Язык Рефал — взгляд со стороны[1]

Бойко Банчев

Аннотация

Знакомство с языком Рефал полезно программисту хотя бы потому, что этот функциональный язык не похож ни на один из других — среди них он занимает особое место и по возрасту, и по происхождению, и по назначению, и по стилю. Достойно сожаления то что, несмотря на свои качества, язык не очень популярен.

Статья знакомит читателя с Рефалом. Язык так прост, что его описание почти целиком вмещается в статью — в общем, за исключением стандартных функций, которых тоже немного. Простота сама по себе — положительное качество, но читатель убедится, что оно не единственно.

Помимо описания самого Рефала, представлен взгляд автора на место, достоинства и слабые стороны языка.

Knowing the Refal language is useful to a programmer, if for nothing else than for the language's uniqueness—with respect to its age, its origin, its intended purpose, and style. It is regrettable that, in spite of its qualities, the language is not very popular.

This article gives an introduction to Refal. The language is so simple that it is almost entirely described in this text—except for, mostly, the standard functions, of which there are not many anyway. Simplicity is a virtue by itself, but the reader will receive evidence that it is not the only one.

Apart from describing Refal proper, the article presents the author's view on the language's place, merits, and weak points.

1 Вместо введения

Рефал — язык программирования. Сам я, знакомясь с новым языком, прежде всего хочу увидеть пример программы на нем, так что давайте начнем с примера.

```
$ENTRY Go {= <Read>}
Read {, <Card>:
        {   0 =
        ; e.1 = <Prout <Pal e.1>>
                <Read>}}
Pal {         = 'да'
;         s.1 = 'да'
; s.1 e.2 s.1 = <Pal e.2>
;         e.1 = 'нет'}
```

Это — программа на варианте языка, именуемом Рефал-5. На нем записаны и все остальные примеры, и вообще, главным образом он рассматривается далее.

Программа читает одну за другой строки текста из стандартного устройства ввода, для каждой строки определяя, является ли она палиндромом. `Go`, как можно догадаться — то же самое, что `main` в C — обязательное имя главной функции. Она объявлена при помощи ключевого слова `$ENTRY`, чтобы была видна «извне». `Card` — функция чтения строки со стандартного ввода. Функция же `Read` обращается к ней, пока та выдает строку текста `e.1` (`0` есть признак конца «файла» ввода), которую она передает `Pal`. После того, как последняя выдаст `'да'` или `'нет'`, `Read` обращается к самой себе для получения новой строки: хвостовая рекурсия. `Go` запрещено быть рекурсивной, поэтому цикл чтения-обработки она вынуждена возложить на кого-нибудь.

`Pal` сначала проверяет, не является ли цепочка, ее аргумент, пустой или состоящей из единственной литеры `s.1` — тогда ясно, что она — палиндром, и выдается соответствующий результат. Если это не так, цепочка сопоставляется с образцом `s.1 e.2 s.1`, успешно сопоставляющимся с любой цепочкой, у которой первая и последняя литера одинаковы. Успех сопоставления влечет вычисление `Pal` над серединой `e.2` цепочки, так что в конце концов `Pal` выдаст то, что она выдает при работе над укороченной цепочкой — опять хвостовая рекурсия. Любой другой состав входной цепочки означает, что она — не палиндром; в этом случае цепочка успешно сопоставится с образцом `e.1` и будет выдан ответ `'нет'`[2].

[1] Статья опубликована в журнале *«Практика функционального программирования»*, Выпуск 7 (апрель 2011). Настоящий вариант отличается исправлением нескольких неточностей (они отмечены сносками), добавленным в конце разделом, а также оформлением программного текста.

[2] «Палиндромная функция» `Pal` имеет в Рефале статус факториальной функции в большинстве других функциональных языков: обязательный первый пример рекурсии. Противиться этому освященному мифическими человеко-веками правилу я не стану, но предпочитаю дать не отдельную функцию, а полную программу.

2 Функциональный язык — а какой именно?

Два обстоятельства делают Рефал интересной темой обсуждения в этом журнале. Во-первых, это — язык функционального стиля. Во-вторых, насколько мне известно, это единственный язык, который может быть сочтен «русским языком программирования» — если у языка программирования может быть национальность. Дело прежде всего в том, что Рефал создан в России (СССР) и полностью самобытен в своих идеях, но также и в том, что принцип вычислений в нем подобен алгорифмам Маркова.

Чтобы увидеть место Рефала в функциональном программировании, представляется нужным задаться вопросом «Что такое функциональный стиль программирования?». На этот вопрос ответить нелегко, и это — к счастью. К счастью, потому что трудность ответа свидетельствует об относительной развитости данного направления, и в частности — об имеющемся в нем разнообразии и взглядов на суть программирования, и проистекающих из них практических решений в виде конкретных языков, приёмов программирования и т. д.

Действительно, какие разновидности функционального стиля можно выделить? Согласно λ-исчислению, программирование есть *абстрагирование и применение*, т. е. образование некоторых функций и их применение. При этом смысл функций, их аргументов, равно как и смысл применения, несущественны. Существенно то, что программы — выражения, построенные исключительно из этих двух видов операций. Разумеется, конкретные языки программирования дополняют эту общую модель разного рода особенностями, будь то единообразие представления всей программы (Lisp, REBOL), типовые системы с автоматизированным выводом (ML, Haskell) или полная абстрактность понятия типа (Russell). Все же суть одна: порождение функций и применение.

Другая модель вычислений сосредотачивается на применении — поэтому назовем ее *аппликативной*. В ней применение функций имеет место преимущественно или даже только над другими функциями, и при том оно — единственный способ получить любую новую функцию; образование абстрагированием отсутствует. Это — модель комбинáторного исчисления, FP/FL, в значительной степени APL/J/K, или же безаргументного стиля, скажем в Haskell. При этом «настоящие данные и результат» программы только подразумеваются. Их можно косвенно определить как «то, над чем программа работает, когда ее запустят» и «то, что она выдает», но непосредственно на них не ссылаются.

Некоторым информатикам аппликативная модель кажется ненужно сложной, поэтому вместо общей операции применения они предпочитают ограничиться одной только композицией функций. Как в аппликативном стиле, все порождаемые программой объекты — функции, но порождение происходит применением только композиции. Саму операцию композиции тогда записывать не приходится: она выражается просто сочленением текстов композируемых функций, а потому данный стиль называют *конкатенативным*. Наиболее заметными представителями можно назвать Joy, Factor и (подмножество) PostScript.

На рисунке показано определение функции нахождения среднего арифметического значения списка чисел в разных функциональных стилях и языках.

абстрагировано	`(lambda (ns) (/ (reduce #'+ ns) (length ns)))`	Lisp
	`fun {$ A} {FoldL A Number.'+' 0}/{Length A} end`	Oz
	`\ns -> sum ns / genericLength ns`	Haskell
аппликативно	`÷∘[/+,length]`	FP
	`/[sum,tally]`	Nial
	`liftM2 (/) sum genericLength`	Haskell
конкатенативно	`[sum] [size] cleave /`	Joy
	`count [vec_sum] dip /`	Cat
	`dup 0 exch {add} forall exch length div`	PostScript

Рис. 1: Функции нахождения среднего арифметического значения

Итак, есть по крайней мере три очень разных варианта функционального стиля. При этом они получаются один из другого как бы сужением допускаемых возможностей. (Сужение, кстати, не проявляется ограничением, но не будем обсуждать, почему это так.) Где среди них Рефал? Самое интересное, что нигде: он представитель — и этим он особенно примечателен — другого направления мышления, имеющего мало общего с λ-исчислением или с рассмотрением функций как полноправных значений и результатов в программе.

Рефал функционален, потому что действие программы в нем состоит в вычислении функции, вызванных из нее функций и т. д. и, как правило, функции не имеют побочных эффектов. Вместе с тем, в отличие от большинства функциональных языков, основное внимание уделяется не функциям, а данным, точнее — связи между структурой обрабатываемых данных и структурой

вычислений. Еще точнее, функции в Рефале определяются при помощи механизма сопоставления входных данных с образцом, так что строение данного входного значения непосредственно определяет, какие вычисления над ним производить: это видно уже во вводном примере.

