно обращаться через переменную `s.Loop`, которая выполняет роль имени функции.

Примечание. Y-комбинатор — единственная функция высшего порядка в библиотеке, которая не работает с функторами.

Примечание 2. Если вам непонятно, как работает Y-комбинатор, относитесь к нему как к ~~магии~~ особому синтаксису для записи рекурсивных вложенных функций.

4.5.6 MapReduce

```
<MapReduce t.Func t.Acc t.Elem*> == t.Acc” e.Mapped*,
где <t.Func t.Acc t.Elem> == t.Acc’ e.Mapped
```

Функция `MapReduce` сочетает в себе свойства функций `Map` (преобразование каждого элемента последовательности функтором) и `Reduce` (свёртка каждого элемента последовательности с аккумулятором). Функтор принимает два терма: текущее значение аккумулятора и очередной элемент, возвращает модифицированное значение аккумулятора и некоторое объектное выражение — результат трансформации элемента. Функция `MapReduce` применяет функтор последовательно к каждому элементу и очередному значению аккумулятора, возвращает результирующее значение аккумулятора и результаты трансформации всех элементов. Приведём пример — напишем функцию, которая загружает файл и нумерует в нём строчки:

```
LoadNumerated {
  e.FileName =
    <DelAccumulator  // См. далее
      <MapReduce
        {
          s.Number (e.Line) =
            <Inc s.Number> (s.Number e.Line);
        }
        1  // начальное значение счётчика
        <LoadFile e.FileName>
      >
    >;
}
```

Вложенная функция принимает два терма: номер текущей строки и строку из файла, возвращает номер следующей строки (на единицу больше) и преобразованную строку файла, первым термом содержащую число — её номер. Значение аккумулятора после анализа нам не нужно, оно отбрасывается при помощи функции `DelAccumulator`.

4.5.7 UnBracket

```
<UnBracket (e.Expr)> == e.Expr

$ENTRY UnBracket {
  (e.Expr) = e.Expr;
}
```

Снимает скобки с выражения. Может быть полезна в сочетании с другими функциями высшего порядка.

4.5.8 DelAccumulator

```
<DelAccumulator t.Acc e.Data> == e.Data

$ENTRY DelAccumulator {
  t.Acc e.Tail = e.Tail;
}
```

Функция `DelAccumulator` отбрасывает первый терм от своего аргумента, используется в сочетании с функцией `MapReduce` — когда значение аккумулятора по завершении вычислений стало уже не нужным.

4.5.9 Seq

```
<Seq t.Func1 t.Func2 … t.FuncN> ==
  { e.Arg = <t.FuncN … <t.Func2 <t.Func1 e.Arg>>…>; }

$ENTRY Seq {
  t.Func = t.Func;

  t.Func e.Funcs =
    {
      e.Arg = <Fetch <Apply t.Func e.Arg> <Seq e.Funcs>>;
    };

  /* пусто */ = { e.Arg = e.Arg; };
}
```

Функция `Seq` возвращает композицию функторов, переданных в качестве аргумента. Как правило, используется в сочетании с функцией `Fetch` следующим образом:

```
<Fetch
  аргумент
  <Seq
    {
      образец =
        результат 1;
    }
    {
      образец 1 =
        результат 2;
    }
    {
      образец 2 =
        результат 3;
    }
    {
      образец 3 =
```

```
        результат цепочки
    }
  >
>
```

Одинаковыми цветами здесь показаны выражения с общим форматом — видно, что они располагаются по соседству. При выполнении цепочки функций, перечисленных в аргументе Seq, выражение как бы «просеивается» через эти функторы.

4.6 Пример кода, использующий функции высших порядков

Следующая функция осуществляет синтаксический анализ .ini-файла. Даётся в иллюстративных целях, к примеру, вопросы корректности имён ключей и содержимого параметров не рассматриваются. Определения идентификаторов и директивы $EXTERN для внешних функций опущены.

