Короткая экскурсия в функциональное программирование

Ю.А.Широков

7 ноября 2013 г.

Оглавление

1. Зачем нужно функциональное программирование

> Программы должны писаться для того,чтобы их читали люди, и только во вторую очередь для выполнения машиной.
>
> Х.Абельсон, Дж.Дж.Сассман

1.1. В погоне за ясностью

Понимаем ли мы программы, которые пишем?

Если речь о маленьких программах, решающих не слишком сложные задачи, то обычно понимаем, конечно. Чтобы сделать непонятной простую программу, надо специально постараться. Кстати, некоторые и стараются — есть даже специальные шуточные языки программирования, на которых самые нехитрые программы тяжело писать и тем более читать. Например, brainfuck, на котором программа, выводящая на экран строку «Hello, world!», выглядит так:

```
+++++ +++++
[
    > +++++ ++
    > +++++ +++++
    > +++
    > +
    <<<< -
]
> ++ .
> + .
+++++ ++ .
.
+++ .
> ++ .
<< +++++ +++++ +++++ .
```

```
> .
+++ .
----- - .
----- --- .
> + .
> .
```

Или whitespace, в котором конструкции языка записываются невидимыми символами (пробел, табуляция, перевод строки), а все видимые символы используются только для комментариев.

Но с большими задачами человеку становится уже не до шуток. Грустно, но у нашего мозга много ограничений, здорово мешающих при занятиях математикой и программированием. Например, кратковременная память среднего человека может удерживать только около семи однотипных вещей (психологи называют это ограничение «кошелёк Миллера»).

Поэтому большую программу приходится писать по кусочкам. Чем больше эти кусочки, чем сильнее они связаны друг с другом, тем проще запутаться и ошибиться. На самом деле, даже самые лучшие программисты часто делают ошибки в программах, которые они пишут.

Чтобы найти уже существующие ошибки и не наделать новых, дописывая программу, программист должен прочитать её код и понять, что он делает. Потому что даже если он сам писал эту программу, то, как именно устроен каждый её кусок, он довольно быстро забывает. А уж если программу пишут вместе несколько программистов, то каждому обычно приходится сначала довольно долго читать чужой код, и только потом дописывать к нему свой.

Можно, конечно, попробовать упростить будущим читателям программы работу, написав комментарии. Вот только часто, поправив программу, поправить комментарии забывают, и они только сбивают с толку. А ведь ещё бывают случаи, когда комментарий с самого начала был неверен!

Поэтому язык программирования должен быть таким, чтобы программа легко читалась.

Транслятор — программа, переводящая текст на языке программирования в машинные команды.

Люди начали пытаться сделать языки программирования выразительными и простыми для чтения сразу же, как только мощность компьютеров позволила писать трансляторы для таких языков. Ещё в конце 1960-х годов Эдсгер Дейкстра предложил отказаться от оператора GOTO (переход на метку), чтобы сделать структуру программ понятнее.

Дело в том, что от возможности «прыгнуть» в любое место программы

Рис. 1.1.: Использование GOTO опасно (`http://xkcd.com/292/`, русский перевод с `http://xkcd.ru/292/`).

из любого другого больше вреда, чем пользы. Пользуясь этой возможностью, можно создать программу, про которую будет очень тяжело понять, как она работает, да и последствия её работы могут быть непредсказуемыми (см. рис. 1.1).

Дейкстра и его единомышленники призывали использовать в программах только последовательное исполнение, циклы (операторы `for`, `while` и т.п.), ветвление (операторы `if`, `case`) и вызовы функций. Такой стиль назвали «структурным программированием», потому что он помогал сделать наглядной структуру программы.

Некоторое время даже достаточно большую по тогдашним меркам программу можно было сохранить понятной, соблюдая правила структурного стиля.

Но компьютеры становились мощнее, программы для них — больше, и правил структурного программирования стало недостаточно. Тогда программисты начали выдумывать новые способы сохранить программы понятными.

Некоторые из этих способов оказались не слишком удачными, как, например, попытка приблизить язык программирования к человеческому языку. Результат этой попытки, Кобол, оказался многословным и не слишком выразительным. (К сожалению, к моменту, когда это стало всем ясно, на Коболе уже написали огромное количество программ, и некоторые из них используются даже сейчас.)

Другие идеи сработали. Например, идея авторов языков Симула и Smalltalk: разделить код программы на кусочки, работающие с какой-то одной структурой данных, и разрешить этим кусочкам общаться между собой только с помощью чётко оговоренного набора сообщений. Тогда каждый такой кусочек можно переделывать как угодно, не беспокоясь, что сломаешь что-то в других частях программы — главное, чтобы его ответы на сообщения снаружи оставались такими же, как были.

Такие кусочки стали называть «объектами», а стиль программирования, при котором программа так устроена — «объектно-ориентированным программированием (ООП)».

Объектно-ориентированное программирование сильно упростило программистам жизнь, и в 1980-е, а особенно — в 1990-е годы очень многие программисты перешли на объектно-ориентированные языки. Но со временем стало ясно, что такие языки хорошо подходят не для всех задач. Кроме того, компьютеры стали ещё мощнее, программы — ещё больше, и программистам захотелось найти ещё какие-нибудь способы увеличить выразительность языков программирования.

1.2. Выгоды функционального программирования

Функциональное программирование (часто для краткости пишут просто ФП) — один из способов сделать код коротким и понятным одновременно. Есть у него и другие полезные качества, о которых — чуть позже.

Главная идея ФП очень простая: программу будет гораздо проще читать и отлаживать, если вызов функции с одними и теми же аргументами всегда будет возвращать одно и то же значение[1].

Идея эта довольно старая, и языки, в которых можно писать такие программы, появились довольно давно. Но первые трансляторы этих языков порождали не слишком быстрые и очень жадные до памяти программы, и большинство программистов старалось с ними не связываться. Возились с функциональными языками в основном математики и специалисты по искусственному интеллекту.

Но понемногу ситуация стала меняться. Во-первых, трансляторы функциональных языков становились всё совершеннее и порождали всё более и более компактные и шустрые программы. Во-вторых, мощность компьютеров выросла настолько, что требования этих языков к памяти и процессорному времени стали казаться довольно скромными. Наконец, в-третьих, языки, созданные во времена, когда программисты тряслись над каждой ячейкой памяти и каждым тактом процессора, сами обросли множеством механизмов, жадных до памяти и процессорного времени.

Многие из этих механизмов, кстати, были позаимствованы из функциональных языков. Более того, обнаружили такую закономерность: код большой программы на низкоуровневом языке, которую пишет грамотный программист, со временем начинает всё больше напоминать

[1] Я совершенно сознательно не даю здесь строгого определения функционального языка; если читателю удобнее сначала ознакомиться с основами теории, а уже потом применять её на практике, он может заглянуть на страницу 80.

в одних местах код программы на высокоуровневом языке, а в других — транслятор такого языка. Так что довод «функциональные языки — слишком высокоуровневые» перестал работать.

Кроме того, появились задачи, которые могут нормально работать только на десятках или даже тысячах процессоров: Интернет-сервисы, расчёт спецэффектов для кино, расчёт поведения сложных органических молекул в биоинформатике. Функциональные языки очень хорошо подходят для таких задач.

> Любая достаточно сложная программа на Си или Фортране содержит заново написанную, неспецифицированную, глючную и медленную реализацию половины языка Common Lisp.
>
> Десятое правило Гринспуна

Функциональные языки делают невозможными многие типы неприятных ошибок, часто встречающихся в программах на языках вроде C, Java или Python.

Программы на функциональных языках можно оптимизировать методами, которые сложно или даже невозможно использовать в прочих языках программирования.

Наконец, для программы на функциональном языке можно гораздо надёжнее проверить правильность её работы, чем для программ на языках других типов.

Разумеется, за всё приходится платить — программы на функциональных языках очень сильно отличаются от программ на привычных большинству программистов языках вроде C, Паскаля и Java. Но, несмотря на это, функциональные языки, такие, как Эрланг, Haskell, F#, Scala и Clojure, становятся всё популярнее, а в традиционные языки, такие, как Питон, C#, Ruby, добавляют всё больше элементов функционального программирования.

Разбираться, откуда именно берутся выгоды функционального программирования и так ли странно устроены программы на функциональных языках, лучше всего на практике. Такой практикой я и предлагаю заняться.

Эта книжка предлагает вам примеры приёмов функционального программирования на языке Эрланг с короткими комментариями и несложные задачи, позволяющие опробовать эти приёмы самостоятельно. Воспринимайте её как короткую экскурсию, которая позволит вам составить общее представление о мире ФП и решить, интересно ли вам будет познакомиться с ним поближе.

2. Знакомство с языком Эрланг

> Может показаться, что при разработке языков программирования нужно руководствоваться тем, что машина может сделать. Но если учесть, что язык программирования — это мост между пользователем и компьютером, это инструмент пользователя, становится понятно, что не менее важно учитывать, что человек может придумать.
>
> Э.В.Дейкстра,
> «Программирование как вид человеческой деятельности»

2.1. Почему Эрланг

Язык Эрланг редко используют для обучения программированию. На фоне стройных и изящных академических языков вроде Scheme и Haskell он выглядит откровенно неказисто. Эрланг — «рабочая лошадка». С самого начала его приспосабливали для реальных задач, жертвуя последовательностью и строгостью ради мощи и удобства.

Но именно поэтому он как нельзя лучше подходит для короткой экскурсии вроде нашей. У создателей Эрланга есть воронья привычка тащить в язык все замечательные блестящие штучки из других языков, которые только попадутся им на глаза. Нет, наверное, функционального языка, из которого Эрланг не позаимствовал бы какую-нибудь интересную особенность. Кстати, этим Эрланг похож на другой популярный язык — Питон. В результате оба этих языка позволяют просто и естественно выразить почти любую программистскую идею (в том числе и не слишком удачную, что многих раздражает).

Кроме того, Эрланг (в отличие, скажем, от Haskell) прост в изучении.

Говорят, что у среднего программиста уходит где-то неделя от момента, когда он впервые видит учебник по Эрлангу, до момента, когда он становится способен использовать этот язык в работе. А нам для целей этой книги понадобится узнать о языке гораздо меньше, чем профессиональному программисту.

Да, Эрланг терпим к мелким неряшливостям, незамысловат (а местами, как мы увидим, даже простоват[1]). Но именно это позволит нам сходу приступить к применению приёмов функционального программирования в собственном коде, не углубляясь в теоретические тонкости.

Агнер Краруп Эрланг (1878–1929) — датский математик, основатель теории массового обслуживания (ТМО). Кроме языка программирования, в его честь названа единица измерения трафика в телекоммуникационных сетях.

Если честно, я думаю, что и более плотному знакомству с другими функциональными языками такой подход скорее поспособствует — по крайней мере, будет понятно, ради чего стоит эти языки учить.

Ну и, наконец, Эрланг — не только функциональный, но и параллельный язык. В этой книжке об этой его стороне говорится совсем немного, но вообще-то именно возможности параллельного программирования считаются главной сильной стороной Эрланга. Задач, в которых не обойтись без параллельных вычислений, с каждым годом всё больше, а Эрланг считается одной из лучших сред для организации таких вычислений, так что знакомство с ним будет полезным почти любому человеку, имеющему дело с программированием.

2.2. Диалог с интерпретатором

В старых учебниках по программированию можно прочитать, что трансляторы языков (то есть программы, которые переводят текст на языке программирования в машинные команды) бывают двух типов: интерпретаторы и компиляторы. Компилятор переводит в машинный язык всю программу целиком, а интерпретатор берёт команды из файла с программой одну за другой и выполняет их «на ходу».

Сейчас чистых интерпретаторов почти не осталось. Программы на современных языках программирования переводятся либо в машинные команды для того компьютера, на котором они будут исполняться (как, скажем, в C++ или Haskell), либо в команды воображаемой «машины»,

[1]Например, проблема обработки строк решена с какой-то совсем уж кавалерийской лихостью.

которые на настоящем компьютере будет исполнять специальная программа (как в Python или Java). Её так и называют — «Python-машина», «Java-машина» и т.д. Транслятор Эрланга устроен точно так же — это компилятор, переводящий программы в машинный код виртуальной «Эрланг-машины».