Следовательно, Рефал можно определить как *сопоставительный функциональный* язык. При этом, однако, следует остерегаться прямой аналогии с тем, что знакомо по языкам типа Haskell и ML. Сопоставление в этих языках связано с определением типов и наличием в этих типах конструкторов, т. е. с необходимостью построения некоторого уровня (или нескольких уровней) абстракции над вводимыми данными. В Рефале же функции работают прямо над входным текстом, преобразуя его, наводя структуру и далее таким же способом обрабатывая получившееся. Не строятся иерархии типов и конструкторов. В той мере, в какой структура нужна, она возводится непосредственно из данных и над ними. При этом значения элементарных данных понимаются либо конкретно как текст или числа, либо, более абстрактно, как символы — толкование условно и является вопросом соглашения или осмысления программы, но не касается ее организации.

В связи со сказанным нужно иметь ввиду и то, что Рефал намного старше упомянутых языков, и в частности сопоставления, основанного на конструктивных типах.

Хотя подход Рефала к программированию имеет свои недостатки, и о них будет сказано далее, по сравнению с ориентированными на функции подходы он более прагматичен — и в этом смысле более привлекателен — своей непосредственной направленностью на данные.

3 Начала программирования на Рефале

3.1 Структура и выполнение программ

Программа на Рефале состоит из одной или более функций. Каждая функция задается именем и последовательностью *предложений* в фигурных скобках. Любое предложение имеет две части, между которыми стоит знак равенства `=`:

образец `=` *выражение*

Предложения отделяются одно от другого точкой с запятой.

У любой функции на Рефале ровно один аргумент. Его значение задается любым допустимым в Рефале выражением.

Вычисление функции состоит в сравнении ее аргумента с образцами, рассматриваемыми один за другим по порядку предложений. Неуспех сопоставления приводит к попытке сопоставления со следующим образцом. Как только найдется соответствие, вычисляется выражение за знаком `=`, и значение выдается как результат функции. Неуспех последнего, а значит всех сопоставлений, есть ошибка применения функции.

В самом простом виде выражение является константой, но чаще оно состоит в вызове одной или нескольких функций. Аргумент данной функции сам есть выражение, так что он может быть вызовом другой или той же самой функции и так далее. Таким образом, вызов функции имеет последствием одно или больше сопоставлений, за которыми следуют, в общем случае, новые вызовы функций.

Среди функций программы одна отмечена особым именем как главная. Выполнение программы начинается с нее, а дальше продолжается, как только что было описано.

Подчеркнем, что функции никогда не вызываются сами по себе, а только в результате активирования вычисления некоторого выражения вследствие успешного сопоставления. Другой особенностью является отсутствие глобальных переменных, так же как и вложенности функций. Все это приводит к тому, что контекст вычисления любого выражения строго локальный: в выражении могут встречаться только переменные из соответствующего образца, которые и получают значения именно благодаря успешному сопоставлению.

3.2 Значения

Весьма существенным в программировании на Рефале является понимание значений данных и их структуры.

Объект данных может быть атомарным или составным. Атом — это *литера*, *слово* или *число*. Пример слов: `"как тихо"` и `"b12"`; поскольку второе из них «простое», его можно записать и без кавычек: `b12`. (Простыми считаются слова, являющиеся допустимыми в качестве имен программных объектов: они начинаются буквой и состоят из букв, цифр и литер `-` и `_`.) Пример литерных значений: `'x'`, `'5'` и `'+'`.

Ввиду их особой роли в синтаксисе языка, для некоторых литерных значений используется запись со знаком `\`: `\'`, `\"`, `\n`, `\(`, `\<` и т. д. Через `\x` можно записывать литеры в шестнадцатеричном ASCII кодировании, например `\x20` — знак пробела.

Составные данные задаются *последовательностъями*. Последовательность может быть пустой или состоять из атомов и других последовательностей. Так как любой атом есть последовательность с одним членом, можно считать что всякое значение — последовательность.

Последовательность можно записать перечислением ее членов, как в

```
ok 13 '*' "все" "просто"
```

Пробелы между отдельными элементами последовательности надо понимать как применения операции сцепления.

Сцеплением атомы один с другим не сливаются, а только выстраиваются в последовательность. В этом смысле уместно рассматривать его как конструктор значений-последовательностей, именно — *конструктор сцепления*.

Для вложения одной последовательности в другую применяется *конструктор вложения*, записываемый круглыми скобками около включаемого значения. Это и единственная роль круглых скобок в Рефале, поэтому их называют *структурными скобками*. Так, например,

```
the five (boxing (wizards jump)) quickly
```

– последовательность из четырех элементов, один из которых — последовательность `boxing (wizards jump)`, у которой второй элемент — последовательность `wizards jump`.

Таким образом, все нетривиальные последовательности — те, которые непусты и не являются атомами (а это и есть все составные значения) — образованы конструкторами сцепления и вложения. Всякая непустая последовательность есть не что иное как дерево, у которого в концевых узлах стоят атомы, а в остальных — обособленные (под-)последовательности.

Единичные кавычки используются не только для обозначения отдельных литер, но и вообще для последовательностей литер, так что

```
'народ'
```

– то же самое что

```
'н' 'а' 'р' 'о' 'д'
```

или

```
'на' 'род'
```

Заметим, что в последнем случае та же последовательность получена сцеплением двух более коротких: так как аргументы сцепления есть последовательности, а не атомы, они под действием сцепления сливаются.

Можно считать, что единичные кавычки предназначены для последовательностей (а не одиночных экземпляров) литер и что нет способа записать отдельную литеру, кроме как частью последовательности[3]. Последовательности литер играют ту же роль, что тип цепочек в других языках: их можно сцеплять, в них можно искать, из них можно выделять составляющие их литеры или части большей длины.

Слова, с другой стороны, цепочками не являются. Как значения они атомарны, без элементов, но стандартными функциями из слова можно получить цепочку, а из цепочки образовать слово.

Пустую последовательность удобно записать через `''` («пустая последовательность литер»), но во многих случаях ее даже не записывают, а подразумевают. Во всяком случае, `''` — единственный способ задать пустую последовательность в явном виде. `()` пустой последовательностью не является — это последовательность с одним элементом, а сам он — пустая последовательность. Также и пустое слово `""` — это не пустая последовательность, а именно слово, или же последовательность с одним элементом, являющимся пустым словом, т. е. словом длины 0.

С практической точки зрения о последовательностях важно отметить, что они представлены в памяти так, что доступ к их элементам симметричен относительно двух возможных направлений. Такое представление может опираться на двунаправленные списки или на другую структуру, обеспечивающую указанную симметричность, вплоть до последовательного («векторного») расположения.

В Рефале можно работать с неограниченно большими целыми числами, однако они атомами не являются. «Большое» целое число представляется последовательностью целых атомов, которые в этом случае называют «макроцифрами». Отрицательное число, независимо от его величины, тоже представляется не непосредственно, а последовательностью, начинающейся с атома `'-'`. Стандартные арифметические функции языка работают и с одноатомными, и с многоатомными числами,

[3]Исключением являются литеры, записанные через \: см. раздел *Пример: синтактический анализ выражений языка Рефал*.

однако нужно иметь ввиду, что последние сами по себе не являются структурно обособленными значениями. Так, для отделения одного числа от следующего за ним, в общем, нужно использовать структурные скобки, а `'-' 357` может быть как записью отрицательного, так и просто последовательностью двух атомов.

3.3 Выражения

Выражения — это та конструкция языка, из которой путем вычислений получаются значения. В Рефале все значения — последовательности из атомов и подпоследовательностей, что отражено в записи константных выражений. Выражения общего вида отличаются от константных только включением переменных и вызовов функций.

Обращение к функции записывается в угловых скобках, где на первом месте стоит ее имя, а затем записан аргумент.

Значение переменной или вызова функции участвует в формировании значения соответствующего выражения точно так же, как если бы оно стояло непосредственно в этом выражении на месте переменной или вызова. Вложение при таком включении не подразумевается, его нужно всегда выражать явным образом структурными скобками.

Рассмотрим выражение

```
foo 'bar' e.baz <f qux etc> 'all' that stuff
```

В нем `e.baz` — переменая, а `f qux etc` — вызов функции `f` с аргументом `qux etc`. Допустим, значение `e.baz` — `'corge'`, а вызов `f` возвращает `'gra' ult 'garply'`. Тогда значением выражения является

```
foo 'barcorgegra' ult 'garplyall' that stuff
```

В любом случае, включаемые замещением переменной или вызова функции последовательности теряют свою обособленность, входя в результат не сами собой, а своими элементами. Если указанное выражение изменить на

```
foo 'bar' (e.baz) <f qux etc> 'all' that stuff
```

то его значением будет

```
foo 'bar' ('corge') 'gra' ult 'garplyall' that stuff
```

3.4 Функции, образцы, переменные

Угловые скобки в вызове функции имеют двоякую роль. Можно понимать `<` как «операцию вызова», без которой имя функции было бы просто словом, представляющим себя самого. Знак же `>` обозначает конец выражения, являющегося аргументом вызова — ведь аргумент, хотя и один, может быть любым выражением.

