Простой Рефал. Руководство пользователя

Оглавление

[1 История диалекта 1](#_Toc417578249)

[2 Начинающим 2](#_Toc417578250)

[3 Язык Простой Рефал: синтаксис и семантика 2](#_Toc417578251)

[3.1 Структура программы 2](#_Toc417578252)

[3.2 Программные элементы (объявления и определения) 3](#_Toc417578253)

[3.2.1 Основные понятия 3](#_Toc417578254)

[3.2.2 Объявления функций ($FORWARD и $EXTERN) 4](#_Toc417578255)

[3.2.3 Идентификаторы ($LABEL) 5](#_Toc417578256)

[3.2.4 Пустые функции ($ENUM и $EENUM) 5](#_Toc417578257)

[3.2.5 Статические ящики ($SWAP и $ESWAP) 6](#_Toc417578258)

[3.2.6 Функции 7](#_Toc417578259)

[3.3 Разделы про функции 7](#_Toc417578260)

[4 Библиотека функций 7](#_Toc417578261)

[4.1 Используемые обозначения 7](#_Toc417578262)

[4.2 Базовая библиотека (Library) 7](#_Toc417578263)

[4.3 Расширенная библиотека (LibraryEx) 7](#_Toc417578264)

[5 Компилятор Простой Рефал: особенности реализации 7](#_Toc417578265)

[6 Интерфейс с языком C++ 7](#_Toc417578266)

[6.1 Вычислительная модель 7](#_Toc417578267)

[6.2 Написание внешних функций 7](#_Toc417578268)

[6.2.1 Быстрый и грязный способ 7](#_Toc417578269)

[6.2.2 Написание функции вручную 7](#_Toc417578270)

[7 Установка компилятора 7](#_Toc417578271)

[8 Известные ошибки 7](#_Toc417578272)

[9 Список литературы 7](#_Toc417578273)

# История диалекта

РЕФАЛ — РЕкурсивный Функциональный АЛгоритмический язык[[1]](#footnote-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]](#footnote-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).

# 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.

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

# Язык Простой Рефал: синтаксис и семантика

## Структура программы

Программа на Простом Рефале (далее для краткости — на Рефале) состоит из набора единиц трансляции — исходных файлов, написанных на Рефале и C++, 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** — объявлений и определений программных сущностей, независимо транслируемых в соответствующие программные конструкты на C++.

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++). Однострочные 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 программная сущность 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).

## Программные элементы (объявления и определения)

### Основные понятия

**Программные элементы** делятся на две категории: объявления и определения программных сущностей. **Программные сущности** делятся на функции и идентификаторы. **Объявления** служат для уведомления компилятора о том, что где-то (в текущем файле ниже по тексту программы или в другой единице трансляции) определена программная сущность с заданным именем и заданными свойствами (вид сущности, тип доступа). **Определения**, собственно, определяют данную программную сущность. **Функции**, как правило, содержат исполнимый код. **Идентификаторы** пополняют пространство атомов (рефал-символов) определёнными пользователем атомами-идентификаторами. В единице трансляции может быть определено не более одной функции и одного идентификатора с заданным именем.

Каждая функция имеет тип доступа. Функции с **локальным** типом доступа видимости (локальные функции) доступны только в том файле, где они определены, в различных единицах трансляции могут существовать локальные функции с одинаковыми именами. На функции с **глобальным** типом доступа видимости (т. н. **entry-функции**) можно ссылаться из других единиц трансляции при помощи директивы (ключевого слова) $EXTERN. Entry-функции должны иметь уникальные имена в рамках всей программы. *Примечание:* имя локальной функции в одной из единиц трансляции может совпадать с именем entry-функции из другой единицы при условии, что в первой из них entry-функция не импортируется директивой $EXTERN. Для идентификаторов понятие типа доступа не определено: в различных единицах трансляции могут быть определения одноимённых идентификаторов (это роднит идентификаторы с локальными функциями), но при этом все эти определения определяют один и тот же идентификатор на всю программу (это роднит их с entry-функциями), т. е. определения идентификаторов можно трактовать как их объявления идентификаторов, определённых в некоторой вымышленной единице трансляции, содержащей все возможные идентификаторы.

В целевом коде на C++ локальные функции соответствуют функциям, определённым с модификатором static, entry-функции — функциям без этого модификатора. Идентификаторы компилируются в довольно хитрую конструкцию, о которой будет рассказано в соответствующем разделе.

Под **пространством имён** будем понимать множество имён функций (с атрибутами области типа доступа и определённости), идентификаторов и переменных (про переменные будет рассказано далее), доступных в данной точке программы.

Синтаксис программного элемента:

ProgramElement =

FunctionDeclaration

| IdentifierDefinition

| EnumDefinition

| SwapDefinition

| FunctionDefinition

| ";".

*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.

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

Синтаксис:

FunctionDeclaration = ( "$FORWARD" | "$EXTERN" ) NameList.

NameList = NAME { "," NAME } ";".

Объявление функций помещает перечисленные имена в пространство имён как локальные функции (если указана директива $FORWARD), либо как entry-функции (если указана директива $EXTERN). Если в пространстве имён уже присутствует одно из указанных имён как имя функции, но имеет другой тип доступа, то сигнализируется синтаксическая ошибка.