Полный текст примера можно найти в каталоге doc/example настоящего дистрибутива.

```
/*
  <LoadIniFile e.FileName>
    == t.IniFile t.ErrorMessage*

  t.IniFile ::= (t.Section*)
  t.Section ::= (s.LineNumber t.SectionName t.Parameter*)
  t.SectionName ::= #General | (e.Name)
  t.Parameter = (s.LineNumber (e.Name) e.Value)

  t.ErrorMessage ::=
    (s.LineNumber #BadLine) |
    (s.LineNumber #EmptyParameterName) |
    (s.LineNumber #ReassignParameter e.ParameterName)
*/
$ENTRY LoadIniFile {
  e.FileName =
    <Fetch
      <LoadFile e.FileName>
      <Seq
        // Нумеруем строки файла
        (MapReduce
          {
            s.LineNumber (e.Line) =
              <Inc s.LineNumber> (s.LineNumber e.Line);
          }
          1
        )
        DelAccumulator
        // Удаляем комментарии и начальные и концевые пробелы
        (Map {
          (s.LineNumber e.Line ';' e.Comment) = (s.LineNumber <Trim e.Line>);
          (s.LineNumber e.Line) = (s.LineNumber <Trim e.Line>);
        })
        // Удаляем пустые строки
        (Map {
          (s.LineNumber) = /* пусто */;
          (s.LineNumber e.Line) = (s.LineNumber e.Line);
        })
        // Парсим строки
        (Map {
          (s.LineNumber '[' e.SectionName ']') =
            (s.LineNumber #Section <Trim e.SectionName>);

          (s.LineNumber e.Parameter '=' e.Value) =
            (s.LineNumber #Parameter (<Trim e.Parameter>) <Trim e.Value>);

          (s.LineNumber e.Other) = (s.LineNumber #BadLine);
```

```
        })
        // Группируем параметры в секции
        (Reduce
          {
            (
              e.Sections-B (s.SectionPos (e.Section) e.Params) e.Sections-E
              (e.Errors)
            )
            (s.LineNumber #Section e.Section) =
              (
                e.Sections-B e.Sections-E (s.SectionPos (e.Section) e.Params)
                (e.Errors)
              );

            (e.Sections (e.Errors)) (s.LineNumber #Section e.Section) =
              (e.Sections (s.LineNumber (e.Section)) (e.Errors));

            (e.Sections (e.Errors)) (s.LineNumber #Parameter () e.Value) =
              (e.Sections
                (e.Errors (s.LineNumber #EmptyParameterName))
              );

            (e.Sections
              (s.SectionPos (e.LastSection)
                e.Params-B (s.ParamPos (e.Param) e.OldValue) e.Params-E
              )
              (e.Errors)
            )
            (s.LineNumber #Parameter (e.Param) e.NewValue) =
              (e.Sections
                (s.SectionPos (e.LastSection)
                  e.Params-B (s.ParamPos (e.Param) e.NewValue) e.Params-E
                )
                (e.Errors (s.LineNumber #ReassignParameter e.Param))
              );

            (e.Sections (s.SectionPos (e.LastSection) e.Params) (e.Errors))
            (s.LineNumber #Parameter (e.Param) e.Value) =
              (e.Sections
                (s.SectionPos (e.LastSection)
                  e.Params (s.LineNumber (e.Param) e.Value)
                )
                (e.Errors)
              );

            (e.Sections (e.Errors)) (s.LineNumber #BadLine) =
              (e.Sections (e.Errors (s.LineNumber #BadLine)));
          }
          ((1 #General) (/* ошибки */))
        )
        // Формируем результат функции
        {
          (e.Sections (e.Errors)) =
            (e.Sections) e.Errors;
        }
      >
    >;
}
```

5 Компилятор Простого Рефала: реализация

В дистрибутив компилятора входит довольно много файлов (их перечислению посвящён раздел 9), однако для разработки программ достаточно иметь только следующие пять: `srefc.exe` (на unix-like — `srefc`), `refalrts.h`, `refalrts.cpp`, `Library.cpp` и `LibraryEx.sref`:

- `srefc.exe` — исполнимый файл компилятора.
- `refalrts.h` — компонент рантайма. В `refalrts.h` находятся объявления функции и определения структур данных, используемых в сгенерированном коде на C++. Сгенерированный код всегда начинается с `#include "refalrts.h"`, поэтому путь к этому файлу должен быть в списке путей поисках заголовочных файлов компилятора C++ (обычно, опция **`-I`** компилятора).
- `refalrts.cpp` — компонент рантайма. `refalrts.cpp` содержит систему поддержки времени выполнения: определения функций, объявленных в `refalrts.h`, структуру данных поля зрения, реализацию абстрактной рефал-машины, менеджер памяти и функцию `main()` языка C++.
- `Library.cpp` — библиотека `Library`, см. 4.2.
- `LibraryEx.sref` — библиотека `LibraryEx`, см. 4.3.