Когда части аргумента вызова функции следует рассматривать как отдельные значения, можно применить вложение, т. е. обособление структурными скобками. Тогда условно можно считать, что у функции несколько аргументов. Наиболее часто для этих целей заключают в скобках каждый «аргумент», возможно, за исключением последнего. Соответственным образом в определении функции надо составлять и образцы распознавания аргумента. Так, функция

```
inv3 {(e.1) (e.2) e.3 = (e.3) (e.2) e.1}
```

меняет местами первый и последний из своих трех аргументов. При обращении к ней

```
<inv3 ("он" "сам") ("хотел" "уехать") "туда">
```

несмотря на то, что формально аргументом является лишь одна последовательность, но она расчленяется образцом нужным образом, и получаем

```
("туда") ("хотел" "уехать") "он" "сам"
```

В предложениях Рефала скобки слева знака `=` являются частями образца, а справа — формируемого значения. В общем, образцы описывают структуру и состав успешно сопоставляющихся с ними выражений-аргументов при помощи констант, переменных и структурных скобок. Константы и скобки соответствуют тем же самым элементам сопоставляемого образцу выражения, а переменные получают значения соответствующих частей того же выражения. Так, если в образце встречается

`(e.x '.')`, переменная `e.x` сопоставляется с некоторой частью аргумента только в том случае, когда эта часть обособлена в подпоследовательность, оканчивающуюся точкой; сама точка в значение `e.x` не входит.

Ход и успешность сопоставления определяются не только структурой образца, но и типами участвующих в нем переменных. Тип переменной задается буквой с точкой в начале ее имени. Задание типа обязательно и указывает на вид структуры значений, которые могут сопоставлятся с переменной.

Тип `e.`, как в примере выше, отвечает любому значению — пустой или непустой последовательности. Тип `s.` (см. пример в начале статьи) соответствует атому («символу»). Рассмотрим в качестве примера еще функцию

```
Flatten {   =
;   s.h e.t = s.h <Flatten e.t>
; (e.h) e.t = <Flatten e.h> <Flatten e.t>}
```

которая образует последовательность из всех своих атомов, сохраняя относительный порядок между ними (плоская проекция содержания последовательности). Согласно второму и третьему предложениям, если в начале аргумента стоит атом, он непосредственно переносится в результат, а если подпоследовательность, то ее часть результата вычисляется рекурсивным вызовом. В обоих случаях остаток аргумента обрабатывается рекурсивно. Тип `s.` переменной дает возможность вычленить именно атом, а не бóльшую часть аргумента. Как и выше, скобки в образце используются для извлечения содержимого подпоследовательности, пропуская сами скобки.

Первое же предложение функции `Flatten` говорит, что для пустого аргумента пустым является и результат. «Пустое» значение в образце или в вычисляемой части предложения можно было бы задать и явным образом через `''`. Для вызова функции с пустым аргументом также можно записать `''` или не писать ничего (но скобки вызова все-таки нужны). К примеру, `<Flatten>` является правильным вызовом.

Взглянув еще раз на функцию `Pal` в самом первом примере, обратим внимание, что повторное употребление имени переменной в образце соответствует не только структурно подобным, а именно одинаковым значениям данных частей аргумента. Механизм сопоставления таков, что аргумент соответствует образцу только в этом случае. Иллюстрацией тому является и следующая функция:

```
In-seq {s.x e.1 s.x e.2 = T (e.1) e.2
;     (e.x) e.1 e.x e.2 = T (e.1) e.2
;                   e.x = F}
```

Будем считать, что у нее два аргумента, и она проверяет, входит ли первый во второй. Точнее, когда первый аргумент является атомом, то функция проверяет, встречается ли он на верхнем уровне второго аргумента (в этом предложении мы отклоняемся от описанного выше соглашения о разделении аргументов скобками). Если же первый аргумент — подпоследовательность, то все ее елементы должны встречаться один за другим в том же порядке во втором аргументе, опять же на верхнем уровне. Условившись передавать успех поиска словом `T`, а неуспех — `F`, функция возвращает один из этих атомов. В случае успеха она также передает в точку вызова контекст, т. е. префикс и суффикс найденного, заключая префикс в структурные скобки. Функцию можно использовать для поиска вхождения одной цепочки в другую, но она применима и к более разнообразным аргументам.

Приведем примеры вызовов `In-seq` и соответствующие результаты:

```
<In-seq go a le go rist>             ->  T (a le) rist
<In-seq a a le go rist>              ->  T () le go rist
<In-seq (le go) a le go rist>        ->  T (a) rist
<In-seq (le go) a (le go) rist>      ->  F
<In-seq ((le go)) a (le go) rist>    ->  T (a) rist
<In-seq ((le go)) a ((le go) rist)>  ->  F
```

Уславливание относительно `T` и `F` как здесь является типичным. В Рефале булевых значений нет, поэтому приходится их имитировать. Впрочем, в конкретном случае можно было бы обойтись без флага истинности и возвращать только префикс и суффикс или пустое значение, поскольку в случае успеха результат будет непустым, и значит, отличить успех от неуспеха всегда возможно. Действительно, даже если и префикс, и суффикс пусты — первый аргумент в точности равен второму — получим `()`, а не `''`.

Если к сказанному еще условиться заключать в скобки не первый, а второй аргумент функции `In-seq`, то ее определение можно упростить до следующего:

```
In-seq {e.x (e.1 e.x e.2) = (e.1) e.2
;                   e.z = }
```

Сделаем следующее наблюдение. Соглашения о различении аргументов нужны, вводятся легко и не слишком перегружают определения функций, однако по одному виду этих определений, да и по вызовам тех же функций в принципе невозможно разобраться, сколько у функции аргументов и каковы они в структурном или содержательном отношении. Тем более, что эти соглашения бывают разными. В этом смысле программы обладают низкой степенью самоочевидности. Как минимум, мы с необходимостью приходим к выводу, что в Рефале требование к документированию программ нужно ставить особенно жестко.

Наряду со структурными типами `e.` и `s.` имеется и третий: `t.`. Он соответствует «терму» — понятию, которым обозначают либо атом, либо выражение в скобках.

В совокупности три типа отражают три степени общности значений: атомы являются также и термами, а термы — выражениями общего вида. Это учитывается, в частности, при упорядочивании предложений функции, а также при выборе переменных в рамках образца, так как переменная более общего вида имеет тенденцию сопоставляться более успешно и с более крупными отрывками значений-аргументов. Крайним случаем является уже встречавшееся `e. = ...`, где переменная сопоставляется с любым выражением. Это — идиоматическое предложение для обеспечения перехвата некоторого значения целиком или успешности выполнения функции.

Наличие термов как самостоятельного структурного типа, формально говоря, не является существенным для распознавания, так как образцы для любых структур можно составлять и без него. Однако оно дает удобную абстракцию для скрытия возможной неатомарности на сколь угодно высоком уровне. Так, любую последовательность можно понимать как однородную, состоящую из термов. Вообще, любой узел в иерархии любого составного значения можно рассматривать как терм, тем самым считая его атомарной сущностью, независимо от его действительного состава.

В следующем примере снова определяется функция поиска, но на этот раз первый аргумент есть терм, в каждом предложении обозначенный через `t.x`, который ищется во втором аргументе на любом уровне вложения. При этом считается, что нужно найти значение, равное `t.x` в целом, а не как в `In-seq` в виде подпоследовательности. Возвращается только атом `T` либо `F`.

```
In-term {     t.x = F
; t.x e.1 t.x e.2 = T
;     t.x s.1 e.2 = <In-term t.x e.2>
; t.x (e.1) e.2, <In-term t.x e.1>: {
                   T = T
                 ; F = <In-term t.x e.2>}}
```

Логика действия функции, отраженная в предложениях в соответствующем порядке, такова. В пустой последовательности `t.x` не встречается. Если `t.x` встречается на верхнем уровне значения второго аргумента, поиск успешен. Если же нет, но первый член последовательности есть атом, то отбрасываем его и ищем далее рекурсивным вызовом. Если, наконец, последовательность начинается подпоследовательностью `e.1`, производим поиск в `e.1`. Если он успешен, то на этом и заканчиваем, а если нет — ищем еще в остатке аргумента `e.2`.