Директива $FORWARD необходима при определении взаимно рекурсивных локальных функций. Кроме того, читаемость программ часто улучшается, если определение функции не предшествует, а следует за её использованием (особенно, при написании парсера методом рекурсивного спуска). Данная директива позволяет придерживаться такого стиля программирования.

Директива $EXTERN почти всегда используется для ссылки на entry-функцию, определённую в другой единице трансляции. Однако может быть полезна при обращении к entry-функции, определённой в той же единице трансляции, но ниже по тексту программы.

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);

### Идентификаторы ($LABEL)

Синтаксис:

IdentifierDefinition = "$LABEL" NameList.

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 С++ 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.

### Пустые функции ($ENUM и $EENUM)

Синтаксис:

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

Определение пустых функций помещает перечисленные имена в пространство имён как определённые имена локальных (директива $ENUM) или entry-функций (директива $EENUM — **e**ntry **enum**). Если в пространстве имён уже присутствует одно из этих имён как определённая функция или как функция с другим типом доступа, то сигнализируется синтаксическая ошибка.

Функция, определённая таким образом, является пустой, её тело не имеет ни одного предложения (см. далее про синтаксис функций). Вызов такой функции всегда приводит к авосту (аварийному останову) — невозможности сопоставления.

Код

$ENUM L;

$EENUM E;

идентичен следующему (см. далее синтаксис функций):

L { /\* пусто \*/ }

$ENTRY E { /\* пусто \*/ }

Данная конструкция устаревшая и не рекомендуется для дальнейшего применения (**deprecated**) в качестве замены идентификаторов. Первоначально в Простом Рефале отсутствовали идентификаторы, вместо них использовались пустые функции. Они предназначались для той же цели, что и идентификаторы, а именно создание новых атомов с описательными именами — в качестве таких атомов использовались пустые функции. Сейчас есть более удобное средство для этой цели — собственно сами идентификаторы, рекомендуем использовать именно их. Однако, пустые функции встречаются в старом коде (в том числе и в исходном тексте самого компилятора), а также в стандартной библиотеке (которую надо в этом плане актуализировать), поэтому знать эту конструкцию надо. По сравнению с идентификаторами, пустые функции имеют следующие недостатки:

1. При использовании $EENUM необходимо следить, чтобы функция была определена только в одной единице трансляции, а в других она импортировалась как $EXTERN. Данная асимметрия затрудняет программирование.
2. При использовании транслятора MS Visual C++ 2013 и установки наивысшей оптимизации компилятор устраняет дублирующиеся определения функций, даже с разными именами. В результате все атомы пустых функций (поскольку их тело полностью идентично) становятся неразличимыми, что противоречит их предназначению. (Возможно, эта ошибка будет исправлена в дальнейшем.)

Единственное оправдание использования пустых функций — как метки для абстрактных типов данных (см. далее).

*Примечание.* Идея пустых функций заимствована из РЕФАЛа-2 (Алешин, и др., 1991).

### Статические ящики ($SWAP и $ESWAP)

Синтаксис:

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

Определение статических ящиков помещает перечисленные имена в пространство имён как определённые имена локальных (директива $SWAP) или entry-функций (директива $ESWAP). Если в пространстве имён уже присутствует одно из этих имён как определённая функция или как функция с другим типом доступа, то сигнализируется синтаксическая ошибка.

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. Таким образом, данная функция может выполнять роль глобальной переменной в программе, которая может хранить некоторое значение и обменивает его на свой аргумент. Для чтения значения из статического ящика необходимо сначала вызвать ящик с любым аргументом, запомнить то, что он возвратил, снова вызвать с тем, что он возвратил за первый вызов и отбросить результат второго вызова — первый вызов извлекает хранимое значение, второй его восстанавливает. Для записи следует вызвать статический ящик с новым значением и отбросить результат вызова.

*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, и др., 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.

### Функции

Синтаксис:

FunctionDefinition = [ "$ENTRY" ] NAME Block.

Определение функции помещает имя функции в пространство имён как определённое имя локальной (без директивы $ENTRY) или entry-функции (c директивой $ENTRY). Если в пространстве имён уже присутствует данное имя как определённая функция или как функция с другим типом доступа, то сигнализируется синтаксическая ошибка. Синтаксис блока и семантика выполнения функций будут рассмотрены в последующих разделах.

## Разделы про функции

# Библиотека функций

## Используемые обозначения

## Базовая библиотека (Library)

## Расширенная библиотека (LibraryEx)

# Компилятор Простой Рефал: особенности реализации

# Интерфейс с языком C++

## Вычислительная модель

## Написание внешних функций

### Быстрый и грязный способ

### Написание функции вручную

# Установка компилятора

# Известные ошибки

# Список литературы

**Алешин А. Ю. [и др.]** Система программирования РЕФАЛ-2 для IBM PC, PDP-11 и VAX-11. Руководство пользователя [Книга]. - Москва : [б.н.], 1991.

**Немытых А. П.** Лекции по языку программирования РЕФАЛ. [Раздел книги] // Cборник трудов по функциональному языку программирования Рефал. - Переславль-Залесский : Изд-во "Сборник", 2014. - Т. 1. - ISBN 978-5-9905410-1-6.

**Турчин В. Ф.** Фрейм: РЕФАЛ-5. Руководство и справочник [В Интернете] // Содружество «РЕФАЛ/Суперкомпиляция». - 1989 г.. - 16 апреля 2015 г.. - http://refal.ru/rf5\_frm.htm.

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