5.1 Ключи командной строки, вызов компилятора

Синтаксис командной строки:

```
srefc [-c компилятор_C++] [ -d путь_поиска …] имя_модуля[.sref|.cpp]…
srefc @конфигурационный_файл
```

Опции:

- **`-c компилятор_C++`** — префикс командной строки вызова компилятора C++. Если эта опция отсутствует, то компилятор просто произведёт компиляцию в код на C++. Если данная опция присутствует, то после завершения компиляции всех исходных текстов, будет вызван компилятор C++ путём приписывания к префиксу всех имён .cpp-файлов: как являющихся результатом трансляции из .sref-файлов, так и .cpp-файлов, указанных в командной строке компилятора.
- **`-d путь_поиска`** (опция может повторяться) — задаёт путь поиска единиц трансляции, указанных в командной строке. По умолчанию поиск начинается с текущего каталога, затем во всех каталогах, указанных в опциях **`-d`**.
- **`имя_модуля[.sref|.cpp]`** — задаёт имя одной из единиц трансляции. Если расширение `.sref` или `.cpp` отсутствует, то сначала проверяется наличие файла с данным именем и расширением `.sref`, если такой файл отсутствует, то с расширением `.cpp`. Если файл не найден, продолжается поиск в следующем пути поиска.
- **`@конфигурационный_файл`** — файл, содержащий аргументы командной строки по одному на строчку. Поскольку опции **`-c`** и **`-d`** требуют после себя отдельного аргумента командной строки, они должны располагаться на отдельных строчках. Выглядит странно, возможно, в будущих версиях будет исправлено.

Рассмотрим пример. Пусть в текущем каталоге находятся файлы main.sref, http-parser.sref, `sockets.cpp` и `build.prj`, в каталоге `C:\Library` находятся файлы `Library.cpp`, `LibraryEx.sref`, `refalrts.h` и `refalrts.cpp`. Компилятор вызывается следующим образом:

```
srefc @build.prj
```

Файл `build.prj` содержит следующий текст:

```
-c
g++ -IC:\Library -omyprog.exe -lwininet
-d
C:\Library
main
http-parser.sref
sockets
Library
LibraryEx
refalrts
```

Префикс командной строки содержит вызов компилятора g++, со следующими опциями: путь поиска заголовочных файлов `C:\Library`, компиляция в файл `myprog.exe`, подключение библиотеки `wininet`. Путь поиска единиц трансляции — `C:\Library`. Единицы трансляции: `main`, `http-parser.sref`, `sockets`, `Library`, `LibraryEx`, `refalrts`.

Рассмотрим, как будет выполняться этот вызов компилятора:

1. Сначала компилятор произведёт поиск всех единиц трансляции:
- `main` — расширения нет, поэтому будет осуществляться поиск файла `main.sref` или `main.cpp` в каждом из каталогов поиска. Первый каталог поиска — текущий каталог. Компилятор ищет `main.sref` в текущем каталоге и находит его.
- `http-parser.sref` — расширение есть, значит будет осуществляться поиск только файла `http-parser.sref`. Файл обнаруживается в первом каталоге поиска — в текущем каталоге.
- `sockets` — расширения нет. В первом каталоге поиска — текущем каталоге файл `sockets.sref` не обнаруживается, зато обнаруживается `sockets.cpp`. Простой Рефал компилировать его не будет, однако файл будет передан компилятору C++.
- Library — расширения нет. В текущем каталоге файлы `Library.sref` и `Library.cpp` отсутствуют. В следующем каталоге поиска — в C:\Library отсутствует файл `Library.sref`, но присутствует `Library.cpp`. Файл `C:\Library\Library.cpp` будет передан компилятору C++.
- `LibraryEx` — расширения нет. В текущем каталоге файлы `LibraryEx.sref` и `LibraryEx.cpp` отсутствуют. В следующем каталоге поиска — в `C:\Library` присутствует файл `LibraryEx.sref`. Файл `C:\Library\LibraryEx.sref` будет откомпилирован Простым Рефалом, получившийся файл будет передан компилятору C++.
- `refalrts` — его поиск аналогичен поиску `Library`.
2. Затем компилятор переведёт файлы `main.sref`, `http-parser.sref` и `C:\Library\LibraryEx.sref` на C++.
3. На третьем этапе файлы `sockets.cpp`, `C:\Library\Library.cpp`, `C:\Library\refalrts.cpp` и файлы, получившиеся в результате трансляции на предыдущем этапе, будут переданы компилятору C++ путём формирования командной строки из префикса и имён файлов.

5.2 Макросы препроцессора C++

Опции командной строки, перечисленные в предыдущем разделе, не допускают тонкой настройки кодогенерации и поведения во время выполнения (типа оптимизаций или средств отладки). Однако, некоторые такие возможности реализация предоставляет — для управления ими используются макросы препроцессора C++. Для глобальной установки макроса компиляторы C++, как правило, используют опцию `-D`, после которой указывается либо просто имя определяемого макроса (если макрос затем проверяется директивой #ifdef), либо имя и значение, разделённые знаком =. Пример:

```
g++ -otest.exe -DTARGET_WINDOWS -DPAGE_SIZE=4096 test.cpp
```

Здесь устанавливаются макросы `TARGET_WINDOWS` и `PAGE_SIZE`, причём последний получает значение `4096`.