Здесь, впрочем как и в функции `Read` программы для распознавания палиндромов, применяется так называемая *«с»-конструкция* (от предлога «с»). В ней сначала вычисляется выражение над переменными, получившими значения благодаря успешному сопоставлению образца. Затем значение этого выражения сопоставляется по заданному блоку предложений, а результат вызванного сопоставлением вычисления нового выражения выдается как результат исходной функции. Блок предложений играет роль вложенной безымянной функции, которая как бы применяется для уточнения и ветвления основного сопоставления.

Есть еще *конструкция «где»*. По виду она подобна «с»-конструкции и тоже имеет уточняющий сопоставления смысл, но вместо блока и связанного с ним ветвления вторичное сопоставление идет только в одном направлении. Довольно типичный пример использования «где» — проверка принадлежности успешно сопоставленного значения данному множеству. Допустив, что аргумент некоторой функции — цепочка литер, предложение

```
e.1 s.c s.c e.2, 'aeiou': e.3 s.c e.4 = s.c
```

сначала присвоит `s.c` значение дважды подряд встречающейся литеры. Затем новым сопоставлением оно проверит, является ли уже найденное `s.c` одной из гласных букв латиницы и только если это имеет место, выдается результат. Если же вторичное сопоставление неуспешно, предложение попытается удовлетворить основное сопоставление повторно (удлиняя сопоставляемую с `e.1` часть), после чего, в случае успеха, опять произойдет вторичное сопоставление, и т. д.

Если по данному предложению сопоставить цепочку `'attendee'`, произойдет следующее. Сначала, согласно образцу `e.1 s.c s.c e.2`, переменные `e.1`, `s.c` и `e.2` получат значения соответственно `'a'`, `'t'` и `'endee'`. Затем идет попытка сопоставить цепочку `'aeiou'` с образцом `e.3 s.c e.4`, но

так как в 'aeiou' нет 't' (значение s.c), сопоставление неуспешно. Это приводит к повторному поиску сопоставления с образцом e.1 s.c s.c e.2, в результате которого переменным e.1, s.c и e.2 будут присвоены 'attend', 'e' и ''. Теперь уже сопоставление 'aeiou' с e.3 s.c e.4 успешно, поскольку 'e' — значение s.c — входит в эту цепочку. Буква 'e' и выдается в конечном счете предложением. При сопоставлении же цепочки 'Mississippi' переменной s.c значение присваивается три раза, и каждый раз присвоенное отбрасывается конструкцией «где».

Ввиду схожести «с»- и «где»-конструкций и отсутствия ветвления во второй, ее можно рассматривать как частный случай первой. Однако в чем-то она и мощнее: повторение попытки сопоставления, как описано выше, возможно только с применением конструкции «где», но не «с».

Несколько конструкций «где» можно применять последовательно, а конструкции «с» — естественным образом вставлять одну в другую. Два вида конструкций можно и сочетать. Во всех случаях имеет место последовательное уточнение сопоставления.

Заметим что, применяя несколько «где» одну за другой, получаем конвейер поисков сопоставлений с возвратным поведением (backtracking). Точки поиска-возврата вложены одна в другую: откат из любой из них (неуспех сопоставления) служит сигналом возобновления поиска в предыдущей.

С формальной точки зрения конструкции «с» и «где» являются лишь синтактическими удобствами, без которых можно обойтись, но их практическая полезность высока: они помогают выражаться более четко и непосредственно, уменьшая, в частности, вынужденное введение функций вспомогательного характера.

4 Пример: синтактический анализ выражений языка Рефал

Рассмотрим несколько больший пример программирования на Рефале. Приведенный ниже текст является модулем, реализующим синтактический анализ константных выражений языка Рефал. По заданной цепочке текста функция expr, следуя правилам грамматики языка, находит записанное в цепочке значение-последовательность.

Из всех функций модуля только expr является доступной вне него. Чтобы ссылаться на нее из другого места программы, нужно добавить определение

```
$EXTRN expr;
```

Для выполнения основной работы expr вызывает parse, предваряя свой аргумент пустой последовательностью (), содержание которой растет по ходу распознавания в цепочке значений членов результирующей последовательности. Если цепочка успешно переводится в последовательность на Рефале, parse выдаст эту же последовательность в скобках, которые expr снимет. Если же цепочку удается перевести не целиком, expr получит либо слово F, либо результат в скобках с непустым остатком цепочки за скобками — в любом случае цепочка оказалась неправильным выражением и expr выдаст F.

Вспомогательные функции name и number распознают соответственно слова простого вида (без ") и числа (макроцифры). Функция quoted переводит подцепочку вида '...' в цепочку литер на Рефале, а "..." — в слово общего вида. Ответственность функции special — распознание литер, заданных через \; сама она вызывает xpair для шестнадцатерично заданных и srep — для остальных особых литер.

Для представления различных классов литер в обрабатываемой цепочке используются функции с letter по xdigit. Они вызываются только с пустым аргументом и не вычисляют ничего, но тем не менее возвращают значения. Такие функции принято применять в роли «глобальных констант» (которыми в собственном смысле язык не располагает).

Функция size — оболочка Lenw, отбрасывающая лишнюю часть результата последней. (Lenw — стандартная функция для нахождения длины последовательности, но, кроме необычного имени, она отличается еще более необычным результатом: часть его — та же последовательность.)

Использованы также стандартные функции Lower — для преобразования содержимого цепочки в строчные буквы — и Implode_Ext — для получения символьного атома из цепочки.

Любопытной особенностью Рефала-5 является возможность использовать \-запись литер не только между кавычками '...' и "...", но также и самостоятельно — тогда подразумевается наличие единичных кавычек. Например, \" \n и \"\n обе равнозначны '\"' '\n' или же '\"\n', а \'A\', так же как и \' A \', обозначает последовательность ' A ' из двух литер ' со словом A между ними. Это учитывается в анализе выражений, при вызове special и из quoted, и непосредственно из parse, а также, для упрощения и сокращения текста, используется в определениях функций везде, где уместно.

```
letter {= 'abcdefghijklmnopqrstuvwxyz'}
digit {= '0123456789'}
```

```
namedlm {= '-_'}
spechar {= '\'\"\\()<>tnrx'}
xdigit {= <digit> 'abcdef'}

size {e.x, <Lenw e.x>: s.n e.z = s.n}

$ENTRY
expr {e.x, <parse () e.x>: {(e.res) = e.res;  e.z = F}}

parse {
  (e.x) = (e.x)
; (e.x) \' e.r1, <quoted \' () e.r1>:
    {F = F;  (e.q) e.r2 = <parse (e.x e.q) e.r2>}
; (e.x) \" e.r1, <quoted \" () e.r1>:
    {F = F;  s.q e.r2 = <parse (e.x s.q) e.r2>}
; (e.x) \\ s.c1 e.r1, <special s.c1 e.r1>:
    {F = F;  s.c2 e.r2 = <parse (e.x s.c2) e.r2>}
; (e.x) s.c e.r1, s.c: {
    s.a, <letter>: e.1 s.a e.2, <name (s.a) e.r1>: s.n e.r2
      = <parse (e.x s.n) e.r2>
  ; s.d, <digit>: e.1 s.d e.2, <number (s.d) e.r1>: s.n e.r2
      = <parse (e.x s.n) e.r2>
  ; '(', <parse () e.r1>:
    {(e.y) ')' e.r2 = <parse (e.x (e.y)) e.r2>; e.z = F}
  ; ' ' = <parse (e.x) e.r1>
  ; s.z = (e.x) s.c e.r1}}

name {
  (e.x) s.c e.r, <letter> <digit> <namedlm>: e.1 s.c e.2
    = <name (e.x s.c) e.r>
; (e.x) e.r = <Implode_Ext e.x> e.r}

number {
  (e.x) s.c e.r, <digit>: e.1 s.c e.2
    = <number (e.x s.c) e.r>
; (e.x) e.r = <Numb e.x> e.r}

quoted {
  s.d (e.x) s.d e.r, s.d:
    {\' = (e.x) e.r;  \" = <Implode_Ext e.x> e.r}
; s.d (e.x) \\ s.c1 e.r1, <special s.c1 e.r1>:
    {F = F;  s.c2 e.r2 = <quoted s.d (e.x s.c2) e.r2>}
; s.d (e.x) s.c e.r = <quoted s.d (e.x s.c) e.r>
; e.x = F}

special {
  s.c e.r1, <spechar>: e.1 s.c e.2, s.c: {
    'x', e.r1: s.a s.b e.r2, <xpair <Lower s.a s.b>>:
      {F = F;  s.x = s.x e.r2}
  ; e.z = <srep s.c> e.r1}}

xpair {
  s.1 s.2, <xdigit>: e.1 s.1 e.2, <xdigit>: e.3 s.2 e.4
    = <Chr <+ <* 16 <size e.1>> <size e.3>>>
; e.z = F}

srep {\' = \';  \" = \";  \( = \(;  \) = \);  \< = \<
;     \> = \>;  't' = \t;  'n' = \n;  'r' = \r;  \\ = \\}
```