Результат работы *компилятора* — файл с программой на машинном языке, а *интерпретатор* берёт команды из файла с программой одну за другой и выполняет их «на ходу».

Но для некоторых языков (в том числе и для Эрланга) написаны специальные программы, в которых можно вводить выражения языка одно за другим и сразу видеть результат их вычисления. Они называются «интерактивными интерпретаторами». Интерактивный интерпретатор — один из главных инструментов программиста на Эрланге. Знакомиться с Эрлангом тоже проще всего с помощью этой программы. Если запустить её (в большинстве операционных систем это делается командой «erl»), на экране окажется информация о версии и приглашение интерпретатора[2]:

```
1>
```

Любое выражение в Эрланге должно заканчиваться точкой. Например:

```
1> 2 + 3.
5
```

Кстати, явных ограничений на размер числа, в отличие от языков вроде C или Java, в Эрланге нет.

```
2> 10000000000000000000000000000000000000000000000000000000000000000000000000000000 * 100500.
1005000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Вещественные числа представляются примерно так же, как в других языках.

```
3> 2.5 + 2.5.
5.0

4> 5 / 2.
2.5
```

[2]Приглашение приводится и во всех последующих примерах, чтобы удобно было отличать то, что вводит пользователь, от того, что выводит компьютер. Запомните это и не считайте приглашение частью выражения.

Вещественное число можно урезать до ближайшего снизу целого или округлить:

```
5> trunc(2.5).
2

6> round(2.5).
3
```

А целое — разделить нацело или посчитать остаток от его деления на другое целое:

```
7> 27 div 5.
5

8> 27 rem 5.
2
```

С помощью функций из модуля `math` (подробный рассказ о модулях будет ниже) можно возводить числа в степень и извлекать из них квадратный корень:

```
9> math:pow(5, 2).
25.0

10> math:sqrt(25).
5.0
```

Как и в большинстве других современных языков программирования, в Эрланге есть логические значения — `true` («истина») и `false` («ложь»). Это именно логические значения, а не значения логического типа — логического типа в Эрланге нет[3]. Зато в Эрланге есть атомы — любое слово, начинающееся с маленькой латинской буквы и не являющееся зарезервированным словом Эрланга, можно использовать в коде как оно есть, интерпретатор будет считать его значением типа «атом».

Атомы обычно используют в ситуациях, когда в каких-то данных возможно всего несколько вариантов значений — скажем, названия цветов, степени важности сообщения или должности сотрудников. Так, например, в программе, работающей с небольшим набором цветов, мы могли бы определить атомы `red`, `orange`, `yellow` и т.д. Те из читателей, кто знаком с языком C, возможно, вспомнят перечислимые типы (`enum`),

[3]Читатель, знакомый с Паскалем или Питоном, знает, что такое значения логического типа — это `true` и `false` в Паскале и `True` и `False` в Питоне. Читателю, знакомому только с языком C, где логические выражения возвращают 0 или 1, лучше просто попытаться понять приводимые ниже примеры.

которые используются в C для решения похожих задач. Обратите внимание, что атомы — это именно значения, а не имена; «присваивать значения» атомам нельзя.

Некоторое количество атомов в Эрланге уже определены, в том числе `true` и `false`. Именно логические значения получаются в результате операций сравнения...

```
11> 2 > 3.
false

12> 2 < 3.
true
```

...и проверки на равенство:

```
13> 2 =:= 3.
false

14> 3 =:= 3.
true

15> 3 =/= 3.
false

16> 2 =/= 3.
true
```

Обратите внимание, что оператор проверки на равенство в Эрланге выглядит как «=:=». Выглядящий более привычно для знакомых с C, Java или Питоном оператор «==» в Эрланге тоже есть, но наверное, единственное, для чего он применяется — сравнение вещественных чисел с целыми:

```
17> 3 =:= 3.0.
false

18> 3 == 3.0.
true
```

Для логических выражений доступен стандартный набор операций — `not` (отрицание), `and` (и), `or` (или), `xor` (исключающее или).

```
19> not false.
true
```

```
20> true and false.
false

21> true or false.
true

22> true xor true.
false

23> true xor false.
true
```

2.3. Переменные... или не переменные?

Эрланг считает именами переменных всё, что написано с большой латинской буквы. Присвоить переменной значение можно с помощью оператора «=»:

```
24> X = 5.
5
```

После чего переменную можно использовать в эрланговских выражениях:

```
25> X.
5

26> X + 2.
7

27> X * X.
25
```

Но если мы попытаемся, например, увеличить значение переменной на 1 привычным нам по другим языкам способом, нас ждёт сюрприз:

```
28> X = X + 1.
** exception error: no match of right hand side value 6
```

Это вступил в действие один из принципов функционального программирования: функциональное программирование не предполагает произвольного изменения значений переменных. После того, как переменная связана со значением (именно так предпочитают говорить об этой операции, избегая термина «присваивание»), изменить значение переменной нельзя.

Императивный язык — язык, программа на котором состоит из инструкций по изменению состояния программы. Как структурные (Паскаль, C), так и объектно-ориентированные (Питон, C++, Java) языки являются императивными.

Другими словами, понятие «переменной» в Эрланге ближе к математической идее переменной, чем к переменным в традиционных языках программирования: встречая переменную в математическом уравнении, скажем, $x^2 = 2x$, мы предполагаем, что везде, где употребляется x, имеется в виду одно и то же значение; привычные нам по императивным языкам $x = x + 1$ и подобные ему выражения в этом смысле абсурдны. Вот и переменной `X` соответствует одно и то же значение во всём том фрагменте, в котором она используется.

Важно, однако, понимать, что переменные в функциональных языках — это именно переменные, а не константы, как иногда поспешно заключают люди, пытающиеся разобраться с функциональным языком после программирования на языках императивных. Каждый раз, когда исполняется фрагмент кода, содержащий переменную, она может принимать новое значение — точно так же, как x в математическом уравнении может обозначать любое из его решений, которых может быть сколько угодно.

2.4. Кортежи

Кортеж, или упорядоченная $n-$ка — объединение нескольких значений. Кортеж записывается в фигурных скобках:

```
29> Tuple1 = {1, 2, 3}.
{1,2,3}
```

Нет никаких ограничений на хранение в одном кортеже значений разных типов...

```
30> Tuple2 = {1.0, 2, ''abc''}.
{1.0,2,''abc''}
```

...и на вкладывание кортежей в кортежи:

```
31> {0, Tuple1, Tuple2}.
{0,{1,2,3},{1.0,2,''abc''}}
```

Среди программистов на Эрланге считается хорошей практикой включать в кортеж первым элементом атом, описывающий его назначение:

```
{answer, ''life the universe and everything'', 42}.
```

2.5. Списки

Список, как и кортеж — это способ объединить несколько значений. Но если для выполнения операций с кортежем мы должны знать длину этого кортежа, то для списков мы можем описать операции, работающие со списком любой конечной длины (некоторые функциональные языки позволяют описать и операции для работы с «бесконечными» списками, и эти операции даже бывают иногда удобны и полезны; в Эрланге, правда, такого нет). На практике, конечно, длина списка ограничена объёмом памяти машины, на которой работает наша программа.

Список — главная структура данных в большинстве функциональных языков.

Про список полезно думать как про структуру данных, состоящую обычно из *головы* и *хвоста*. Голова списка — это его первый элемент; хвост — все остальные. Список может быть и пустым. У пустого списка хвоста нет. У списка из одного элемента значение головы — этот элемент, а значение хвоста — пустой список. Вообще, последним элементом любого списка считается пустой список (`[]`). Основные операции со списком — узнать значение головы списка, узнать значение хвоста списка и составить новый список, объявив значения его головы и хвоста.

Вот как это делается в Эрланге.

Самый простой способ записать список — перечислить его элементы в квадратных скобках через запятую:

```
32> [1, 2, 3].
[1,2,3]
```

Элементами списка могут быть символы...

```
33> [’a’, ’b’, ’c’].
[a,b,c]
```

...строки...

```
34> [’’hello’’, ’’world’’].
[’’hello’’,’’world’’]
```

...и даже другие списки:

```
35> [[1, 2, 3], [4, 5, 6]].
[[1,2,3],[4,5,6]]
```

Элементы списка могут быть разного типа:

```
36> [1, 2, [’a’, 3, ’’hello’’]].
[1,2,[a,3,’’hello’’]]
```

Прибавить голову к уже существующему списку можно с помощью операции «|»:

```
37> L = [2, 3].
[2,3]

38> [1|L].
[1,2,3]
```

Можно приставить голову к пустому списку, получив список из одного элемента:

```
39> [1|[]].
[1]
```

Нетрудно видеть, что строгое определение списка можно сформулировать так: список либо пуст, либо состоит из головы и хвоста, причём хвост списка — тоже список. То есть запись `[1,2,3]` — это, на самом деле, сокращение для `[1|[2|[3|[]]]]`. Интерпретатор Эрланга позволяет нам убедиться, что это действительно так:

```
40> [1|[2|[3|[]]]].
[1,2,3]
```

Это очень важная идея, её необходимо запомнить — мы неоднократно будем использовать её в дальнейшем.

2.6. Сопоставление с образцом

Возможно, читатель заметил, что эрланговский оператор «=» ни разу не был назван оператором присваивания. Это не случайно — на операторы «:=» языка Паскаль или «=» Питона, Java или C он похож только на первый взгляд. На самом деле «=» в Эрланге обозначает гораздо более мощную и интересную вещь — *сопоставление с образцом*.

При выполнении этой операции Эрланг вычисляет выражения по обе стороны от оператора «=». Если эти выражения состоят только из констант и уже привязанных ко значениям переменных, просто проверяется, что их значения совпадают:

```
41> 5 = 5.
5

42> 5 = 7.
** exception error: no match of right hand side value 7
```

Успешное сопоставление вернёт значение, которое вычисляют оба выражения; неудачное приведёт к ошибке.

Если же справа от «=» найдутся несвязанные переменные, Эрланг попробует связать с ними значения из выражения слева от «=» таким образом, чтобы значения выражений совпали. Если это удастся, переменные останутся связанными, если нет, это опять же приведёт к ошибке.

Если справа от оператора «=» указана только одна несвязанная переменная, результат действительно похож на оператор присваивания:

```
43> X = 5.
5

44> X.
5
```

Но с помощью сопоставления с образцом можно делать и более сложные вещи. Например, «разобрать на части» кортеж:

```
45> {flight, From, To} = {flight, ''Moscow'', ''New York''}.
{flight,''Moscow'',''New York''}

46> From.
''Moscow''

47> To.
''New York''
```

Или разделить список на голову и хвост:

```
48> [Head|Tail] = [1, 2, 3].
[1,2,3]

49> Head.
1

50> Tail.
[2,3]
```

И даже вещи вроде «запомнить голову списка, если она совпадает с первым элементом хвоста»:

```
51> [Y|[Y|_]] = [1, 1, 2].
[1,1,2]
```

```
52> Y.
1

53> [Z|[Z|_]] = [1, 2, 3].
** exception error: no match of right hand side value [1,2,3]

54> Z.
* 1: variable 'Z' is unbound
```

Обратите внимание на символ _ в этих примерах. Он называется «анонимной переменной» и считается совпадающим с любым образцом. Анонимная переменная употребляется там, где мы не собираемся использовать часть значения, которая будет подставлена в неё при сопоставлении.

```
55> _ = 5.
5

56> _ = 7.
7

57> _ = ''hello''.
''hello''

58> [_, Second, _] = ['a', 'b', 'c'].