Сгенерированный код поддерживает только один макрос — **`INTERPRET`**. Код построения правых частей предложений транслируется в C++ двумя разными способами: в режиме прямой кодогенерации (код представляет собой последовательность вызовов функций и проверок возвращаемого значения) и интерпретации (для правой части генерируется массив команд и вызов функции интерпретации этого массива). В целевом файле присутствуют оба варианта, по умолчанию выполняется код, построенный первым способом, для компиляции второго варианта следует использовать уже упомянутый макрос `INTERPRET`. Исполнимый файл, скомпилированный в режиме интерпретации, выполняется медленнее приблизительно на 15 % и имеет размер приблизительно на 30 % меньше, чем в режиме прямой кодогенерации.

Рантайм поддерживает конфигурацию несколькими макросами, в основном это включение/выключение отладочных функций. Здесь они будут только перечислены, об их использовании будет рассказываться в следующем разделе:

- **DUMP_FILE** — определяет имя файла, в который будет осуществляться отладочный вывод, параметр должен быть строковым литералом языка С. Пример:

  ```
  -DDUMP_FILE="__dump.txt"
  ```

- **SHOW_DEBUG** — целочисленный параметр, определяет номер шага, после которого на каждом шаге рефал-машина будет выполнять дамп поля зрения.
- **DUMP_FREE_LIST** — когда определён, печатает не только дамп поля зрения, но и дамп списка свободных узлов. Может быть полезен при отладке внешних функций и рантайма.
- **MEMORY_LIMIT** — целочисленный параметр. Если определён, то при попытке распределить память для узлов больше, чем указано в параметре, рефал-машина аварийно остановится с сообщением о нехватке памяти.
- **MODULE_REFAL** — этот же рантайм используется компилятором Модульного Рефала, однако, используемые Модульным Рефалом структуры данных незначительно отличаются от структур Простого Рефала. При компиляции в Простом Рефале определять не нужно.
- **DONT_PRINT_STATISTICS**. При завершении программа по умолчанию выдаёт (на `stderr`) статистику о времени своей работы: общие затраты времени и памяти, а также затраты времени на выполнение отдельных видов вычислений (внешние функции, левые, правые части и т.д.). Макрос отключает вывод этой статистики.

5.3 Утилита srmake

Для компиляции довольно больших проектов довольно утомительно перечислять в командной строке (либо в файле `.prj`) компилятора все необходимые единицы трансляции. В частности, при добавлении нового файла также придётся править сценарий сборки, либо файл с опциями командной строки. А хотелось бы иметь возможность обнаруживать все требуемые файлы автоматически.

Для этой цели была написана утилита `srmake` — она в качестве параметра получает имя одного из исходных файлов, находит все зависимые и вызывает компилятор Простого Рефала для сборки всего проекта. Но для этого в файлах проекта надо указывать, от каких других файлов они зависят — для этого используется специальный комментарий `//FROM`. Синтаксис следующий:

```
//FROM ИмяФайла
```

Расширение у файла (`.cpp` или `.sref`) можно опустить. Утилита `srmake` сканирует предложенный файл в поисках таких комментариев, затем ищет файлы, указанные после `//FROM`, рекурсивно сканирует и их, до тех пор, пока все необходимые единицы трансляции не найдутся. Утилита `srmake` сканирует не только файлы на Простом Рефале, но и файлы на C++, поэтому допустимо писать комментарий `//FROM` и внутри них.

Хорошей практикой является помещение комментария `//FROM` перед директивой `$EXTERN` — не только сообщает утилите `srmake` о зависимости, но и повышает читабельность кода:

```
//FROM Library
$EXTERN WriteLine;

//FROM LibraryEx
$EXTERN Inc, Dec, LoadFile;

//FROM socket
$EXTERN Accept, Bind, Connect, Listen;
```

В LibraryEx есть ссылка на Library, в Library есть ссылка на refalrts, поэтому если используются стандартные библиотеки, то модуль рантайма также будет найден. Если ваш проект прямо или косвенно не использует Library, то ссылку на refalrts нужно будет добавить вручную.

Синтаксис командной строки утилиты srmake напоминает синтаксис самого компилятора srefc:

```
srmake [-s компилятор_srefc] -c компилятор_C++ [ -d путь_поиска … ] имя_модуля[.sref|.cpp]
```

Опции:

- **-s компилятор_srefc** — путь к компилятору Простого Рефала. Если опция не указана, то вызывается srefc, т. е. предполагается, что компилятор имеет имя srefc.exe (srefc на unix-like) и находится в текущем каталоге (только на Windows) или указан в переменной PATH.
- **-c компилятор_C++** — смысл тот же, что и для компилятора Простого Рефала.
- **-d путь_поиска** — смысл тот же, что и для компилятора Простого Рефала.
- **имя_модуля[.sref|.cpp]** — имя одного из файлов исходного текста, параметр должен быть единственным.