Из допустимых данной реализацией анализатора выражений исключены имена переменных и вызовы функций. Собственно грамматический анализ этих частей не вызвал бы затруднений, и даже по большей части уже сделан. Имя переменной за знаком `.` — либо простое слово, либо чис-

ло; имя функции — любое слово, а аргумент — любое выражение. Анализ всего этого и так уже реализован. Однако непосредственно представить результат перевода неконстантных выражений невозможно: пришлось бы либо как-то условиться относительно толкования результата, либо перейти с программирования на уровень метапрограммирования, который в таком случае должен быть заранее обеспечен языком. Впрочем, о второй из этих возможностей упоминается в следующем разделе.

5 Дополнительные возможности

Программирующий на Рефале имеет в своем распоряжении некоторые средства императивного программирования, хотя они ограничены и считается хорошим стилем воздерживаться от их использования насколько возможно.

Прежде всего, императивным является, конечно, ввод-ввывод. В отношении ввода-вывода через файлы и стандартные устройства в Рефале нет ничего, на что стоило бы обращать внимание знакомому с такими же средствами в популярных императивных языках.

Более интересен и несколько необычен механизм «закапывания/выкапывания».

Выделенной для этой цели функцией можно создать особый объект данных и назначить ему имя. По назначенному имени в других функциях объекту можно присвоить значение или извлечь его. Можно также присвоить значение, сохранив («закопать») уже имеющееся, и делать это сколько нужно раз. И наоборот, сохраненные значения можно извлекать в обратном запоминанию порядке. Таким образом, объект данного типа есть на самом деле стек. Более того, в нем можно сохранять любые значения, а так как функциями управления объектами можно пользоваться везде, то и все эти объекты доступны из любой точки программы. Другими словами, они являются глобальными именованными стеками универсального назначения.

Считается, что стеки глобальных значений могут ускорить доступ к данным, устраняя необходимость явной передачи последних между функциями, однако по этому пути можно очень легко прийти к программе, «нефункциональной» не только с точки зрения стиля выражения, но в каком угодно смысле.

В целях поддержки создания больших программ их можно разбить на модули, для каждого модуля определяя доступные из других модулей имена его функций. Пример модуля уже был показан в предыдущем разделе.

В каждом модуле может быть определена и сделана доступной функция `Go` — это будет означать, что модуль может быть главным при некотором выполнении программы. Какому из таких модулей действительно быть главным — уточняется лишь в момент запуска программы: исполнение начинается с функции `Go` этого модуля.

Диалектом Рефал-5 активно поддерживается возможность метапрограммирования. Через встроенную функцию отрывок программы можно перевести в похожую, но все же отличающуюся форму: это названо «погружением в метакод». И наоборот, полученный указанным или другим способом метакод при помощи другой функции «подымают из метакода» и тут же исполняют. Так как метакодовым представлением можно орудовать как данными, то можно подвергнуть преобразованию любую часть программы, включая сам преобразователь, а также можно и порождать-исполнять совсем новые куски. Таким способом можно организовать разного рода кусочное, контролированное исполнение, в отладочных и других целях.

Данный процесс — отнюдь не интерпретирование текста. Погружение можно делать лениво, помечая, а не преобразуя соответствующие части, с тем чтобы подъем тоже ничего не преобразовал. Кроме того, закодированная программа частично выполняется, насколько это возможно в зависимости от наличия в ней переменных с неопределенными значениями, так что при подъеме выполняется только некоторое остаточное множество действий.

Поддержка метапрограммирования отсутствует в более поздних диалектах, то ли ради простоты реализации, то ли для упрощения определения самого языка.

В реализации Рефала предусмотрена возможность трассировки. О ней идет речь в следующем разделе.

6 Трассировщик

Интересной и практически ценной особенностью трассировщика является то, что точки прерывания задаются указанием функции и образца аргумента, при соответствии с которым происходит прерывание. При этом образец не обязан совпадать с образцом, используемым в определении функции[4].

[4] Е. Кирпичев указывает на то, что трассировщик системы программирования языка Erlang работает аналогичным образом.

Введение новых, «трассировочных» образцов и переменных превращает трассировщик в инструмент исследования программы, при помощи которого без каких бы то ни было изменений в ней самой можно проводить наблюдения над ее поведением с разных точек зрения. Рассмотрим для примера построение двоичного дерева поиска из целых чисел. Дерево либо пусто, либо имеет вид

(*левое поддерево*) *число* (*правое поддерево*)

Функция `insert` вталкивает число в дерево:

```
insert {s.x = () s.x ()
; s.x (e.1) s.y (e.2), <Compare s.x s.y>: {
    '-' = (<insert s.x e.1>) s.y (e.2)
  ; s.c = (e.1) s.y (<insert s.x e.2>)}}
```

Функция же `build` строит дерево из последовательности чисел в структурных скобках, накапливая результат во «второй аргумент»:

```
build {  () e.r = e.r
; (s.x e.z) e.r = <build (e.z) <insert s.x e.r>>}
```

Допустим, в программе она вызывается так:

```
<build (3 1 6 7 2 5 4)>
```

Запустив программу через трассировщик, установим такую точку прерывания:

```
set <build (5 4) e.r>
```

Это момент, в котором в изначально пустом дереве уже вставлено все за исключением последних двух чисел, `5` и `4`. Частичный результат-дерево будет доступен через переменную `e.r`. Командой `g` исполняем программу до указанной точки останова и написав `p e.r` печатаем значение `e.r`:

```
(() 1 (() 2 ())) 3 (()6 (() 7 ()))
```

Это же, но в контексте вызова функции, в котором оно имеет место, можно получить и командой `p v` («показать текущее выражение»).

Почти то же самое, на чуть нижнем уровне вычисления, можно получить, выбрав точку останова другим образом:

```
set <insert 2 e.r>
```

Теперь дерево (`p e.r`) состоит только из первых четырех чисел (`3`, `1`, `6` и `7`), а числу `2`, являясь аргументом `insert`, лишь предстоит быть добавленным. `p v` показывает, что вычисляемое выражение состоит из вызова `build` и вложенного в нем вызова `insert`.

Установив точку прерывания

```
set <insert 4 t.1 t.rt t.3>
```

можно проследить путь включения числа `4`: скомандовав `g` и затем `p t.rt` получим `3`; повторив то же самое еще два раза, получим еще `6` и `5` — включение происходит в (под)деревья с корневыми узлами `3`, `6` и `5`, в этом порядке.

Еще возможность — проследить все включения узлов `s.n` в поддеревья, состоящие к моменту включения из единственного узла `s.rt` (листевые узлы):

```
set <insert s.n () s.rt ()>
```

– в данной точке программа остановится четыре раза, со значениями `s.n` и `s.rt` соответственно `1` и `3`, `7` и `6`, `2` и `1` и `4` и `5`.

Так же прост перехват включений узлов `s.n` в поддеревья с корневыми узлами `s.rt`, у которых пусто только левое поддерево:

```
set <insert s.n () s.rt (t.1 t.2 t.3)>
```

(Для данного примера единственная пара таких узлов — `5` и `6`.

Функция `build` здесь определена в хвосто-рекурсивной форме, а ее результат накапливается явным образом в ее аргумент. Ее можно было бы определить и несколько проще:

```
build { = ;  s.x e.z = <insert s.x <build e.z>>}
```

и вызывать

```
<build 3 1 6 7 2 5 4>
```

однако в такой форме ее не так удобно трассировать. Хвостовая рекурсия не только более эффективна, но и лучше уживается с наблюдением поведения программы.

7 Диалекты

Рефал был создан во второй половине 1960-х годов В. Турчиным, сначала в качестве метаязыка для описания семантики других языков, а вскоре затем — и как универсальный язык программирования для обработки текстовой и символьной информации.