59> Second.
b
```

2.7. Строки

То, как в Эрланге реализованы строки — пожалуй, одна из самых неудачных, если не сказать уродливых, сторон языка. Эрланг считает строками любые списки целых чисел, где все числа попадают в диапазон печатных кодов таблицы символов ASCII:

```
60> [104, 101, 108, 108, 111].
''hello''
```

В этом примере все числа являются ASCII-кодами печатных символов, поэтому Эрланг по умолчанию отобразит не сами числа, а символы, кодами которых они являются.

Специальной командой можно заставить Эрланг считать строками списки целых чисел, в которых все числа попадают в диапазон символов Unicode, но и это не делает такие «строки» сильно удобнее в использовании.

На практике их применяют мало, стараясь пользоваться вместо них битовым типом, о котором рассказывается ниже.

Зато с эрланговским представлением строк связан удобный, хотя и редко нужный инструмент для работы с текстом — оператор «$», позволяющий получить код символа:

```
61> $a.
97

62> $ю.
192

63> $#.
35
```

Своеобразие эрланговских «строк» заставляет использовать именно получаемые таким образом коды символов в выражениях сопоставления с образцом.

2.8. Ввод и вывод

Начнём с традиционного примера вывода строки на экран.

```
64> io:format(''Hello, world!~n'').
Hello, world!
ok
```

Строка «Hello, world!» выводится на экран как есть, а «~n» — обозначение символа перевода строки.

Если нам надо вывести на экран значение переменной или выражения, мы должны передать функции io:format два аргумента: первый — строка, в которой мы оставим места для подстановки значений, а второй — список самих значений:

```
65> A = ''user''.
''user''
```

```
66> io:format(''Hello, ~s!~n'', [A]).
Hello, user!
ok
```

Строка с оставленными «дырками» для значений называется *строкой формата*. Начинающиеся с тильды (знака «~») последовательности — это *команды формата*. Во втором примере мы использовали две команды формата — команду «~s», означающую «здесь будет выведено значение типа ”строка”», и уже знакомую нам по второму примеру команду «перевод строки» («~n»).

Большинство значений Эрланга выводятся на экран с помощью команды формата «~w» (или «~p», делающей примерно то же самое, но выводящей длинные значения в несколько строк):

```
67> io:format(''~w + ~w = ~w~n'', [1, 2, 1+2]).
1 + 2 = 3
ok

68> io:format(''This is a list: ~w~n'', [[1, 2, 3]]).
This is a list: [1,2,3]
ok
```

Значения подставляются в строку формата в том порядке, в котором они следуют во втором аргументе функции `io:format`: на место первой команды подстановки помещается первый элемент списка, на место второй команды — второй элемент и т.д.

Для ввода значений используются функции `io:read/1` и `io:fread/2`.

Функция `io:read/1` принимает текст выводимой для пользователя подсказки и ожидает на вход корректное выражение Эрланга (заканчивающееся точкой). Если выражение введено и синтаксически правильно, она возвращает кортеж `{ok, Value}`, где `Value` — введённое значение; в противном случае — кортеж `{error, Reason}`, где `Reason` — причина ошибки:

```
69> {ok, L} = io:read(''Enter a list, please: '').
Enter a list, please: [1, 2, 3].
{ok,[1,2,3]}

70> L.
[1,2,3]

71> {ok, {answer, N}} = io:read(''What is the answer? '').
What is the answer? {answer, 42}.
{ok,{answer,42}}
```

```
72> N.
42

73> io:read('> ').
> {not a tuple.
{error,{1,erl_parse,['syntax error before: ','tuple']}}
```

Функция `io:fread` принимает два аргумента: подсказку и строку формата. В случае, если ей удастся привести введённое значение к типу, указанному строкой формата, она возвращает кортеж `{ok, Values}`, где `Values` — список полученных значений. В случае же ошибки она, как и `io:read/1`, возвращает `{error, Reason}`, где `Reason` — причина ошибки:

```
74> io:fread('Enter a number: ', '~d').
Enter a number: 5
{ok,[5]}

75> io:fread('Enter a number: ', '~s').
Enter a number: 5
{ok,['5']}

76> io:fread('Enter a number: ', '~d').
Enter a number: Foo
{error,{fread,integer}}
```

Не старайтесь проверять, что вернули функции ввода — правильное значение или ошибку. Сразу сопоставляйте их результат с кортежем `{ok, Value}`, где `Value` — необходимое вам значение.

```
77> {ok, [Name]} = io:fread('Enter your name: ', '~s').
Enter your name: Foo
{ok,['Foo']}

78> io:format('Hello, ~s~n', [Name]).
Hello, Foo
ok
```

С помощью функции `io:format` можно делать ещё множество полезных вещей, например, выводить числа в разных системах счисления:

```
79> io:format('~.10B~n', [10]).
10
ok
```

```
80> io:format(''~.2B~n'', [10]).
1010
ok

81> io:format(''~.16B~n'', [10]).
A
ok
```

Это пример команды с аргументом: команда `B` выводит число в нужной нам системе счисления — мы должны указать `.10` для десятичной, `.2` для двоичной и `.16` для шестнадцатеричной.

2.9. Битовые строки

Битовые строки — одна из самых удобных и замечательных возможностей языка Эрланг. Это мощный и красивый инструмент, предназначенный прежде всего для работы с бинарными форматами файлов и сетевых протоколов («родина» Эрланга — телекоммуникационная индустрия, где умение работать с такими форматами жизненно важно), но полезный для очень многих задач.

Записываются битовые строки в двойных угловых скобках. Битовая строка может быть либо пустой, либо состоящей из одного или нескольких сегментов, разделённых запятыми.

```
82> <<>>.
<<>>

83> <<1>>.
<<1>>

84> <<1, 2, 3>>.
<<1,2,3>>
```

Мы можем указать для сегмента длину и тип, что позволяет очень легко читать и изменять данные в двоичном виде, извлекать из него значения и переводить их обратно в двоичный вид. Например, вот так мы можем с помощью битовых строк разделить одно восьмибитное значение на два куска по четыре бита:

```
85> <<A:4, B:4>> = <<2#10101100>>.
<<''¬''>>
```

```
86> io:format(''~.2B~n'', [A]).
1010
ok

87> io:format(''~.2B~n'', [B]).
1100
ok
```

Выражение `:4` после имён переменных `A` и `B` это как раз и есть указание длины (в битах) извлекаемого из битовой строки сегмента.

С помощью конструкции `2#` мы указываем, что записываем число в двоичном виде (странный символ, в который это число превращает Эрланг — результат обсуждавшейся на стр.19 манеры интерпретатора Эрланга воспринимать небольшие целые числа как коды символов из таблицы ASCII).

Как и в случае со списками, можно с помощью сопоставления с образцом разделять битовые строки на «голову», помещаемую в одну переменную, и «хвост» неизвестной длины, помещаемый в другую. Мы можем «откусывать» от начала битовой строки любое число битов, при условии, что в «хвосте» останется целое число байт:

```
88> <<X:2, Y:6, Rest/binary>> = <<1, 2, 3, 4, 5>>.
<<1,2,3,4,5>>

89> X.
0

90> Y.
1

91> Rest.
<<2,3,4,5>>
```

В этом примере число 1, хранящееся как бинарное значение 00000001, мы разделили на фрагменты длиной два бита (переменная `X`) и шесть бит (переменная `Y`), а остаток битовой строки разместили в переменной `Rest`.

Обратите внимание на то, что «хвост» битовой строки надо помечать с помощью конструкции `/binary`, чтобы обозначить, что это тоже битовая строка.

2.10. Записи

Представим, что нам надо хранить координаты точки на плоскости. Можно, конечно, поместить их в список из двух элементов: `[X, Y]`. Но при чтении кода будет трудно понять, где какая координата. Кроме того, велик риск, что функции, работающей с такими списками, по ошибке передадут какой-нибудь другой список, а компилятор никак не поможет нам поймать эту ошибку — он ведь ничего не знает о том, что в этом конкретном списке мы храним именно координаты.

Можно хранить координаты в кортеже: `{X, Y}`. Тут вероятность спутать такие данные с чем-то ещё немного меньше, но проблема понимания кода остаётся: как, увидев кортеж `{3, 5}`, понять, где какая координата?

Можно добавить к каждому значению атом-аннотацию: `{{x, 3}, {y, 5}}`. Но код, сопоставляющий с образцом такие кортежи, будет тяжеловато читать и не слишком интересно писать (особенно если мы описываем не точку на плоскости, а, скажем, персонажа в компьютерной игре, и характеристик, которые мы храним, у нас десятки).

В Эрланге есть более удобный способ хранить такие значения: *записи* (records). Они довольно похожи на записи языка Паскаль или структуры Си.

Записи нельзя определять в интерпретаторе. Для создания записи нужно будет создать файл с именем, совпадающим с именем записи, и расширением `.hrl`. Предположим, наша запись, хранящая координаты точки, будет называться `point`. Тогда её определение будет выглядеть так:

```
-record(point, {x, y}).
```

и располагаться в файле `point.hrl`. Здесь `point` — это имя записи, а `x` и `y` — имена полей, хранящих координаты X и Y соответственно.

Чтобы воспользоваться определением записи `point`, надо дать в интерпретаторе команду `rr`, указав в качестве аргумента расположение и имя файла:

```
92> rr("point.hrl").
[point]
```

После этого можно создавать новые значения объявленного типа, указывая после знака # имя записи, и, по желанию, значения полей внутри фигурных скобок:

```
93> #point{x = 3, y = 5}.
#point{x = 3,y = 5}
```

Порядок следования полей значения не имеет, только имена:

```
94> #point{y = 5, x = 3}.
#point{x = 3,y = 5}
```

Если значения полей не указываются, в поле помещается атом `undefined`:

```
95> #point.
#point{x = undefined,y = undefined}
```

Но это поведение можно изменить, указав для полей значения по умолчанию:

```
-record(point, {x=0, y=0}).
```

Если мы после этого перечитаем файл с определением записи:

```
96> rr(''point.hrl'').
[point]
```

вместо `undefined` по умолчанию будут подставляться указанные нами значения:

```
97> #point{}.
#point{x = 0,y = 0}

98> #point{x = 5}.
#point{x = 5,y = 0}

99> #point{y = 5}.
#point{x = 0,y = 5}
```

Значения типа «запись» можно, естественно, присваивать переменным:

```
100> A = #point{x = 3, y = 5}.
#point{x = 3,y = 5}

101> A.
#point{x = 3,y = 5}
```

Извлечь из записи отдельные поля можно с помощью выражения, состоящего из знака #, имени записи и имени поля:

```
102> A#point.x.
3

103> A#point.y.
5
```

2.11. Комментарии

Как бы ни был выразителен код, комментарии ему всё равно необходимы. Компилятор Эрланга считает комментарием всё от символа % до конца строки:

```
104> math:sqrt(25) % Это --- комментарий.
104> .
5.0
```

3. Азбука ФП

Чтобы понять рекурсию, нужно понять рекурсию.

Народная мудрость

3.1. Функции

Попробуем определить какую-нибудь функцию. Например, функцию, возводящую число в квадрат. Это очень просто:

```
square(X) -> X * X.
```

Определение функции начинается с её имени (наша функция, стало быть, называется `square`). Дальше перечисляются аргументы (у этой функции аргумент один, `X`). После этого идёт символ «->», отделяющий объявление функции от её тела, а дальше само тело функции. Здесь это выражение `X * X`. Результат его вычисления будет возвращён как значение функции.

Давайте ей воспользуемся. Правда, оболочки интепретатора нам для этого будет уже недостаточно.

Дело вот в чём. Даже в одной программе функций обычно очень много. А большинство программ ещё используют сторонние библиотеки, в каждой из которых тоже огромное количество функций. Если бы Эрланг различал эти функции по только именам, был бы очень велик риск, что имена каких-то функций совпадут. Это было бы похоже на страну, где людей пытаются различать только по именам, без фамилий и прочих дополнительных данных. Поэтому, чтобы использовать определение функции в Эрланге, надо «дать ей фамилию» — включить в *модуль*.

Модуль — это наделённый определённым именем набор функций (на самом деле, не только функций, но это сейчас не так важно), обычно объединённых одной задачей. Каждый модуль описывается в своём файле. Давайте опишем модуль для нашей функции `square`.

