Простой Рефал. Руководство пользователя

Оглавление

[1 Язык Простой Рефал: синтаксис и семантика 2](#_Toc527398243)

[1.1 Структура программы 2](#_Toc527398244)

[1.2 Программные элементы (объявления и определения) 2](#_Toc527398245)

[1.2.1 Объявления функций ($FORWARD и $EXTERN) 2](#_Toc527398246)

[1.2.2 Регулярные функции 2](#_Toc527398247)

[1.3 Функции, семантика и сравнение с РЕФАЛом-5 2](#_Toc527398248)

[1.3.1 Синтаксис функций, общее с РЕФАЛом-5 2](#_Toc527398249)

[2 Библиотека функций 3](#_Toc527398250)

[2.1 Расширенная библиотека (LibraryEx) 3](#_Toc527398251)

[2.2 LibraryEx: надстройки над функциями базовой библиотеки и другие полезные функции 3](#_Toc527398252)

[2.2.1 LoadFile 3](#_Toc527398253)

[2.2.2 SaveFile 3](#_Toc527398254)

[2.2.3 Inc, Dec 3](#_Toc527398255)

[2.2.4 ArgList 3](#_Toc527398256)

[2.2.5 Trim 4](#_Toc527398257)

[2.3 LibraryEx: функции высших порядков 4](#_Toc527398258)

[2.3.1 Понятие функтора, Apply 4](#_Toc527398259)

[2.3.2 Map 5](#_Toc527398260)

[2.3.3 Reduce 5](#_Toc527398261)

[2.3.4 MapReduce 5](#_Toc527398262)

[2.3.5 DelAccumulator 6](#_Toc527398263)

[3 Компилятор Простого Рефала: реализация 6](#_Toc527398264)

[3.1 Ключи командной строки, вызов компилятора 6](#_Toc527398265)

[3.2 Макросы препроцессора C++ 8](#_Toc527398266)

[4 Отладка программ на Простом Рефале 9](#_Toc527398267)

[4.1 Использование отладчика C++ 9](#_Toc527398268)

[4.2 Средства отладки рантайма 9](#_Toc527398269)

[4.3 Идиомы отладки 10](#_Toc527398270)

[5 Интерфейс с языком C++ 13](#_Toc527398271)

[5.1 Вычислительная модель 13](#_Toc527398272)

[5.2 Написание внешних функций 13](#_Toc527398273)

[5.2.1 Быстрый и грязный способ 13](#_Toc527398274)

[5.2.2 Написание функции вручную 13](#_Toc527398275)

[6 Список литературы 13](#_Toc527398276)

# Язык Простой Рефал: синтаксис и семантика

## Структура программы

## Программные элементы (объявления и определения)

### Объявления функций ($FORWARD и $EXTERN)

…

Объявления функций на Рефале компилируются в соответствующие объявления функций на C++, $FORWARD — с модификатором static, $EXTERN — с модификатором extern.

Пример:

$FORWARD A, B, C;

$EXTERN E, F, G;

Компилируется в:

static refalrts::FnResult A(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

static refalrts::FnResult B(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

static refalrts::FnResult C(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

extern refalrts::FnResult E(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

extern refalrts::FnResult F(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

extern refalrts::FnResult G(refalrts::Iter arg\_begin, refalrts::Iter arg\_end);

### Регулярные 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++.

## Функции, семантика и сравнение с РЕФАЛом-5

…

### Синтаксис функций, общее с РЕФАЛом-5

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

* …
* *(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).

…

# 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)).

## 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.

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

### 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.

### SaveFile

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

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

### 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>.

### 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.

### Trim

<Trim e.String> == e.String’

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

## LibraryEx: функции высших порядков

### 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)>[[1]](#footnote-1)  
// == ('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) =

<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.

### 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.

### 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)

### 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.

### 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.

# 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.

## 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.

1. Затем компилятор переведёт файлы main.sref, http-parser.sref и C:\Library\LibraryEx.sref на С++.
2. 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.

## 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 C. 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.

# 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.

## 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.

## 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).

## 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>

>;

}

1. 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.

1. 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>

>;

}

1. 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.

# Интерфейс с языком C++

## Вычислительная модель

## Написание внешних функций

### Быстрый и грязный способ

### Написание функции вручную

# 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 *Cборник 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

1. Нотация, используемая в комментарии, объясняется в разделе 4.1. [↑](#footnote-ref-1)