Практическое применение Рефала со временем привело к совершенствованию элементов первоначального — не очень удобного — синтаксиса, а также к формированию диалектов языка. Для удобства обозначения исходного языка, который и оказался общим подмножеством диалектов, ввели понятие Базисный Рефал.

В настоящее время основными диалектами языка считаются Рефал-2, Рефал-5, Рефал-6 и Рефал+. Рефал-2 [1] — самый ранний, появившийся почти непосредственно после рождения языка. На протяжение десятка или больше лет он оставался единственным диалектом и фактически отождествлялся с самым языком. С современной точки зрения его синтаксис громоздок и устарел [2]. Хотя реализации Рефала-2 имеются и сегодня, он не развивался уже много лет и, если он где-то и используется, то это незаметно.

Рефал-5 [3, 4, 5] создан в середине 1980-х тем же В. Турчиным[5]. Он тоже на сегодня не развивается[6], но остается в употреблении. Его следует считать классическим диалектом Рефала. (Напомним, что в этой статье рассматривается почти исключительно Рефал-5.)

Рефал-6 [6, 7] был задуман как реализация Рефала-5, но в конце концов вырос в самостоятельный диалект. Через него в язык вошли обработка неуспехов сопоставлений и функций и другие удобства, а также, на библиотечном уровне, ассоциативные таблицы и другие дополнительные структуры данных. На сегодня он тоже не развивается. Любопытная особенность: единственно в этом диалекте реализованы вещественные числа и арифметика над ними. Вместе с тем, однако, в нем нет средств для ввода таких чисел или извлечения их значений по текстовой цепочке с соответствующим содержанием[7].

Рефал+ (также Рефал Плюс) [8, 10] является самым продвинутым диалектом, и притом единственным, который развивался (правда, не сам язык, но его реализация) в последние несколько лет. Дополнительные по сравнению с остальными диалектами свойства Рефала+ сохраняют стиль программирования на Рефале и общий вид программ, но повышают их выразительность и лаконичность. Некоторые из этих дополнений — таблицы, перехват неуспехов — подобны имеющимся в Рефале-6[8] [9]. Примеры других свойств: неограниченно большие целые числа как атомарные значения, а не последовательности макроцифр; возможность формировать признак успеха сопоставления на основе успехов и неуспехов частичных сопоставлений; типизирование и обязательное прототипирование функций (в терминах структурных типов — атомов, термов и последовательностей — значений Рефала).

По-видимому, часть нововведений Рефала+ привела к значительному увеличению количества синтактических правил и ухудшению понимаемости определения языка и программ на нем. В грамматике Рефала+ имеется примерно 100 синтактических категорий (в языке C их немного больше 60). К тому же, среди имен этих категорий встречаются «забор», «перестройка», «огражденная тропа», «образцовое распутье», «жесткое выражение» и «жесткий край». Последнее вызывает симпатию к неординарности мышления авторов диалекта, но вряд ли способствует овладению языка.

8 Достоинства и недочеты

Во время своего создания язык Рефал был в нескольких важных отношениях весьма передовым и даже опережающим свою современность. *Лежащие в его основе идеи, тем более их сочетание, предвосхитили тенденции в развитии программирования, проявившиеся лишь десятки лет спустя.* Перечислим основные достоинства Рефала.

- Декларативный, а не командный и не основанный на понятии состояния, стиль программирования. Рефал — один из первых и очень немногих таких языков.
- Сопоставление с образцом как способ определения функций и структурирования вычислений разветвлением. И в этом отношении Рефал сильно опередил другие языки. Можно дополнить,

[5] К тому времени автор языка работал уже в США, где некоторое время преподавал Рефал в Городском университете Нью-Йорка (The City University of New York).

[6] Хотя планы на то есть — см. [19], раздел «Задачи».

[7] Здесь я ошибался — и ввод, и извлечение из цепочки вещественных чисел в Рефале-6 были реализованы, но константы этого типа в программе транслятор все же не допускает.

[8] На ранних стадиях своего развития два диалекта взаимно обогащались.

[9] А. В. Климов, автор современного Рефала-6, сообщил мне, что на самом деле структуры данных и перехват неуспехов в этом диалекте появились под влиянием именно Рефала+. С другой стороны, оригинальным нововведением Рефала-6 была единообразная форма представления этих структур.

что применение сопоставления в программировании исследовалось в то же самое время еще очень ограниченно и отнюдь не в декларативном контексте.

- У функции лишь один аргумент и результат. Разнообразие и общность применимости достигаются не количеством аргументов, а приданием подходящего строения единственному аргументу функции. Так же и насчет результата. Этот принцип хорошо известен по современным функциональным языкам, где ему следуют практически везде, но сорок с лишним лет назад господствовали другие представления.
- Неограниченная последовательность с симметричным доступом к элементам — основная структура данных в языке. Последовательности являются гораздо более ценной структурой по сравнению с (однонаправленными) списками[10].
- Реализация посредством компилирования исходной программы в программу на языке виртуальной машины, которая затем интерпретируется.
- Автоматическое выделение и освобождение памяти.

Последние два свойства тоже не так часто встречались во время становления Рефала.

Удивительно, но данная совокупность свойств как будто относится к языку, изобретенному сегодня, и к тому же продвинутому!

У Рефала, однако, есть и немало недостатков. Часть их относится к языку вообще, другая — к Рефалу как к представителю функционального стиля. Рассмотрим наиболее существенные.

У Рефала отсутствует возможность построения и именования структур данных и тем более — задания определенных программистом типов. «Знание типов» со стороны языка исчерпывается различением атомарного значения от составного (последовательности) и видов атомарных значений (числа, литеры и символы). Это приемлемо для небольших программ, но сильно затрудняет создание крупных.

Проблемы возникают даже на уровне основных видов значений. За исключением Рефала+, большое целое число, и даже небольшое отрицательное — не атомарное значение, а последовательность[11]. Из последнего вытекает необходимость рассматривать два вида чисел — атомарные и составные, а также вкладывать составное число в другую последовательность, скобками отгораживая его от окружающего контекста — без этого произошло бы сцепление. И, конечно же, из-за этих усложнений возрастает ожидаемость ошибок в текстах программ.

Аскетизм в отношении типов проявляется и в отсутствии в языке булевых значений и булевой арифметики. Вместе с тем нет и какой бы то ни было формы условного ветвления, если не считать выбора предложения в рамках данной функции на основе успешности сопоставления с одним из нескольких образцов. К примеру, сравнение по величине двух чисел состоит в применении функции `compare`, выдающей `'+'`, `'-'` или `'0'`, с последующим сопоставлением результата с этими тремя образцами. Однако успех/неуспех сопоставления — всего лишь неполноценная и часто неуклюжая имитация булевой арифметики и выбора действия.

Для выражения повторяющихся действий в функциональном языке естественно рассчитывать на рекурсию. Во многих языках, однако, для часто возникающих схем повторения предусматриваются специализированные под них конструкции высокого уровня, или по крайней мере стандартные функции, которые, вбирая в себя рекурсию, скрывают ее от программиста. Такими являются, скажем, определители списков (list comprehensions), функции вида *map*, *zip*, *fold* и пр. В Рефале подобного вида конструкций нет и поэтому рекурсию приходится выражать всегда в явной форме. Это приводит к большому количеству вспомогательных функций, из-за чего программа имеет тенденцию становиться чересчур раздробленной, громоздкой, а ее смысл — расплывчатым.

Заметим, что проблема раздробленности и расплывания смысла усугубляется тем, что функции не могут быть вложенными[12], а значит, отношение подчиненности или различение смысловых уровней нельзя полноценно передать структурой текста программы.

С другой стороны, почти недоступна и возможность создания функций высшего порядка. Это потому, что функции в Рефале не есть значения: нельзя создавать безымянные функции, тем более замыкания. Самое близкое к функции-значению, что является возможным — взять имя или адрес данной функции: в этом отношении Рефал не превосходит C.

Функции стандартной библиотеки[13] тоже не имеют ничего общего с функциональным стилем программирования. На самом деле, библиотеку трудно отнести даже вообще к Рефалу. Трудно объяснить почти полное отсутствие в ней функций для работы с текстовыми цепочками, равно как и с последовательностями — ведь именно это и есть данные в языке. Отсутствуют даже арифмети-

[10] Кстати, спискам и сегодня в Haskell, ML и т. д. уделяется, похоже, незаслуженно большое внимание по сравнению с последовательностями с прямым доступом. По-моему, это — атавизм.

[11] В действительности, целые числа являются атомарными и в Рефале-6, на что я не обратил внимания до выхода статьи.