```
-module(square).
-export([square/1]).
```

```
square(X) ->
    X * X.
```

Посмотрим, что здесь к чему. Кроме кода функции, в тексте модуля мы видим ещё две строки:

```
-module(square).
-export([square/1]).
```

Такие, начинающиеся с дефиса, строки — это специальные команды для компилятора Эрланга, они называются *директивами*. Первая из них (`module`) задаёт имя модуля (оно такое же, как и имя функции — `square`), а вторая — задаёт список функций, которые можно использовать не только внутри модуля, но и в любой другой программе на Эрланге или в интерактивном интепретаторе (нам сейчас нужно именно это).

Число, написанное после имени функции через косую черту — это *арность* функции, то есть количество аргументов, которые она принимает. В Эрланге можно задавать функции с одним и тем же именем, но разной арностью, поэтому, когда нам нужно точно указать функцию, мы должны указать её арность наряду с именем и модулем.

Арность функции — это количество её аргументов. Функции с одинаковым именем, но разной арностью для Эрланга — разные.

Этот код надо разместить в файле `square.erl`, расположенном в том же каталоге, в котором вы запускаете оболочку интерпретатора Эрланга. Теперь его необходимо загрузить в систему. Для этого модуль сначала компилируется, то есть переводится в «машинный код» Эрланг-машины:

```
105> c(square).
{ok,square}
```

После этого мы, наконец, можем воспользоваться нашей функцией:

```
106> square:square(2).
4

107> square:square(3).
9
```

Обратите внимание, что используется полное имя: имя модуля, потом двоеточие и имя самой функции.

Если в теле функции несколько выражений, они разделяются запятыми. Возвращён будет результат последнего из них. А предыдущие выражения могут использоваться для того, чтобы задать значения дополнительным переменным или выполнить операции, имеющие *побочный эффект* —

скажем, вывести что-то на экран, прочитать пользовательский ввод или отправить сообщение по сети.

Изменим функцию `square` так, чтобы она выводила на экран значение своего аргумента.

```
square(X) ->
    io:format(''X=~w~n'', [X]),
    X * X.
```

Теперь первая строка функции `square` выводит на экран содержимое переменной `X`.

Если мы перекомпилируем модуль `square` и заново загрузим его в память, а потом выполним функцию `square`, мы увидим, что её поведение немного изменилось:

```
108> square:square(2).
X=2
4

109> square:square(3).
X=3
9
```

Мы уже обсуждали на странице 17, что связывание переменных с их значениями производится в Эрланге с помощью оператора сопоставления с образцом. Связывание аргументов функции с их значениями при вызове — не исключение, в этой ситуации тоже применяется сопоставление с образцом. Причём можно написать несколько вариантов тела функции, из которых выполнится тот, для которого сопоставление с образцом пройдёт успешно.

Варианты функции разделяются точками с запятой, в конце последнего ставится точка. Вот, например, функция, принимающая название цвета и возвращающая название дополнительного к нему:

```
complementary(red) -> green;
complementary(orange) -> blue;
complementary(yellow) -> violet;
complementary(green) -> red;
complementary(blue) -> orange;
complementary(violet) -> yellow.
```

При вызове этой функции Эрланг будет сопоставлять аргумент с атомами `red`, `orange`, `yellow` и т.д., пока не произойдёт успешного сопоставления. После удачного сопоставления с образцом будет выполнено соответствующее тело функции, и дальнейшие варианты проверяться не будут.

Выражения внутри функции разделяются запятыми, варианты функции разделяются точкой с запятой. В конце каждой функции ставится точка.

После того, как мы определили эту функцию и загрузили содержащий её модуль, скажем, `colors`, мы сможем использовать её. Сделайте это, чтобы посмотреть, как работает функция, определённая через сопоставление с образцом:

```
110> colors:complementary(red).
green

111> colors:complementary(yellow).
violet

112> colors:complementary(violet).
yellow
```

В описании функций можно использовать анонимные переменные (начинающиеся с _; что это такое, объяснялось в разделе 2.6, на странице 19). Вот, например, функция, позволяющая не получать ошибки при попытке деления на ноль:

```
safe_division(_, 0) -> undefined;
safe_division(N, M) -> N / M.
```

Использовать здесь анонимную переменную не только можно, но и нужно: если бы мы вместо _ использовали бы в первом варианте функции именованную переменную, например, `N`, при компиляции такого кода Эрланг выдал бы нам предупреждение:

```
Warning: variable 'N' is unused
```

И действительно — в случае с делением на 0 нам ведь не важно, *что именно* мы собираемся разделить на 0, результат от этого не зависит. Поэтому связывание неиспользуемого параметра с именем только запутало бы код, от чего Эрланг и пытается нас предостеречь.

Ещё одно замечание по поводу функции `safe_division`. Надо предупредить, что в большинстве случаев использование такой функции в реальных вычислениях — не самая лучшая идея. Вообще, в сообществе Эрланг-программистов принято руководствоваться принципом «Let it fail» — «Пусть падает». В отличие, скажем, от программистов на Java, программисты на Эрланг обычно пишут код, который *не* обрабатывает собственные ошибки. Это связано с тем, что они сомневаются в способности программы самостоятельно разобраться с возникшей проблемой. Практика показывает, что так написанные программы оказываются более простыми в отладке и надёжными — «перехват ошибок» во время исполнения только затрудняет их поиск, потому что некорректное поведение в результате становится заметным сильно позже момента, когда

что-то пошло не так. Гораздо удобнее, когда программа падает прямо в момент ошибки (тем более, что в Эрланге программа может упасть *не целиком*, но об этом позже).

Важно ещё запомнить, что порядок, в котором расписаны варианты функции для разных образцов, важен: сначала Эрланг попробует первый образец, потом второй и т.д. Так что начинать описывать варианты надо всегда с самых конкретных. Например, в функции `safe_division` образец `(N,M)` успешно сопоставится во всех случаях, в том числе и тогда, когда M=0. Поэтому чтобы перехватить случай с M=0, мы проверяем его первым.

Задача 3.1 *Определите функцию `nand` (НЕ-И), таблица истинности для которой приведена ниже, не пользуясь встроенными булевыми функциями Эрланга.*

	true	false
true	*false*	*true*
false	*true*	*true*

Задача 3.2 **Пользуясь только функцией `nand` из задачи 3.1, определите собственные аналоги основных логических функций Эрланга — `and`, `or` и `not`.*

3.2. Ветвление

Как и в большинстве языков программирования, в Эрланге есть операторы для выражения идеи «если соблюдается какое-то условие, сделать одно, а в противном случае сделать другое» — операторы ветвления.

Если условие, которое надо проверить — это сложное выражение, возвращающее атомы `true` или `false`, используют оператор `if`. Вот как, например, можно переписать с его помощью функцию `safe_division` со страницы 31:

```
safe_division(N, M) ->
    if
        M =:= 0 -> undefined;
        true    -> N / M
    end.
```

При выполнении оператора `if` выражения (которых может быть и больше двух) будут проверяться в том порядке, в котором они записаны, пока не найдётся истинное (то есть возвращающее `true`). После этого будет выполнен код, следующий за этим выражением после оператора «->», и результат его исполнения будет считаться результатом

исполнения всего блока `if`. Для того, чтобы записать код, который должен быть выполнен, если ни одно из явно выраженных условий не верно, последним выражением в `if` записывают `true` — получается приблизительный аналог `else` в Си, Паскале или Питоне.

Если нужно просто проверить значение переменной или выражения, то удобнее оператор `case`. Вот так будет выглядеть функция, которая принимает целое число и возвращает строку «Even», если это число — чётное, и «Odd» — если нечётное:

```
oddeven(N) ->
    case N rem 2 of
        0  -> ''Even'';
        1  -> ''Odd'';
        -1 -> ''Odd''
    end.
```

Синтаксис оператора `case` такой: ключевое слово `case`, потом выражение, которое будет сопоставляться с образцами, потом ключевое слово `of`, потом образцы, с которыми будет сопоставляться выражение, и, после «->», действия, которые надо выполнить после успешного сопоставления. Заканчивается `case`, как и `if`, ключевым словом `end`.

Тело операторов `if` и `case` должно заканчиваться ключевым словом `end` и точкой. Варианты действий разделяются точкой с запятой, после последнего из них точка с запятой не ставится.

Важное замечание: и в случае с `if`, и в случае с `case` хотя бы одна ветка должна быть выполнена, иначе выражение завершится с ошибкой! То есть эти операторы гораздо меньше похожи на `if` и `case` императивных языков, чем может показаться на первый взгляд: у нас нет возможности сказать «если выполняется такое-то условие, сделай вот это, иначе *не делай ничего*». Связано это с тем, что в функциональных языках любое выражение должно возвращать какое-то значение, а ситуация, когда выражение может вернуть значение, а может и не вернуть — неправильная (в некоторых функциональных языках, таких как OCaml, Scala и Haskell, есть специальные способы надёжной работы с такими выражениями, но в Эрланге их нет).

Напоследок надо предостеречь от злоупотребления операторами `if` и `case`. Помните про сопоставление с образцом в объявлении функции — очень часто оно позволяет решить ту же задачу, что и `case`, но код при этом получается короче, элегантнее и понятнее.

Задача 3.3 *Напишите функцию, которая предлагает пользователю ввести своё имя и в случае, если это ваше имя (учитывая сложные отношения Эрланга с Unicode, используйте транслитерацию) выводит «Hello, author!», а в противном случае выводит «Hello, Username», где Username — введённое пользователем имя.*

Задача 3.4 *Определите функцию `sign`, принимающую число и возвращающую -1 для всех отрицательных чисел, 0 для 0 и 1 для всех положительных чисел.*

3.3. Гварды

Гварды (охранные выражения) работают очень просто: код, сопровождаемый гвардом, будет выполнен только в том случае, если охранное выражение истинно (то есть возвращает `true`). Они нам, на самом деле, уже встречались: оператор `if`, описанный на странице 32, представляет собой как раз последовательность гвардов, каждый из которых «охраняет» определённое действие.

Но чаще всего гварды применяются в функциях, определённых через сопоставление с образцом. Использование гвардов позволяет практически не использовать выражения `if` и `case`, что упрощает чтение кода. Функция с гвардом определяется через ключевое слово `when`:

```
-module(max).
-export([max/2]).

max(X,  Y) when X > Y -> X;
max(_X, Y) -> Y.
```

В этом примере первое сопоставление с образцом будет успешным только если гвард «`X > Y`» окажется истинным, иначе Эрланг-машина перейдёт к следующему образцу.

Обратите внимание, что во втором варианте функции анонимная переменная записана как «`_X`». Для компилятора такая запись ничем не отличается от «`_`», но программисту так проще понять, на месте какого параметра в этом варианте стоит анонимная переменная.

По соображениям производительности внутри гвардов запрещено использовать функции, кроме небольшого числа встроенных функций Эрланга (обратите внимание — гвардов внутри `if` это тоже касается).

Помимо заголовков функций и оператора `if`, гварды также можно использовать внутри оператора `case`.

Задача 3.5 *Реализуйте функцию, получающую числа N и M и определяющую, является ли M делителем N.*

3.4. Рекурсия

Рекурсивной (как, наверное, большинство читателей знает) называется функция, вызывающая сама себя. Даже некоторые хорошо знакомые с программированием люди считают рекурсию чем-то вроде экзотического программистского трюка. На самом же деле, особенно в функциональных языках, рекурсия — простой и естественный способ решения большинства задач. Очень многие алгоритмы проще и понятнее всего записываются в виде рекурсивных функций. Правда, самая простая форма записи — не всегда самая эффективная. Часто рекурсивные алгоритмы на сколько-нибудь серьёзных объёмах входных данных потребляют неприлично большое количество памяти. Однако и с этим можно бороться (например, методами, о которых будет рассказано в разделе 4.1).

Про рекурсивные функции принято рассказывать на примере рекурсивного вычисления степени или факториала. Не будем нарушать эту почтенную традицию.

```
-module(power).
-export([power/2]).

power(_, 0) -> 1;
power(N, M) -> N * power(N, M-1).
```

Запись функции `power` очень проста — по сути, это индуктивное определение операции возведения в степень. «Нулевая степень любого числа это 1; любая другая степень m числа n это n в степени $m-1$, умноженное на n». А понять, как это работает, можно вот каким образом.

Будем подставлять на место рекурсивного вызова результат его исполнения. Предположим, нам надо вычислить пятую степень числа 2. «Раскроем» рекурсивные вызовы `power`, то есть на месте каждого из них запишем те операции, которые он совершит.