6 Отладка программ на Простом Рефале

Программ без ошибок не бывает, поэтому хорошая реализация языка программирования должна иметь средства, облегчающие поиск и исправление ошибок. В этом разделе мы расскажем о способах отладки программ на Простом Рефале.

6.1 Использование отладчика C++

Компилятор Простого Рефала порождает код на C++, поэтому первое, что может прийти в голову программисту — использовать отладчик для C++. Такой вариант возможен, но он имеет ряд недостатков:

- Отладка ведётся на низком уровне — на уровне сопоставления отдельных элементов выражения в образце (атомов, скобок), построения отдельных элементов результата.
- Поле зрения представляет собой двусвязный список, поэтому исследовать его в отладчике очень неудобно.
- Сгенерированные функции имеют большой объём, что снижает наглядность при чтении кода.
- Программисту необходимо понимать, как компилятор преобразует код на Рефале в C++.

Однако, этот метод имеет свою сферу применения: отладка внешних функций и рантайма.

6.2 Средства отладки рантайма

Пошагового отладчика для Простого Рефала нет (возможно, пока нет). Зато есть посмертный отладчик и трассировщик.

При аварийном останове программы (при невозможности сопоставления, либо при недостатке памяти) рантайм осуществляет выдачу дампа всего поля зрения, по умолчанию вывод осуществляется в stderr. В этом случае выводятся:

- причина ошибки:
  - RECOGNITION IMPOSSIBLE — невозможность сопоставления,
  - NO MEMORY — недостаточно памяти,
- номер шага, на котором произошла ошибка,
- вызов функции, приведший к ошибке,
- дамп всего поля зрения.

Наиболее распространённая ошибка в программе — это невозможность сопоставления, и такого отладочного вывода, как правило, достаточно для локализации и исправления ошибки. Однако, если

по-прежнему остаётся непонятно, каким образом сформировалось ошибочное поле зрения, можно пройти по шагам и получить последовательность дампов до аварийного останова. Для этого надо установить макрос препроцессора `SHOW_DEBUG`, присвоив ему целое число — номер шага, начиная с которого надо выводить дамп.

Дамп поля зрения, как правило, достаточно объёмен, и читать его с консоли неудобно. Можно, конечно, его перенаправить в файл средствами оболочки (`prog 2>err.txt`), но, чтобы постоянно так не писать, можно установить другой макрос — `DUMP_FILE`, присвоив ему имя файла для отладочного вывода (`-DDUMP_FILE="err.txt"`).

Другой вид проблем — исчерпание свободной памяти. Оно может произойти по двум причинам: либо программа требует очень много памяти (более 2 Гбайт на 32-разрядных системах и более суммарного объёма ОЗУ и файла подкачки на 64-разрядных системах), либо программа зацикливается и на каждой итерации добавляет новые данные в поле зрения. В первом случае нужно либо поискать более оптимальный алгоритм, либо не подавать на вход такие объёмные задачи. Во втором случае — очевидно, исправлять ошибку.

При невозможности выделить память рантайм, как и в случае невозможности сопоставления, аварийно прерывает программу и выводит дамп. Но лучше до этого не доводить. Во-первых, из-за потребления памяти компьютер начнёт тормозить и свопить, что только замедлит процесс, во-вторых, дамп огромного поля зрения будет огромен и выводиться будет очень долго.

Для удобства отладки памяти предусмотрен макрос `MEMORY_LIMIT`, которому следует присвоить целое число — максимально допустимый объём памяти в узлах. Один узел поля зрения хранит в себе либо один атом, либо одну скобку; на 32-разрядной системе узел имеет объём 20 байт, на 64-разрядной — 40 байт. При установке этого макроса программа будет падать не при исчерпании свободной памяти (когда `malloc()` вернёт `NULL`), а при попытке распределить узлов больше, чем значение макроса. На практике `MEMORY_LIMIT` в несколько миллионов хватает за глаза (компилятор Модульного Рефала откомпилирован с `-DMEMORY_LIMIT=7654321`).

6.3 Идиомы отладки

Встроенный трассировщик может делать только дампы поля зрения целиком и на каждом шаге рефал-машины. Но часто бывает необходимо выводить только значения некоторых переменных при каждом вызове некоторой функции. И здесь приходится пользоваться хорошо известным среди программистов примитивным методом отладки — отладочной печатью — вставкой операторов вывода на консоль в разные части программы:

```
$FORWARD DoFib;

Fibonacci {
  1 = 1;
  s.N = <DoFib 2 s.N 1 1>;
}

$EXTERN WriteLine, Inc, Add;

DoFib {
  s.N s.N s.Prev s.Cur = s.Cur;

  s.K s.N s.Prev s.Cur =
    <WriteLine 's.Prev = ' s.Prev 's.Cur = ' s.Cur>  // Отладочная печать
    <DoFib
      <Inc s.K> s.N s.Cur <Add s.Prev s.Cur>
    >;
}
```

Недостаток такого «лобового» метода вставки отладочного вывода в том, что трассируется только одно предложение, а иногда надо выводить фрагменты формата функции для любого вызова. Конечно, можно добавить отладочный вывод в каждое из предложений функции, но проще использовать нижеприведённую идиому, идиому косвенного вызова.