[12] за исключением вложения блоков «с»-констукций, если рассматривать их как функции

[13] Точности ради, библиотечными они являются в Рефале-6 и Рефале+, а в Рефале-5 они считаются встроенными.

ческие функции нахождения абсолютного значения, меньшего/большего из двух чисел и обращения знака.

У механизма сопоставления имеется то неудобство, что в рамках образца нельзя выразить ни альтернирование, т. е. ветвление сопоставления, ни повторение, за исключением одинаковых частей, цитируемых одной и той же переменной. Другими словами, обобщение сопоставлений, аналогичное легко выражаемому примерно аппаратом регулярных выражений, в Рефале невозможно. Вследствие того некоторые задачи, которые очень легко решить регулярными выражениями, требуют неестественно больших и запутанных программ на Рефале.

Представляется весьма полезным (но в Рефале не так) иметь возможность обращаться с образцами как с данными. Использование переменных образцов повысило бы гибкость сопоставлений. С другой стороны, образцы можно было бы использовать и как определители типов аргументов и результатов функций.

Недостатком Рефала, препятствующим его применению в современном программировании, является и «замкнутость» языка — прежде всего, отсутствие программного интерфейса к другим языкам и средств обмена данными через Интернет.

Наконец, имеющиеся описания действующих реализаций языка несколько неполны и в какой-то степени устарели.

Все приведенные выше критические замечания относятся в полной мере к Рефалу-5. Рефал-6 и Рефал+ восполняют только небольшую часть указанных пробелов и только частичным образом.

9 Перспективы применения и развития

Несмотря на описанные недостатки, Рефал можно успешно применять в нескольких областях. Может быть самая главная из них — преподавание и изучение начал программирования, а также применений программирования для решения задач в других дисциплинах. Основанием тому служит простота языка и легкость моделирования решений многих не очень сложных задач, где естественно работать с текстовой или символьной информацией. Примеры такого применения даются в [11, 12, 13].

Рефал можно также с успехом использовать для решения многих практических задач анализа и преобразования текста, в том числе для преобразования текстов на формальном (алгоритмическом, описательном и др.) языке и для анализа текстов на естественном языке. Например, преобразование может состоять в снабжении тегами HTML/XML или TEX некоторого текста на основе распознания в нем определенной структуры. Или такой текст, уже снабженный тегами, скажем, HTML, переразметить в LATEX, или переформатировать. В этом отношении интерес может представлять [14], где защищается перспективность применения Рефала для работы с семейством языков XML.

Также благоприятными для Рефала областями являются распознавание и алгоритмы над абстрактными структурами — такими как, например, в области искусственного интеллекта — где нужно интенсивно применять поиск по структурному или содержательному образцу.

Нельзя не отметить, что в СССР Рефал уже имел активное использование в разработке программного обеспечения: систем компьютерной алгебры, компиляторов, программ для специализированных компьютеров, в том числе встраиваемых в изделия космической промышленности [20]. Был также создан экспериментальный Рефал-процессор. Считается, что Рефал сыграл решающую роль для успеха изготовления адаптивной технологии построения серии компиляторов с языка Фортран для специализированных компьютеров. Похоже, что немало из перечисленных работ велись в закрытых или ограниченного доступа научных и научноприкладных центрах, и поэтому публикаций, свидетельствующих о них, почти нет. Некоторые все же можно найти в библиографическом хранилище сайта [18].

Вместе с самым языком В. Турчиным предложен и общий метод глобального оптимизирующего преобразования программ, названный им «суперкомпиляцией» (от англ. *supervising compilation*), в основе которого лежат методы, похожие на абстрактную интерпретацию. Установлено, что благодаря такой глобальной оптимизации из некоторых программ можно получить чувствительно более быстродействующие варианты. Идея суперкомпиляции описана первоначально в [21] и позднее, более подробно, в [22].

Суперкомпилятор под именем Scp4 распространяется с вебсайта Рефала-5 [3] и на сегодня является, по всей видимости, самым достопримечательным применением языка Рефал.

Хотя долгое время суперкомпиляция развивалась в среде только Рефала, в последнее время идут исследования по применению метода к другим функциональным языкам, прежде всего Haskell.

Упомянем две разработки, чья цель — сочетание Рефала с широкоиспользуемыми программами [15]. Refal-SciTE — интегрированная среда программирования на Рефале-5, основанная на очень компактном и гибко настраиваемом текстовом редакторе SciTE. Refal-PHP — система программи-

рования, объединяющая использование Рефала и PHP; там, где как правило используется PHP, доступным становится и Рефал.

Разработчики Рефала+ нашли перспективным реализовать Рефал (в принципе, все главные диалекты) переводом в C++, Java и возможно других языках [9]. Для этой цели они определили промежуточный язык, целевой для трансляторов всех диалектов, который и переводится в упомянутые языки. Окончательно исполняемая программа получается вызовом транслятора конкретного языка.

Понятно, что применение Рефала ограничено из-за приведенных в предыдущей части недостатков языка, поэтому очень жаль, что ни один из основных диалектов на настоящий момент не развивается. Все же попытки осовременивания языка делаются. Так, Рефал-7 [16] вводит вложенные и безымянные функции, а также функции высшего порядка, чем стремится поставить язык, можно сказать, в истинно функциональном контексте: ведь действительно надо считать неестественным в функциональном языке неприятие функций в роли равноправных данных.

К примеру только заметим что, приняв на вооружение безымянные функции, открывается возможность выражать, скажем, композицию таких функций в виде функции, и вообще программировать непосредственно функциями. Хотя дух Рефала несколько иной — ориентированный прежде всего на данные — но программирование на уровне функций является слишком ценной формой абстракции, чтобы пренебречь ею.

Другой отпрыск под именем D-Refal [17] задался целью повысить практическую применимость языка в работе с текстом, совершенствуя механизм образцов. Он вводит конструкции группирования, альтернирования, повторения и отрицания образцов, наподобие образующим операциям регулярных выражений. Программисту предоставляются и определяемые программистом типы: это, по существу, именованные образцы, которые цитируются в других образцах, но их сила больше, чем только в сокращении записи, поскольку они могут быть и рекурсивными, а значит — выражать неограниченную повторность некоторой части сопоставления. Кроме того, D-Refal прилагает усилия в направлении повышения быстродействия сопоставления путем устранения избыточности в нем с помощью ссылок на значения и отсечений. О практической направленности свидетельствуют также введение вещественных чисел и принятие Юникода в роли алфавита обрабатываемого текста.

К сожалению, упомянутые два диалекта не имеют ни популярности основных, ни даже статуса «официально признанных»: на сайте «сообщества Рефала» [18] они не упомянуты и, по-видимому, не обсуждались. Впрочем, и тоже к сожалению, в данном сообществе Рефал уже годами никак и не обсуждается.

Независимо от указанных и, возможно, других улучшений Рефала в будущем, или даже отсутствия улучшений, кажется привлекательным реализовать Рефал как программную библиотеку для языка C. Это позволило бы применять Рефал в качестве расширяющего, сценарного языка почти везде, и заодно — через программирование на C — быть расширяемым самому. Чрезвычайно успешный пример такой симбиотической реализации — язык Lua. Структура Рефала тоже благоприятствует данному подходу.

10 Аналогии

Подходя к концу, вкратце сравним Рефал с другими языками, решающими подобные задачи.

Prolog — хорошо известный пример тоже декларативного языка и тоже основанного на сопоставлении. Однако (оставим в стороне другие различия) в программе на Prolog-е предметом сопоставления является цель, результат, и действие программы состоит в поиске аргументов, при которых цель удовлетворяется, или же в проверке того, удовлетворяется ли она при заданных аргументах. В Рефале же сопоставление касается аргументов, а цель не формулируется явным образом. Похоже что в ряде случаев «прямой» подход Рефала приводит к более наглядным программам, чем «обратный» Prolog-а, но также возможно, что в других случаях как раз наоборот. Что точно известно, так это то, что разница существенным образом сказывается на стиле программирования. Рефал, однако, однозначно выигрывает своими симметричными последовательностьями против однонаправленных списков Prolog-а.