Как известно, большинство проблем в информатике можно решить введением дополнительного уровня косвенности (кроме проблем, вызванных большим числом уровней косвенности ☺). Здесь для добавления отладочной печати мы добавим дополнительную промежуточную функцию.

Допустим, мы хотим распечатать значения переменных функции `DoFib` на каждой итерации. Для этого:

1. Переименуем функцию в `DoFib_Debug`. Следует переименовывать только саму функцию, менять имя в точках вызова самой функции (в том числе и в рекурсивных) не надо.

```
DoFib_Debug {
  s.N s.N s.Prev s.Cur = s.Cur;

  s.K s.N s.Prev s.Cur =
    <DoFib
      <Inc s.K> s.N s.Cur <Add s.Prev s.Cur>
    >;
}
```

2. Добавим перед `DoFib_Debug` новую функцию `DoFib`, которая имеет тот же формат, что и отлаживаемая и просто вызывает `DoFib_Debug`. Если отлаживаемая функция имела модификатор `$ENTRY`, то новая функция также должна иметь модификатор `$ENTRY` (а у отлаживаемой функции модификатор можно убрать)

```
$FORWARD DoFib_Debug;

DoFib {
  s.K s.N s.Prev s.Cur =
    <DoFib_Debug s.K s.N s.Prev s.Cur>;
}

DoFib_Debug {
  s.N s.N s.Prev s.Cur = s.Cur;

  s.K s.N s.Prev s.Cur =
    <DoFib
      <Inc s.K> s.N s.Cur <Add s.Prev s.Cur>
    >;
}
```

   Заметим, что на этом этапе мы имеем корректно работающую программу: все обращения к `DoFib` промежуточная функция перенаправляет к отлаживаемой, которая выполняет те же действия, что и раньше.
3. В промежуточную функцию теперь можно добавлять любую отладочную печать, любые проверки аргументов на корректность и так далее:

```
$FORWARD DoFib_Debug;
$EXTERN Compare;

DoFib {
  s.K s.N s.Prev s.Cur =
    // Это отладочная печать:
    <WriteLine '<DoFib>:'>
    <WriteLine '  s.K = ' s.K>
    <WriteLine '  s.N = ' s.N>
```

```
    <WriteLine '  s.Prev = ' s.Prev>
    <WriteLine '  s.Cur = ' s.Cur>
    <WriteLine>
    // Это assert'ы:
    <{ '<' = ; '=' = ; } <Compare s.K s.N>>
    <{ '<' = ; '=' = ; } <Compare s.Prev s.Cur>>
    // Это косвенный вызов «исследуемой» функции
    <DoFib_Debug s.K s.N s.Prev s.Cur>;
}

DoFib_Debug {
  s.N s.N s.Prev s.Cur = s.Cur;

  s.K s.N s.Prev s.Cur =
    <DoFib
      <Inc s.K> s.N s.Cur <Add s.Prev s.Cur>
    >;
}
```

4. Чтобы убрать отладочную печать, достаточно удалить строки между `DoFib {` и `DoFib_Debug {` включая последнюю, а также `$FORWARD DoFib_Debug;`.

```
$FORWARD DoFib_Debug;
$EXTERN Compare;

DoFib {
  s.K s.N s.Prev s.Cur =
    // Это отладочная печать:
    <WriteLine '<DoFib>:'>
    <WriteLine '  s.K = ' s.K>
    <WriteLine '  s.N = ' s.N>
    <WriteLine '  s.Prev = ' s.Prev>
    <WriteLine '  s.Cur = ' s.Cur>
    <WriteLine>
    // Это assert'ы:
    <{ '<' = ; '=' = ; } <Compare s.K s.N>>
    <{ '<' = ; '=' = ; } <Compare s.Prev s.Cur>>
    // Это косвенный вызов «подопытной» функции
    <DoFib_Debug s.K s.N s.Prev s.Cur>;
}

DoFib_Debug {
  s.N s.N s.Prev s.Cur = s.Cur;

  s.K s.N s.Prev s.Cur =
    <DoFib
      <Inc s.K> s.N s.Cur <Add s.Prev s.Cur>
    >;
}
```

Ошибки, как известно, должны проявляться как можно раньше. Для раннего обнаружения ошибок принято использовать утверждения — конструкции языка программирования, которые проверяют некоторые данные, ничего не делают в случае корректных данных и аварийно завершают программу в случае некорректных. В случае Простого Рефала утверждения могут представлять собой функции, которые не возвращают ничего в случае корректных данных на входе и приводить к автосту в случае некорректных, причём описываться они могут прямо по месту вызова. Пример утверждений можно видеть на листингах кода выше.