Нельзя не упомянуть и Snobol. Этот язык был еще в 1970-е и остается мощным средством программирования в той же области, что и Рефал — текст и символьные преобразования. В Snobol-е тоже основа вычислений — сопоставления, но тем сходство с Рефалом кончается. В этом языке образцы — полноправные данные иерархической структуры, включающие, помимо прочего, вызовы функций и присваивания переменным, которые могут быть безусловными или зависеть от успеха сопоставления. Успешные сопоставления сопровождаются заменой распознанной части текстовой цепочки новым текстом. В этом смысле, хотя Snobol совсем не функциональный язык, он даже

в большей степени «марковский» чем Рефал. В отношении сопоставительной семантики Snobol, несмотря на возраст, очень продвинут. С другой стороны, быстродействие программ на Рефале, как правило, чувствительно выше.

TXL и OmniMark — два современных языка, в которых программа представляет собой совокупность правил распознавания и преобразования текста. В отличие от Рефала управление последовательностью действий в них осуществляется на событийном принципе: выбирается правило с успешным сопоставлением (распознавание — это активирующее работу данного правила событие), в нем происходит некоторое преобразование текста, затем опять распознающим событием выбирается правило и т. д., пока возможно. Можно сказать, что оба языка работают на несколько более высоком семантическом уровне. TXL можно считать чисто функциональным, хотя в нем, как в Рефале, функции не есть данные. Практически ценной особенностью OmniMark-а является возможность режимного переключения с общего распознавания на текст, размеченный в SGML, а значит и XML или HTML.

11 Благодарности и уточнения

Мне очень приятно выразить благодарность редакторам журнала ПФП за приглашение написать эту статью, за ценные советы по улучшению ее содержания, а также за языковую помощь по превращению моего текста в не слишком ужасное для читателя испытание.

Все оставшиеся несовершенства выражения — по вине ограниченности моего владения русским языком.

Также хочу поблагодарить рецензентов статьи за их полезные замечания, которые помогли сгладить неточности и неясности изложения и дополнить его.

Наконец, хочу подчеркнуть, что все мнения здесь — лично мои и ни в какой мере не претендуют на разделяемость в том или ином сообществе программистов.

12 Дополнение: имитация частичного применения и функций высшего порядка

Описанный здесь прием я узнал от А. В. Климова после того, как статья появилась в журнале ПФП. Он интересен с точки зрения функционального программирования, а вместе с тем иллюстрирует своеобразие обращения с функциями как раз в Рефале, так что его включение в обзорный материал по языку более чем естественно.

Рассмотрим такую функцию:

```
apply {
  s.f e.a   = <Mu s.f e.a>
; (e.f) e.a = <apply e.f e.a>}
```

Библиотечная функция `Mu` — косвенный вызыватель функции, чье имя является значением «первого аргумента» `Mu`. Смысл ее в том, что иногда удобно указывать вызываемую функцию именно как значение переменной, тем самым параметризуя вызов и по функции.

`apply` является обобщением `Mu`. Она сначала устраняет скобки около имени функции, пока они есть, а затем применяет `Mu`. Так как вместе с именем за ним в скобках могут стоять и другие значения, они после раскрытия скобок окажутся частью аргумента вызова.

В итоге имеет место имитация применения частично примененной функции — той, что в скобках. Упаковывая имя функции в терм вместе с частью ее аргументов получаем «частичное применение». Результат можно таким же образом «применять» далее, получая более уточненное, но все еще частичное применение, или же, передав его `apply`, действительно вызвать функцию и тем самым окончательно добыть значение.

Погружение вызова в структурные скобки и применение `apply` над таким значением можно рассматривать и соответственно как образование отложенного вычисления и приведение его в действие.

Отметим, что в Рефале-6 применение описанного приема облегчается тем, что обращение к функции (подобной) `apply` не нужно писать явным образом — оно подразумевается.

Для примера использования `apply` построим общую функцию `reduce` обхода последовательности с применением к элементам некоторой функции (под этим же именем или `fold`, `insert` или `accumulate` подобная функция имеется практически в любом функциональном языке):

```
reduce {
  t.f t.1 t.2 e.3 = <reduce t.f <apply t.f t.1 t.2> e.3>
; t.f e.1         = e.1}
```

Теперь воспользуемся `reduce` для вычисления значения числа по заданному основанию счисления и последовательности цифр. Такое вычисление удобно делать по правилу Горнера — фактически вычисляется многочлен с коэффициентами-цифрами — так что соорудим функцию `horn` для повторяющегося основного действия, а именно умножение текущего результата `s.v` на основание `s.r` и добавление значения новой цифры `s.d`:

```
horn {s.r s.v s.d = <+ <* s.r s.v> s.d>}
```

Искомое вычисление состоит в вызове `reduce` с частично примененной `horn` и цифрами. Так, `<reduce (horn 10) 3 5 8>` есть `358`, а `<reduce (horn 16) 3 5 8>` — `856`.

Развитие функционального стиля с применением данного подхода ограничено отсутствием вложенных и безымянных (но с параметрами) функций. Положим, вычисление значения числа как выше нужно оформить как самостоятельную функцию `rad-number` с параметрами `s.r` — основание и `e.d` — цифры, так что нахождение чисел записывалось бы несколько проще и более читаемо: `<rad-number 10 (3 5 8)>`, `<rad-number 12 (7 2 6 9)>` и т. д.

Вспомогательную функцию `horn` хотелось бы иметь вложенной в `rad-number`, а лучше всего — ввиду ее небольшого размера и однократного цитирования — и не именовать, но ни то, ни другое невозможно. А раз `horn` самостоятельна, то и значение `s.r` нужно передавать ей все так же явным образом и, более того, именно как первый аргумент. В итоге:

```
rad-number {s.r (e.d) = <reduce (horn s.r) e.d>}
```

где `horn` — та самая функция, которую мы уже написали.

Список литературы

[1] Рефал-2 – главный сайт. http://refal.net/%7Ebelous/refal2-r.htm

[2] Примеры программ на Рефале-2. http://www.cnshb.ru/vniitei/sw/refal

[3] Рефал-5 – главный сайт. http://botik.ru/pub/local/scp/refal5/refal5.html

[4] Turchin V. F. REFAL-5 programming guide & reference manual.
http://refal.botik.ru/book/html

[5] Турчин В. Ф. РЕФАЛ-5. Руководство по программированию и справочник (перевод несколько устаревшего в отношении определения языка варианта книги [4]).
http://refal.org/rf5_frm.htm

[6] Рефал-6 – главный сайт. http://refal.net/%7Earklimov/refal6

[7] Климов А. Программирование на языке Рефал. (описание Рефала-6)
http://refal.net/%7Earklimov/refal6/manual.htm

[8] Рефал+ – главный сайт. http://rfp.botik.ru

[9] Викисайт, посвящённый развитию Рефала (в основном Рефал+).
http://wiki.botik.ru/Refaldevel

[10] Гурин Р., Романенко С. Язык программирования Рефал Плюс.
http://wiki.botik.ru/Refaldevel/RefalPlusBook

[11] Немытых А. Лекционные записки по Рефалу.
http://www.botik.ru/pub/local/scp/ugp/seminars.html

[12] Немытых А. Введение в Рефал с примерами задач для школы.
ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_informatica_N9_2008_pp25-32.pdf

[13] Корлюков А. Лекционные записки по Рефалу. http://www.refal.net/%7Ekorlukov/refbook

[14] Turchin V. Refal: the language for processing XML documents.
http://www.math.bas.bg/bantchev/place/refal/refal-for-xml.pdf

[15] Refal-SciTE и Refal-PHP. http://refal.net/%7Ebelous

[16] Скоробогатов С., Чеповский А. Язык Refal с функциями высшего порядка.
http://iu9.bmstu.ru/science/refal.php

[17] Язык программирования D-Refal. http://ulm.uni.udm.ru/%7Esoft/d-refal

[18] Сообщество «Рефал/Суперкомпиляция». http://refal.net. Ссылки на реализации и публикации и другие ресурсы

[19] «Институт Рефала». http://refal.botik.ru Онлайн библиотека и другие ресурсы по Рефалу, нормальным алгорифмам и пр.

[20] Ефимов Г. Б., Зуева Е. Ю., Щенков И. Б. Из истории развития и применения компьютерной алгебры в Институте прикладной математики им. М. В. Келдыша. http://www.ict.edu.ru/ft/004313/prep2003_27.pdf

[21] Turchin V. F. A supercompiler system based on the language Refal. ACM SIGPLAN Notices, Vol. 14 (1979), No. 2, 46-54. http://www.math.bas.bg/bantchev/place/refal/sc-based-on-refal.djvu

[22] Turchin V. F. The concept of a supercompiler. ACM Transactions on programming languages and systems, Vol. 8 (1986), No. 3, 292-325. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.6414