Простой Рефал — динамически типизируемый язык, а это значит, что ошибки несоответствия типов можно поймать только во время выполнения. Для раннего обнаружения ошибок, связанных с некорректным использованием АТД-термов, их родные модули могут экспортировать функции-утверждения, проверяющие, что данный терм является АТД-термом данного вида.

И, напоследок, пару советов, как можно упростить отладку программы. Пишите максимально подробные образцы и используйте АТД-термы для абстрактных типов данных.

Иногда возникает соблазн заменить часть некоторого образца на переменную, особенно в последнем предложении (ведь все частные случаи перехвачены предложениями выше, не так ли). С одной стороны хочется сократить время набора текста программы, с другой стороны — (преждевременно) оптимизировать — кажется, что на разбор недифференцированной переменной тратится меньше ресурсов, чем на конкретное значение. Не делайте так — поступая таким образом, вы лишаете программу дополнительного барьера от ошибки. Во-первых, нет гарантии, что при вызове в функцию передаётся корректное значение — в таком случае ошибка может быть поймана позже, либо не поймана вообще — программа не сломается, но может выдавать неправильные результаты. Во-вторых, это касается случая последнего предложения, нет гарантии, что предыдущие образцы перехватывают все возможные варианты.

Но не стоит и перегибать палку — писать слишком подробные образцы, которые нарушают инкапсуляцию. Если в интерфейсе некоторой функции заявлено, что она возвращает терм, но мы знаем, что в текущей реализации этот терм всегда скобочный, то не надо явно этот скобочный терм прописывать. Ведь интерфейс может измениться (функция будет возвращать атом или АТД-терм) и тогда придётся править все точки вызова данной функции.

Использование АТД термов по назначению, то есть для реализации абстрактных типов данных очень хорошо работает в сочетании с предыдущим советом и, в то же время, препятствует упомянутому злоупотреблению им. С одной стороны, если программист регулярно использует скобочные термы и s-переменные для, соответственно, составных термов и атомов (а не более общие t-переменные), то при попытке сопоставить с ними ошибочно переданный АТД-терм мы получим ошибку сопоставления. С другой стороны, вне родного модуля АТД-терм может быть сопоставлен только с t-переменной. Ну и, конечно, при попытке вызвать функцию-метод доступа к абстрактному типу данных с чем-то кроме ожидаемого АТД-терма приведёт к закономерному аварийному завершению программы.

7 Интерфейс с языком C++

7.1 Вычислительная модель

8 Установка компилятора

9 Структура каталогов дистрибутива

От: Александр Коновалов
Отправлено: пятница, 3 октября 2014 г. 10:46
Кому: Constantine Belev

Thu, 02 Oct 2014 23:49:29 +0400 от Constantine Belev <const.belev@yandex.ru>:

Здравствуйте! Прочитал статью Сергея Юрьевича на тему модернизированного векторного представления объектных выражений. Там он также расписывает структуры данных, а также набор инструкций (это, как я понимаю, то, во что будет компилироваться Рефал-программа в данном случае). Посмотрел слайды по простому рефалу, про библиотеку времени выполнения, которая реализует объектные выражения на двунаправленных списках, и компилируется в c++ код. Насколько я понял, моя задача как раз и состоит в написании похожей библиотеки, только с rope-ами. Так вот, я пока пытаюсь понять, где что находится у компилятора Простого Рефала (Кстати, а под unix-системами он запустится? ;-)). Пока пара вопросов нарисовалась.

В статье Скоробогатова нам интересна только первая часть — сравнение различных промежуточных представлений.
Набор инструкций довольно сильно отличается от нашего, поскольку он написан для диалекта Refal-7 (http://iu9.bmstu.ru/science/refal.php).

Где что находится.

- **bootstrap/** — скомпилированные исходные тексты на Си++, откомпилировав их, можно получить exe-шник.
- **Compiler/** — исходные тексты самого компилятора.
  - srefc.sref — главный файл, содержащий функцию Go.
  - Lexer.sref — лексический анализатор.
  - Driver.sref — синтаксический анализатор.
  - Algorithm.sref — генерирует последовательность императивных команд для предложения.
  - Generator.sref — генерирует код на Си++ по последовательности императивных команд.
  - SymTable.sref, Context.sref — таблицы имён функций и переменных.
  - Прочие файлы выполняют служебные функции.
- **LexGen/** — исходные тексты генератора лексера (он нам не интересен).
- **SRLib/** — библиотеки:
  - времени выполнения — refalrts.h, refalrts.cpp,
  - примитивных функций (ввод-вывод, арифметика и др.) — Library.cpp,
  - полезных функций на Рефале — LibraryEx.sref (LibraryEx.cpp).

- **SRMake/** — утилита, подбирающая все файлы исходного текста и запускающая их компиляцию (нам не интересна).
- **srprep/** — препроцессор, отображающий Простой Рефал в Модульный Рефал, использовался на нулевом шаге раскрутки, нам не нужен.
- **Тесты и заготовки/** — тесты, использовавшиеся при разработке и заготовки для примитивных функций.
- **note00?.txt** — внутренняя документация, почитать полезно.

То, что нам потребуется изменять, я подчеркнул.

Под UNIX он запускается. Проверялось под QNX и разными дистрибутивами Linux (для архитектур x86 и x64).
Для этого надо собрать exe-шник из исходных файлов в каталоге bootstrap.

Первое - важно ли то, на чем будет написана библиотека времени выполнения (я к вопросу о том, что можно попробовать написать её на Scala/Go)? Поправьте меня, если это так. Может быть, там вообще компилятор на Рефале как раз-таки и написан, и мне для моего же блага писать нужно будет на нём =)

Не важно. Библиотека времени выполнения переписывалась и на C#. Scala я не знаю, поэтому не могу сказать,
насколько удобно будет писать генератор кода и библиотеку для неё. Для Go можно попробовать. Рефал — малоизвестный язык, вряд ли на этих языках или для этих языков есть его реализации, по крайней мере, я о них ничего не слышал.

Насчет второго, уже не припомню, пока буду читать документацию и мануалы, так просветление быстрее должно наступить.

С уважением, Константин Белёв.

==========
Александр Коновалов aka Маздайщик

10 Список литературы

Алешин, А. Ю., Красовский, А. Г., Романенко, С. А., & Шерстнев, В. Ю. (1991). *Система программирования РЕФАЛ-2 для IBM PC, PDP-11 и VAX-11. Руководство пользователя.* Москва.

Немытых, А. П. (2014). Лекции по языку программирования РЕФАЛ. В *Сборник трудов по функциональному языку программирования Рефал* (Т. 1, стр. 118-165). Переславль-Залесский: Изд-во "Сборник". Получено из http://refal.botik.ru/library/refal2014_issue-I.pdf

Скоробогатов, С. Ю., & Чеповский, А. М. (2006). Язык Refal с функциями высших порядков. *Информационные технологии*(№ 9). Получено из http://iu9.bmstu.ru/science/refal.pdf

Турчин, В. Ф. (1989). *Фрейм: РЕФАЛ-5. Руководство и справочник.* Получено 16 апреля 2015 г., из Содружество «РЕФАЛ/Суперкомпиляция»: http://refal.ru/rf5_frm.htm

Определение идентификатора помещает перечисленные имена в область видимости как имена идентификаторов. Если в области видимости уже присутствует одно из указанных имён как имя идентификатора, то сигнализируется синтаксическая ошибка.

Определение идентификаторов компилируется в набор конструкций вида

```
//$LABEL ИмяИдентификатора
template <typename T>
struct ИмяИдентификатора {
  static const char *name() {
    return "ИмяИдентификатора";
  }
};
```

по одной для каждого из имён после директивы `$LABEL`. Обращение к идентификатору осуществляется конструкцией вида `& ИмяИдентификатора<int>::name`.

Как это работает. Определение идентификатора есть определение шаблонного класса, обращение к нему — конкретизация шаблона некоторым типом (тип не имеет значения, важно только всегда использовать один и тот же, в компиляторе используется `int`). При конкретизации шаблона класса компилятор C++ порождает код метода этого класса. В практике программирования на C++ библиотеки шаблонов размещаются в заголовочных файлах, в результате чего скомпилированный код для одних и тех же шаблонов с одинаковыми параметрами-типам многократно дублируется в объектных файлах. Компоновщик C++ обнаруживает такие случаи и в целевой исполнимый файл помещает только единственный экземпляр данного кода. Возвращаясь к Рефалу. Если в нескольких разных единицах трансляции присутствует определение одного и того же идентификатора, то компилятор C++ породит в соответствующих объектных файлах идентичный код для функции `ИмяИдентификатора<int>::name()`, компоновщик устранит дублирующийся код и в целевой исполнимый файл будет помещён единственный код для этой функции. Соответственно, указатели на эту функцию из других функций из разных единиц трансляции будут идентичны.

Если вы ничего не поняли, относитесь к этому как к магии.