```
power(2, 5) =
2 * power(2, 4) =
2 * (2 * power(2, 3)) =
2 * (2 * (2 * power(2, 2))) =
2 * (2 * (2 * (2 * power(2, 1)))) =
2 * (2 * (2 * (2 * (2 * power(2, 0))))) =
2 * (2 * (2 * (2 * (2 * 1)))) =
2 * (2 * (2 * (2 * 2))) =
2 * (2 * (2 * 4)) =
2 * (2 * 8) =
```

```
2 * 16 =
32
```

Ещё более естественной оказывается запись функции, вычисляющей факториал:

```
-module(factorial).
-export([factorial/1]).

factorial(0) -> 1;
factorial(N) -> N * factorial(N-1).
```

Сравните это с формулой вычисления факториала:

$$n! = \begin{cases} 1, & n = 0, \\ n \cdot (n-1)!, & n > 0. \end{cases}$$

Вычисляя эту функцию ото всех аргументов, за исключением 0, Эрланг будет вызывать её же с аргументом, на единицу меньшим исходного, пока не дойдёт до вызова `factorial(0)`. Так как определение `factorial(0)` не содержит рекурсивных вызовов, интерпретатор пойдёт от него назад по цепочке, подставляя результат вычисления следующей функции в предыдущую, пока не дойдёт до исходного вызова.

Очевидно, что под каждый новый вызов рекурсивной функции Эрланг выделяет новую память для всех используемых в функции переменных — иначе функция просто не смогла бы работать.

Уже видно, что очень многие рекурсивные функции устроены похоже. В них есть описание базового случая, очень простого, и описание того, как свести любой другой случай к базовому. Полезно думать об этом при написании новой рекурсивной функции — сначала понять, как она должна вести себя в базовом случае (для натуральных чисел это часто 1 или 0, для списков — пустой список, для деревьев — лист и т.д.), а потом понять, как свести к базовому любой другой случай.

Записывать их нужно именно в этом порядке — сначала базовый случай, потом общий. Помните, Эрланг сопоставляет с образцом варианты функций в порядке их записи, так что если более общий случай будет записан раньше, до остальных дело просто не дойдёт.

При определении функций через сопоставление с образцом сначала записываются частные случаи, потом общий.

Ту же идею можно применять при отладке программ: если что-то идёт не так, полезно задать себе вопросы: «Описан ли базовый случай?», «Приближает ли нас каждое выполнение функции для общего случая к базовому?», «В том ли порядке расположены образцы аргументов?».

Рекурсией при программировании на Эрланге нам придётся пользоваться часто: дело в том, что алгоритмы, в которых есть повторяющиеся действия, в функциональных языках описываются обычно именно с помощью рекурсивных функций. В Эрланге нет операторов, похожих на `for` и `while` Си, Паскаля или Питона, потому что нет операции присваивания (изменения значения переменной) — а без неё это были бы практически бессмысленные конструкции.

При обучении программированию принято использовать как пример цикла программу, выводящую на экран текст американской народной песни «99 Bottles of Beer» («99 бутылок пива»)[1]:

> 99 bottles of beer on the wall, 99 bottles of beer.
> Take one down and pass it around, 98 bottles of beer on the wall.
>
> 98 bottles of beer on the wall, 98 bottles of beer.
> Take one down and pass it around, 97 bottles of beer on the wall.
>
> ...
>
> 2 bottles of beer on the wall, 2 bottles of beer.
> Take one down and pass it around, 1 bottle of beer on the wall.
>
> 1 bottle of beer on the wall, 1 bottle of beer.
> Take one down and pass it around, no more bottles of beer on the wall.
>
> No more bottles of beer on the wall, no more bottles of beer.
> Go to the store and buy some more, 99 bottles of beer on the wall.

Вот как будет выглядеть такая программа на Эрланге:

```
-module(bottles).
-export([bottles/0]).

bottles(0) ->
    io:format("No more bottles of beer on the wall, "),
    io:format("no more bottles of beer.~n"),
    io:format("Go to the store and buy some more, "),
    io:format("99 bottles of beer on the wall.~n");
bottles(1) ->
    io:format("1 bottle of beer on the wall, "),
    io:format("1 bottle of beer.~n"),
    io:format("Take one down and pass it around, "),
    io:format("no more bottles of beer on the wall.~n"),
    bottles(0);
bottles(N) ->
    NewN = N - 1,
```

[1] На сайте `http://www.99-bottles-of-beer.net/` можно найти примеры такой программы на очень многих языках программирования (на момент написания книги их было полторы тысячи).

```
    io:format("~w␣bottles␣of␣beer␣on␣the␣wall,␣"),
    io:format("~w␣bottles␣of␣beer.~n", [N, N]),
    io:format("Take␣one␣down␣and␣pass␣it␣around,␣"),
    io:format("~w␣bottles␣of␣beer␣on␣the␣wall.~n~n", [NewN]),
    bottles(NewN).

bottles() ->
    bottles(99).
```

Видно, что для нашей конкретной задачи случаи для 1 и 0 можно было бы и объединить, но в реальной жизни такими «оптимизациями» лучше не заниматься, так что не будем это делать и в нашем игрушечном примере.

Обратите внимание, что в этом примере две функции с именем `bottles` и разной арностью: экспортируется функция `bottles/0`, но основная работа делается в `bottles/1`. Так часто делают в программах на Эрланге.

Задача 3.6 *Пользователь вводит с клавиатуры числа; ввод заканчивается нулём. Написать программу, подсчитывающую сумму введёных чисел.*

Задача 3.7 *Написать функцию, которая принимает положительное целое число и выводит на экран список делителей этого числа в порядке возрастания [Златопольский 2007, 10.4].*

Задача 3.8 *Напишите функцию, вычисляющую остаток от деления (аналог встроенного оператора `rem`), используя только операции вычитания и сравнения двух чисел (предполагается, что аргумент — натуральное число) [Miller].*

Задача 3.9 **Пользователь вводит с клавиатуры числа; ввод заканчивается нулём. Составить программу, выводящую на экран все введённые пользователем числа (их количество заранее неизвестно), не пользуясь списками, массивами и подобными им структурами данных. Порядок чисел не важен [Златопольский 2007, 10.3].*

3.5. Игры со списками

На странице 17 уже отмечалось, что определение списка рекурсивно: хвост непустого списка сам представляет собой список. Можно предположить, что такие структуры данных удобно обрабатывать с помощью рекурсивных алгоритмов. Это действительно так. Огромное количество функций в функциональных языках состоит из одних и тех же действий:

- проверить, не пуст ли переданный в качестве аргумента список; если список пуст, закончить работу;
- если список не пуст, взять его голову и сделать с ней какое-то действие;
- вызвать себя же для хвоста списка.[2]

Например, написанная по этой схеме функция вывода на экран элементов списка будет выглядеть так:

```
print_list([]) —>
    io:format(''~n'');
print_list([Head|Tail]) —>
    io:format(''~w '', [Head]),
    print_list(Tail).
```

Первый вариант функции разбирается со случаем, когда аргумент — пустой список. Это вариант просто выводит символ перевода строки. Во втором варианте ещё в объявлении функции список разбивается на голову (переменая `Head`) и хвост (переменная `Tail`). Голова выводится на экран, а хвост передаётся дальше: рано или поздно в хвосте останется только пустой список, он успешно сопоставится с первым образцом и выполнение функции закончится.

Выражение `[H|T] = L` разберёт список `L` на голову `H` и хвост `T`, выражение `L = [H|T]` создаст список `L` с головой `H` и хвостом `L`.

Ещё раз обратите внимание на то, в каком порядке записываются варианты функции: сначала базовый вариант, потом вариант, содержащий рекурсивный вызов.

А вот так будет выглядеть функция, вычисляющая сумму всех элементов списка чисел:

```
summ([]) —>
    0;
summ([Head|Tail]) —>
    Head + summ(Tail).
```

Идея здесь та же самая: первый вариант функции обрабатывает пустой список, второй — разделяет список на голову и хвост, обрабатывает голову и передаёт хвост дальше.

Как уже говорилось в разделе 2.5, составляется список с помощью всё того же оператора «|», поэтому типичная функция, формирующая список, обычно содержит операцию составления списка из текущего элемента и рекурсивного вызова. Вот как, например, будет выглядеть функция,

[2]На самом деле, схема эта встречается настолько часто, что есть способы не расписывать её вручную; про них будет рассказываться в разделе 4.3.

считывающая с клавиатуры числа одно за другим (ввод заканчивается нулём) и затем выводящая их список на экран:

```
-module(inputlist).
-export([numbers/0]).

numbers() ->
    {ok, [Number]} = io:fread('>␣', '~d'),
    case Number of
        0 -> [];
        _ -> [Number|numbers()]
    end.
```

Задача 3.10 *Определите функцию, принимающую в качестве аргументов список и значение и возвращающую список, из которого удалены все элементы, равные переданному значению. Т.е. приняв список `[1, 2, 3, 4, 5]` и значение `2`, эта функция должна возвратить список `[1,3,4,5]`, приняв список `['a', 'b', 'c', 'a', 'd', 'a']` и значение 'a', возвратить список `[b,c,d]` и так далее.[Friedman 1996, 33]*

Задача 3.11 *Определите функцию, разворачивающую список: получив в качестве аргумента список `[1, 2, 3]`, функция должна возвратить список `[3, 2, 1]`, получив список `['a', 'b', 'c']`, должна возвратить список `['c', 'b', 'a']` и т.д.*

Задача 3.12 *Напишите функцию, извлекающую из списка фрагмент: она должна принимать список, номер начального элемента фрагмента и номер его конечного элемента, а возвращать вырезанный фрагмент.*

Задача 3.13 *∗Определите функцию, которая превращает список, содержащий вложенные списки, в обычный «плоский» список. Например, получив список `[[1, 2, 3], 4, 5]`, она должна вернуть `[1,2,3,4,5]`, получив список `[red, [[orange, yellow], green], [[[blue], indigo], violet]]`, должна вернуть список `[red,orange,yellow,green,blue,indigo,violet]` и т.д. [99 problems, 07]*

Разумеется, для типовых задач вроде удаления элемента, разворачивания или сортировки в Эрланге существуют встроенные функции. Но было бы глупо отказываться от хороших простых упражнений, позволяющих почувствовать дух Эрланга и ФП вообще, из-за того, что выполняющий похожие задачи код поставляется вместе с языком.

Самые типовые задачи по обработке списков решают функции из модуля `lists`. Обсуждение части этих функций мы отложим до главы 4, без материала которой будет трудно объяснить, что они делают. Но вот про некоторые есть смысл рассказать прямо сейчас.

Функция `lists:member` проверяет, содержится ли в списке заданный элемент:

```
113> lists:member(1, [1, 2, 3]).
true

114> lists:member(100500, [1, 2, 3]).
false
```

Функция `lists:reverse` разворачивает список:

```
115> lists:reverse([1, 2, 3]).
[3,2,1]
```

Функция `lists:sum` подсчитывает сумму списка чисел:

```
116> lists:sum([1, 2, 3]).
6
```

Функция `lists:seq/2` возвращает идущие подряд числа от первого своего аргумента до второго:

```
117> lists:seq(1, 10).
[1,2,3,4,5,6,7,8,9,10]
```

Функция `lists:seq/3` возвращает идущие подряд числа от первого своего аргумента до второго с шагом, равным третьему аргументу:

```
118> lists:seq(1, 10, 2).
[1,3,5,7,9]
```

Функция `lists:sort` сортирует список:

```
119> lists:sort([3, 1, 2]).
[1,2,3]
```

Задача 3.14 *Проверить, является ли список палиндромом (то есть читается ли он одинаково слева направо и справа налево) [99 problems, 06]. Эту задачу можно (и нужно) решить без использования операторов ветвления (`if` и `case`).*

Задача 3.15 *Совершенным называется число, которое равно сумме всех своих делителей, включая 1, но исключая само число; например, 6 — совершенное число, так как $6 = 1 + 2 + 3$. Определите функцию `is_perfect`, проверяющую, является ли её аргумент совершенным числом [Харпер 1996, 2.5.6].*

3.6. Стек

Напомним, что стек — структура данных, работающая по принципу «последний зашёл — первый вышел» (иногда его ещё обозначают аббревиатурой LIFO, «Last-In-First-Out»). По аналогичному принципу работают, например, автоматный рожок или стержни с дисками в головоломке «Ханойские башни». Стеки используются во множестве алгоритмов, но особенно часто нужда в них возникает в задачах, связанных с синтаксическим разбором.

Традиционно стек описывают как тип данных, поддерживающий три операции: положить элемент на вершину стека (`push`), снять элемент с вершины стека (`pop`) и узнать, что за элемент лежит на вершине стека, не снимая его (`top`). Очевидно, что в Эрланге уже есть структура данных, поддерживающая эти операции: это хорошо знакомый нам список. Добавление к списку новой головы с помощью оператора «|» это `push`, а разделение списка на голову и хвост с помощью сопоставления с образцом даёт нам операции `pop` и `top`.

Разберём работу со стеком на примере функции, вычисляющей арифметическое выражение в обратной польской записи:

```
-module(polish).
-export([calculate/1]).

calculate([], [Top]) -> Top;
calculate(["+"|Rest], [Op1|[Op2|Stack]]) ->
    calculate(Rest, [Op1 + Op2|Stack]);
calculate(["-"|Rest], [Op1|[Op2|Stack]]) ->
    calculate(Rest, [Op2 - Op1|Stack]);
calculate(["*"|Rest], [Op1|[Op2|Stack]]) ->
    calculate(Rest, [Op1 * Op2|Stack]);
calculate(["/"|Rest], [Op2|[Op1|Stack]]) ->
    calculate(Rest, [Op1 / Op2|Stack]);
calculate([Number|Rest], Stack) ->
    calculate(Rest, [Number|Stack]).

calculate(L) -> calculate(L, []).
```

Вряд ли этот пример нужно подробно комментировать — функция является точным изложением сути алгоритма работы стекового калькулятора: если текущий элемент входной последовательности — знак операции, то со стека снимаются её аргументы, с ними выполняется эта операция, и результат кладётся обратно на стек. Все прочие символы считаются числами и сразу кладутся на стек. Когда входная после-

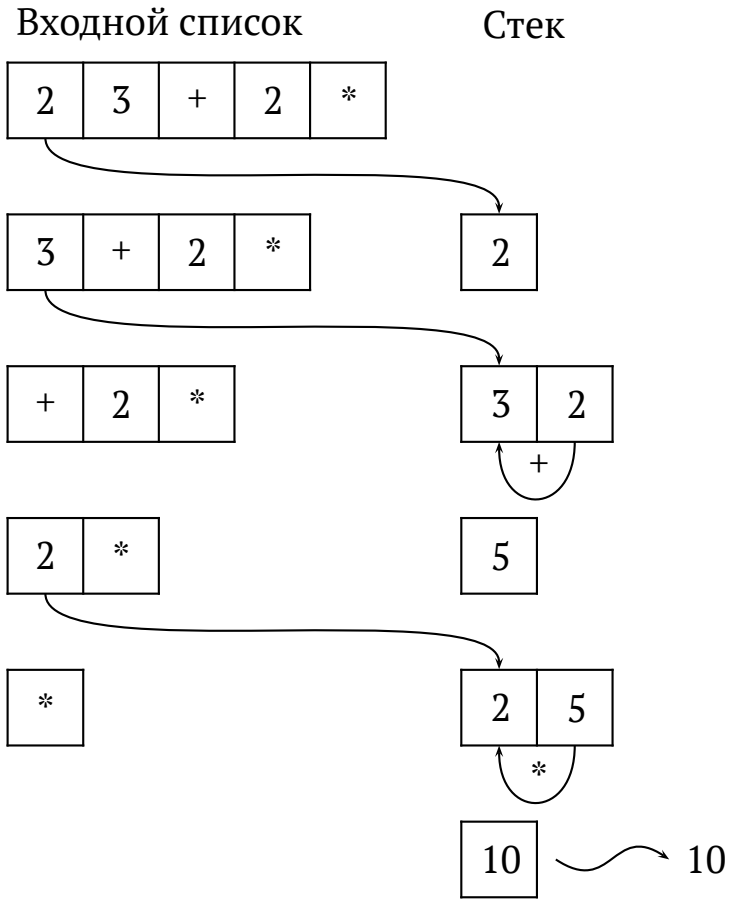


Рис. 3.1.: Функция `polish:calculate/2` вычисляет выражение «2 3 + 2 *», соответствующее выражению «$(2+3)*2$» в обычной инфиксной записи. Слева — входной список (первый аргумент функции), справа — стек (второй аргумент функции).

довательность заканчивается, в стеке должно остаться единственное значение — результат вычислений[3]; он и возвращается функцией (см. рис 3.1).

Обратите внимание, как хорошо подходит для такого рода задач определение функций через сопоставление с образцом — код получается понятный и при этом довольно короткий.

Задача 3.16 *Дана строка, содержащая три вида скобок: круглые («(», «)»), квадратные («[», «]») и угловые («<», «>»). Назовём правильным порядком скобок такой, при котором:*

- *закрывающая скобка встречается только после открывающей скобки того же типа;*
- *нет ни одной открывающей скобки, которой не соответствовала бы парная ей закрывающая, и наоборот;*
- *скобки не перекрываются (все вложенные скобки закрываются раньше, чем охватывающие).*

Примеры правильного порядка скобок:

```
[]()<>
[(<>)]
<([][][])([][][])([][][])>
[(<[(<[]>)(<[]>)(<[]>)]><[(<[]>)(<[]>)(<[]>)]>)]
```

Примеры неправильного порядка скобок:

```
>()
[(<)]
[(<>)])
<([)][][()>
```

Составить программу, которая проверяет правильность порядка скобок в строке.[Шень 2004, 6.1.1]

Задача 3.17 *∗Стек реализуется с помощью эрланговского списка легко — и добавление, и удаление элемента из головы списка требует одного действия. С очередью (структурой, устроеной по принципу FIFO — First In, First Out, «первый зашёл — первый вышел»), всё значительно сложнее.*

[3] Если длина стека по окончании вычислений больше 1, выполнение программы прекратится с ошибкой; это можно было бы предотвратить, изменив образец с `[Top]` на `[Top|_]`, но такое «улучшение» не даст нам ничего, кроме возможности вычислять значения бессмысленных выражений вроде $3\ 2 + 2 * 1$. Поэтому будем придерживаться принципа «Let it fail».

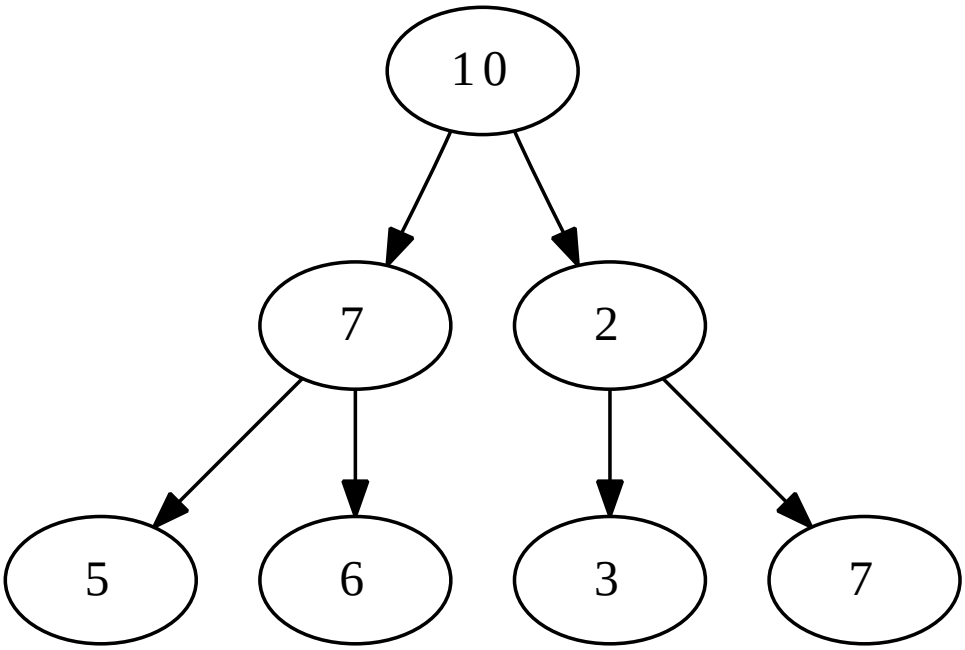


Рис. 3.2.: Двоичное дерево целых чисел.

«Тупая» реализация очереди с помощью списка (новые значения помещаем в голову списка, старые забираем из хвоста) потребует N действий для извлечения каждого из элементов, ведь для доступа к последнему элементу списка надо пройти весь список целиком. Следовательно, если через такую очередь будет пропущено N элементов, это потребует N^2 действий.

Придумайте алгоритм, позволяющий пропустить через очередь N элементов, затратив на это порядка N действий (количество действий должно линейно зависеть от количества обработанных значений). [Шень 2004, 6.2.2]

3.7. Деревья

Деревья в Эрланге, в принципе, можно хранить как вложенные списки. Но более традиционный метод — определить узел дерева как запись. Например, узел двоичного дерева можно определить так:

```
-record(node, {value, left=nil, right=nil}).
```

Очевидно, что `value` здесь — значение узла дерева, а `left` и `right` — значения дочерних узлов. Если дочерние узлы не заданы явно, им будет присвоено значение `nil`. На самом деле, в Эрланге есть специальный атом `undefined` («не определено»), и при отсутствии явного значения в поля `left` и `right` был бы подставлен именно он. Но это длинное слово, которое затруднило бы просмотр структур данных на экране или на

страницах этой книжки.

Таким образом, дерево с рисунка 3.2 в Эрланге будет представлено вот как:

```
#node{value = 10,
      left = #node{value = 7,
                   left = #node{value = 5,left = nil,right = nil},
                  right = #node{value = 6,left = nil,right = nil}},
      right = #node{value = 2,
                    left = #node{value = 3,left = nil,right = nil},
                   right = #node{value = 7,left = nil,right = nil}}}
```

Определим функцию для порождения случайного дерева, она пригодится нам в дальнейшем.

```
-module(trees).
-export([generate/1]).

-record(node, {value, left=nil, right=nil}).

generate(0) -> nil;
generate(L) ->
    #node{
        value=random:uniform(10),
        left=generate(L-1),
        right=generate(L-1)
    }.
```

Функция `uniform/1` из модуля `random` выдаёт случайное целое число в интервале от 1 до своего аргумента (есть ещё функция `random:uniform/0`, возвращающая случайное вещественное число в интервале от 0 до 1). Сама же функция `generate/1` устроена крайне просто. Её аргумент — это количество уровней в дереве, которое необходимо создать. Если количество уровней равно нулю, `generate/1` возвращает `nil`. Если же нет, то она создаёт узел дерева (структуру типа `node`), в дочерних узлах которого размещает результаты вызова `generate/1` для меньшего на 1 количества уровней:

```
3> trees:generate(0).
nil
4> trees:generate(1).
#node{value = 1,left = nil,right = nil}
5> trees:generate(2).
#node{value = 5,
```

```
      left = #node{value = 8,left = nil,right = nil},
      right = #node{value = 10,left = nil,right = nil}}
```

...и т.д.

Обход дерева выполняется так же просто. Вот, например, функция, которая выводит на экран значения вершин дерева:

```
print_tree(nil) -> io:format(''~n'');
print_tree(Node) ->
    print_tree(Node#node.left),
    io:format(''~w'', [Node#node.value]),
    print_tree(Node#node.right).
```

Задача 3.18 *Написать функцию, которая принимает дерево и возвращает его частичную копию, из которой с заданной вероятностью удалялись случайные поддеревья.*

Задача 3.19 *Написать функцию, которая по заданному n считает число всех вершин высоты n в заданном дереве [Шень 2004, 7.2.5]. Для тестирования используйте деревья, пропущенные через функцию из задачи 3.18.*

4. Полезные приёмы

> Радостное возбуждение, сопутствующее компьютерному программированию, происходит из постоянного раскрытия в голове и в компьютере все новых выраженных в виде программ механизмов и из взрыва восприятия, который они порождают.
>
> А.Дж.Перлис

4.1. Хвостовая рекурсия

В разделе 3.4 упоминалось, что циклов в Эрланге нет, и вместо них используется рекурсия. Читатель, понимающий, что именно происходит в памяти компьютера при рекурсивном вызове функции в таких языках, как Си или Паскаль, должен был в этом месте очень сильно удивиться. Действительно, в машинном коде, который порождают компиляторы этих языков, цикл может выполняться сколь угодно долго, а рекурсия довольно быстро приведёт к переполнению стека вызовов и падению программы. И даже если этого не произойдёт, рекурсивная реализация алгоритма будет потреблять значительно больше памяти, чем реализация, использующая цикл.

В функциональных языках, в том числе в Эрланге, эта проблема решена: при соблюдении определённых условий рекурсия может быть очень глубокой, даже бесконечной, и при этом программа будет использовать постоянный объём памяти.

Дело в том, что далеко не всякий рекурсивный алгоритм нуждается в результате рекурсивного вызова. Например, описанная на стр. 37 программа, выводящая текст песни «99 bottles of beer», никак не использует значения, которые возвращают рекурсивные вызовы функции `bottles/1`. Значит, нам совершенно незачем выделять новый стековый кадр для каждого вызова `bottles/1`.

Правда, функция `bottles/1` вообще не возвращает осмысленных значений. Но многие функции, такие значения возвращающие, тоже могут быть переписаны в том же духе. Немного переделаем функцию подсчёта суммы элементов списка, показанную на стр. 39:

```
summ([], Summ) -> Summ;
summ([H|T], Summ) -> summ(T, H+Summ).

summ(L) -> summ(L, 0).
```

В этом варианте функции `summ` тоже не используются значения рекурсивных вызовов — результат накапливается во втором аргументе `summ/2`. Иными словами, нам достаточно знать результат вычисления последнего вызова `summ/2`, а всё вычисленное в каждом из предыдущих вызовов функции Эрланг-машина может с чистой совестью забыть, как только следующему переданы аргументы.

Именно это она и делает. Рекурсивные функции, в которых не нужно возвращаться к предыдущим вызовам, компилятор Эрланга превращает в код, выполняющий, по сути, обычные циклы. Рекурсия, позволяющая эту оптимизацию, по-русски называется хвостовой рекурсией[1]

Часто бывает нужно превратить обычную рекурсивную функцию в функцию с хвостовой рекурсией, чтобы ускорить программу и заставить её использовать меньше памяти. Иногда без хвостовой рекурсии вообще не обойтись (например, в ситуациях, когда нам нужен бесконечный цикл, как во многих серверных приложениях). Но не надо заранее увлекаться излишней оптимизацией. Правильная последовательность действий — сначала добиться стабильной работы программы, и только потом улучшать её производительность.

Задача 4.1 *Найдите среди решений предыдущих задач те, которые уже содержат хвостовую рекурсию.*

Задача 4.2 *Какие из решений предыдущих задач можно переписать так, чтобы они содержали хвостовую рекурсию? Попробуйте это сделать.*

Задача 4.3 *Часто (хотя и не всегда) функции с хвостовой рекурсией не только экономнее по потреблению памяти, но и быстрее своих аналогов с обычной рекурсией. Скомпилируйте оба варианта кода, подсчитывающего сумму элементов списка чисел. Измерьте время их выполнения с помощью вызова функции* `timer:tc(modname, funcname, [lists:seq(1, 1000000)])`., *где*

[1] Этот термин воспроизводит английское tail recursion; нехвостовую рекурсию по-английски называют body recursion или — по-моему, не слишком удачно, учитывая смысл понятия — forward recursion; по-русски аналогичного термина, насколько мне известно, нет.

`modname` — имя модуля, в котором определена функция, а `funcname` — имя самой функции. Этот вызов вернёт кортеж, первый элемент которого — время исполнения переданной функции в микросекундах. Что быстрее?

4.2. Ссылки на функции, анонимные функции

Про функциональные языки говорят, что функции в них являются «полноправными гражданами», first class citizens. Это значит, что с функцией можно делать те же вещи, что и со строковыми и числовыми значениями: присваивать переменной, создавать во время выполнения программы, передавать другой функции в качестве аргумента или возвращать из неё.

В Эрланге для того, чтобы выполнять действия такого рода, используется оператор `fun`. Воспользуемся им, чтобы поместить в переменную функцию `summ` со страницы 39 (я предполагаю, что она находится в модуле `listsumm`, доступном интерпретатору Эрланга):

```
120> F = fun listsumm:summ/1.
#Fun<listsumm.summ.1>

121> F([1, 2, 3, 4, 5]).
15
```

Но оператор `fun` можно применять не только для получения ссылок на уже существующие функции, но и для создания новых. Помните функцию `square` со страницы 28? С помощью `fun` можно определить такую же функцию, не имеющую имени, и поместить её в переменную:

```
122> S = fun(X) -> X * X end.
#Fun<erl_eval.6.82930912>

123> S(2).
4

124> S(3).
9
```

В анонимных функциях, как и в обычных, можно определять несколько вариантов тела функции, выбор между которыми будет определяться через сопоставление с образцом. Вот, например, функция, делающая ту же работу, что и `safe_division` со страницы 31:

```
125> SD = fun(_, 0) -> undefined;
125>          (N, M) -> N / M
125>       end.
#Fun<erl_eval.12.82930912>

126> SD(10,0).
undefined

127> SD(10,2).
5.0
```

Задача 4.4 *∗Вы не можете вызвать анонимную функцию рекурсивно по имени переменной, в которую её помещаете. Никакого специального синтаксиса для рекурсивных вызовов в анонимных функциях в Эрланге тоже нет. Тем не менее, описывать рекурсивные алгоритмы с помощью анонимных функций можно. Как?*

4.3. Функции высшего порядка

Функции, возвращающие другие функции или принимающие их в качестве аргументов, называют функциями высшего порядка. Функции высшего порядка позволяют избежать повторяющегося кода благодаря тому, что многие типовые процедуры, в которых от программы к программе различается только конкретное выполняемое действие, выносятся в стандартную библиотеку.

Например, функция `lists:map/2` принимает функцию и список, применяет функцию к каждому элементу списка и возвращает список результатов:

```
128> L = [1, 2, 3].
[1,2,3]

129> lists:map(fun(X) -> X+2 end, L).
[3,4,5]
```

Функция `lists:filter/2` принимает функцию и список и возвращает только те элементы списка, для которых функция-аргумент вернула `true`:

```
130> L = [1, 2, 3, 4, 5].
[1,2,3,4,5]
```

```
131> lists:filter(fun(X) -> X rem 2 =:= 1 end, L).
[1,3,5]
```

В разделе 3.5 рассказывалось о функции `lists:sort/1`, сортирующей список. Но в модуле `lists` есть ещё одна функция сортировки, более интересная — `lists:sort/2`. Функция `lists:sort/2` принимает список и функцию сравнения, устроенную следующим образом: если первый аргумент функции сравнения меньше или равен второму, она возвращает `true`, иначе — `false`. Вот как, например, будет выглядеть вызов `lists:sort/2`, упорядочивающий список по возрастанию модулей целых чисел:

```
132> L=[10, -2, 5, -100500, -1].
[10,-2,5,-100500,-1]

133> lists:sort(fun(A,B) -> abs(A) =< abs(B) end, L).
[-1,-2,5,10,-100500]
```

Задача 4.5 *Пользуясь функцией `is_perfect` из задачи 3.15, найдите список совершенных чисел в диапазоне от 2 до 1000.*

Задача 4.6 *Реализуйте функцию `maptree`, принимающую функцию F и дерево и возвращающую дерево, где на месте каждого значения X расположен результат операции F(X).*

Две функции, о которых надо рассказать особо — это функции *свёртки списка*, `foldr` и `foldl`. Свёртка списка — это преобразование списка в одиночное значение с помощью применения функции к каждому элементу списка. Например, функция `summ` со страницы 39 выполняет свёртку: она начинает с нуля и (если список не пуст) прибавляет к нему первый элемент списка, потом второй и т.д.:

```
summ([1, 2, 3]) =
1 + summ([2, 3]) =
1 + (2 + summ([3])) =
1 + (2 + (3 + summ[])) =
1 + (2 + (3 + 0)) =
1 + (2 + 3) =
1 + 5 =
6
```

Поглядим на вариант функции `summ`, использующий хвостовую рекурсию (мы уже видели его на стр. 49):

```
summ([], Summ) -> Summ;
summ([H|T], Summ) -> summ(T, H+Summ).

summ(L) -> summ(L, 0).
```

Она, естественно, тоже выполняет свёртку, хотя процесс вычислений выглядит чуть иначе:

```
summ([1, 2, 3]) =
summ([1, 2, 3], 0) =
summ([2, 3], 1 + 0) =
summ([3], 2 + 1) =
summ([], 3 + 3) =
6
```

То, что делает первая версия функции `summ`, называется *правой свёрткой*. Вторая, хвосторекурсивная версия функции `summ` выполняет *левую свёртку* списка. Пошагово расписанный процесс вычислений делает заметным преимущество левой свёртки: она использует постоянный объём памяти.

Очевидно, что результаты правой и левой свёртки будут одинаковыми только для ассоциативных и коммутативных операций. Но если вы используете именно такие операции, рекомендуют использовать левую свёртку.

Обычно `foldl` эффективнее, чем `foldr`

Перейдём к примерам. Как нетрудно догадаться, функция `lists:foldr/3` выполняет правую свёртку, а `lists:foldl/3` левую.

Вот функция, аналогичная эрланговской `lists:max/1`, вычисляющая максимальный элемент списка чисел:

```
listmax([H|T]) -> lists:foldl(fun max/2, H, T).
```

(здесь мы используем встроенную функцию Эрланга `max/2`, возвращающую больший из своих аргументов).

А вот функция, вычисляющая факториал:

```
fac(N) ->
    lists:foldl(fun(A,B) -> A*B end, 1, lists:seq(1, N)).
```

И умножение, и определение максимума коммутативны и ассоциативны, поэтому, разумеется, по смыслу в этих функциях подошли бы как правая, так и левая свёртки.

А вот, например, операция вычисления остатка от деления некоммутативна, поэтому правая и левая свёртки дадут разные результаты:

```
134> lists:foldr(fun(A,B) -> A rem B end, 16, [9, 4, 3]).
0

135> lists:foldl(fun(A,B) -> A rem B end, 16, [9, 4, 3]).
3
```

Задача 4.7 *Подсчитайте длину списка с помощью любой из функций свёртки.*

Задача 4.8 *Определите функцию, разворачивающую список, с помощью левой свёртки. [Абельсон 2010, 2.39]*

Задача 4.9 *Вычисление среднего арифметического для списка `L` нетрудно определить как `lists:sum(L) / length(L)`. К сожалению, для определения длины списка его необходимо пройти целиком, т.е. при таком вычислении список проходится дважды. Как подсчитать среднее арифметическое за один проход? Для сокращения записи используйте свёртку [Бешенов 2010].*

Задача 4.10 *В библиотеке Data.List языка Haskell есть выполняющие свёртку функции foldr1 и foldl1, не принимающие начального значения: вместо него используется последний элемент списка в foldr1 и первый элемент списка в foldl1 (разумеется, передаваемый этим функциям список должен быть непустым). В эрланговском модуле `lists` таких функций нет. Реализуйте их самостоятельно, не используя `lists:foldr/3` и `lists:foldl/3`.*

Задача 4.11 *Во многих языках, не только функциональных, часто организуют циклы, применяя к списку, порождённому функцией вроде `lists:seq`, какое-то действие с помощью функций вроде эрланговских `lists:map/2` или `lists:foldr/lists:foldl`. Другое дело, что, как правило, в этих языках есть механизмы, позволяющие не размещать используемый таким образом список в памяти целиком: генераторы вроде `xrange` в Питоне, ленивые вычисления в Хаскеле и т.д.*

Функция `lists:seq` размещает список в памяти, и если нам нужна большая, но генерируемая по простым правилам последовательность, это может стать проблемой. Давайте отчасти решим эту проблему, определив функцию `foldseq/3`, принимающую начальное и конечное значения последовательности из идущих подряд целых чисел и функцию от двух аргументов F. Функция `foldseq/3` должна возвращать свёртку входной последовательности (начальным значением для свёртки будет выступать первый элемент), но порождаться целиком входная последовательность не должна: после вычисления очередного элемента списка-результата интерпретатор должен «забывать» её очередной элемент.

Выше мы обсуждали функции, принимающие другие функции как аргументы. Несколько реже применяемая, но тоже очень полезная разновидность функций высшего порядка — функции, возвращающие функции.

Определим функцию `addN`, принимающую число и возвращающую функцию, которая прибавляет к своему аргументу это число:

```
-module(add).
-export([addN/1]).

addN(N) ->
    fun(X) -> X + N end.
```

Вот как она работает:

```
136> c(add).
ok,add

137> Add1 = add:addN(1).
#Fun<add.0.129319818>

138> Add1(2).
3

139> Add1(10).
11

140> Add5 = add:addN(5).
#Fun<add.0.129319818>

141> Add5(2).
7

142> Add5(10).
15
```

Этот приём можно применять там, где у нас может возникнуть нужда в функции, поведение которой зависит от какого-то заранее неизвестного параметра. Предположим, мы пишем компьютерную игру, в которой определили для хранения информации об оружии запись следующего вида:

```
-record(weapon, {type, level=0, damage=1}).
```

В поле `type` мы будем хранить тип оружия, в поле `level` — уровень, необходимый персонажу для того, чтобы его использовать, а в поле `damage` — вред, наносимый оружием за один успешный удар.

Пусть в некоторой ситуации (например, герой зашёл в лавку оружейника или нашёл клад) нам нужно отфильтровать список доступного оружия, оставив только то, которое доступно герою с его текущим уровнем. Доступное оружие хранится в списке `Weapons`, примерно вот таком:

```
[#weapon{type = ''Short Sword'',level = 1,damage = 2},
 #weapon{type = ''Long Sword'',level = 3,damage = 10},
 #weapon{type = ''Great Saber'',level = 10,damage = 25}]
```

Функция должна принимать уровень героя и список оружия и работать примерно так:

```
143> rpg:availableWeapons(1, Weapons).
[#weapon{type = ''Short Sword'',level = 1,damage = 2}]

144> rpg:availableWeapons(3, Weapons).
[#weapontype = ''Short Sword'',level = 1,damage = 2,
#weapontype = ''Long Sword'',level = 3,damage = 10]

145> rpg:availableWeapons(10, Weapons).
[#weapontype = ''Short Sword'',level = 1,damage = 2,
#weapontype = ''Long Sword'',level = 3,damage = 10,
#weapontype = ''Great Saber'',level = 10,damage = 25]
```

Мы, конечно, можем определить функцию, фильтрующую список оружия для заданного уровня героя, с помощью анонимной функции:

```
availableWeapons(Level, Weapons) ->
    lists:filter(fun(X) -> X#weapon.level =< Level end,
                 Weapons).
```

Но такой код не очень хорошо читается (и чем сложнее будет проверка, тем уродливее он будет). Правильнее определить вспомогательную функцию `forLevelN`, которая будет возвращать функцию, определяющую, подходит ли оружие для определённого уровня:

```
forLevelN(N) -> fun(X) -> X#weapon.level =< N end.
```

Тогда функция `availableWeapons` становится гораздо изящнее:

```
availableWeapons(Level, Weapons) ->
    lists:filter(forLevelN(Level), Weapons).
```

Задача 4.12 *Определите функцию* `ismemberof`*, принимающую список и возвращающую функцию, проверяющую, является ли её аргумент элементом этого списка. Пример работы* `ismemberof`*:*

```
146> IsDigit = ismemberof:ismemberof(lists:seq(0, 9)).
#Fun<ismemberof.0.45327933>

147> IsDigit(0).
true

148> IsDigit(5).
true

149> IsDigit(10).
false

150> IsVowel = ismemberof:ismemberof([a, e, i, o, u, y]).
#Fun<ismemberof.0.45327933>

151> IsVowel(a).
true

152> IsVowel(k).
false

153> IsVowel(z).
false
```

5. Приятные мелочи

> Низкоуровневым является любой язык, требующий от программиста внимания к несущественному.
>
> А.Дж.Перлис

5.1. Списковые включения

В общем-то, изложенного в предыдущих разделах синтаксиса достаточно, чтобы решать почти любые задачи, не требующие параллельного программирования. Но мы помним, что хороший язык программирования позволяет не просто решать задачи, а решать их наглядно и элегантно. Надо стремиться к тому, чтобы программа была не только инструкцией по вычислению правильного ответа для машины, но и изложением сути решения для человека.

Достичь этой цели помогает *синтаксический сахар* — средства языка, которые не расширяют его возможности, но делают запись ваших решений более короткой и понятной. В этом разделе мы рассмотрим несколько возможностей Эрланга, относящихся к синтаксическому сахару, и начнём со списковых включений[1].

Списковые включения — это способ порождать списки из других списков с помощью конструкций, напоминающих теоретико-множественную запись. Например, множество

$\{x \mid x \in \{1, 100500, 2, -1000, 5\}, x < 3\}$

с помощью списковых включений может быть описано так:

```
[X || X <- [1, 100500, 2, -1000, 5], X < 3].
```

Разберём синтаксис списковых включений подробнее. Начнём со спискового включения, которое возвращает исходный список:

```
154> [X || X <- [1, 2, 3]].
[1,2,3]
```

[1] Так принято переводить на русский английское list comrehensions.

Можно сформулировать то, что делает это списковое включение, так: «для каждого X из списка [1, 2, 3] поместить X в итоговый список». Общее правило в том, что справа от символа || записывается выражение, описывающее элемент списка-результата, а слева — правило, по которому извлекаются значения из исходного списка.

Правило извлечения значений из исходного списка имеет такой вид: переменная, в которую помещаются значения, символ <—, выражение, возвращающее исходный список (его называют *генератором*) и (необязательно) условия, которые должны быть истинными (возвращать true), чтобы значение было обработано (они называются *фильтрами*):

```
155> [X || X <- lists:seq(1, 5)].
[1,2,3,4,5]

156> [X || X <- lists:seq(1, 5), X rem 2 =:= 1].
[1,3,5]
```

В первом из этих примеров списковое включение извлекает из исходного списка все его элементы, во втором — только те, которые проходят фильтр «остаток от деления элемента на 2 равен 1», то есть нечётные.

Выражение слева от || не обязательно должно содержать переменные. Вот так, например, мы можем получить список из восьми единиц:

```
157> [1 || _ <- lists:seq(1, 8)].
[1,1,1,1,1,1,1,1]
```

Смысл этой строчки можно выразить так: «для каждого элемента исходного списка поместить единицу в итоговый список». Так как значения исходного списка мы использовать не собираемся, для их обозначения используется анонимная переменная (_).

Но как правило, конечно, справа от || используют выражение, содержащее элемент исходного списка. Вот, например, как можно вычислить список квадратов чисел от 1 до 5:

```
158> [X*X || X <- lists:seq(1, 5)].
[1,4,9,16,25]
```

Задача 5.1 *Определение функции `is_perfect` из задачи 3.15 с помощью списковых включений можно сделать гораздо короче. Как?*

Задача 5.2 *Написать функцию, принимающую два списка и возвращающую список их общих элементов (элементов, которые есть и в одном списке, и в другом).*

Генераторов в списковом включении может быть больше одного. Вот, например, как выглядит функция, ищущая прямоугольные треугольники с целой длиной гипотенузы для заданного диапазона целых длин катетов:

```
-module(triangles).
-import(lists, [seq/2]).
-import(math, [sqrt/1]).
-export([int_triangles/2]).

i_hyp(A, B) ->
    sqrt(A * A + B * B) == trunc(sqrt(A * A + B * B)).

int_triangles(F, T) ->
    [{X, Y} || X <- seq(F, T), Y <- seq(F, T), i_hyp(X, Y)].
```

Функция `i_hyp` проверяет, получается ли у треугольника гипотенуза целой длины. Списковое включение в функции `int_triangles` отфильтровывает такие длины катетов, для которых `i_hyp` возвращает true.

Обратите внимание, что здесь мы сравниваем целые и вещественные числа, поэтому надо применять оператор «==» вместо обычного «=:=».

В этом примере используется директива компилятора `-import()`; она позволяет вызывать функции из других модулей только по имени, без явного упоминания модуля.

Задача 5.3 *Напишите программу, выводящую на экран таблицу умножения примерно в таком вот виде:*

```
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
...
9 * 5 = 45
9 * 6 = 54
9 * 7 = 63
9 * 8 = 72
9 * 9 = 81
```

Задача 5.4 *∗Решить задачу 5.2, не используя функцию lists:member или её самодельные аналоги.*

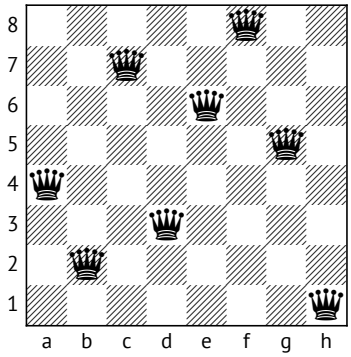


Рис. 5.1.: Одно из решений задачи о восьми ферзях.

Задача 5.5 ∗*«Задача о восьми ферзях» — классическая задача, сформулированная ещё в 1848 году, а в учебниках по программированию встречающаяся по крайней мере с шестидесятых годов XX века: как расставить на шахматной доске 8 ферзей так, чтобы они не били друг друга (т.е. не имели бы общих вертикалей, горизонталей и диагоналей; одно из решений можно видеть на рисунке 5.1).*

Определите функцию, возвращающую все решения задачи о восьми ферзях.

5.2. Операторы ++ и −−

Удобный, хотя и малопроизводительный инструмент — операторы работы со списками: конкатенация («++») и разность («−−»).

Конкатенация списков действует очень просто: она объединяет списки, переданные ей в качестве аргументов:

```
159> [1, 2, 3] ++ [4, 5].
[1,2,3,4,5]

160> ''Hello, '' ++ ''world!''.
''Hello, world!''
```

Оператор «−−» удаляет из своего левого аргумента элементы, встреченные в правом:

```
161> [1, 2, 3, 4, 5] -- [1, 2].
[3,4,5]

162> ''sword'' -- ''s''.
''word''
```

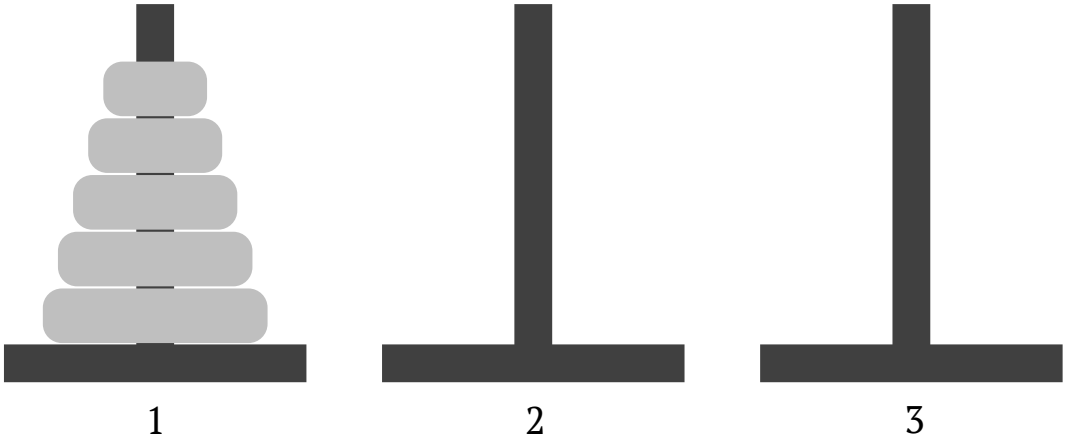


Рис. 5.2.: Ханойские башни

Важная особенность работы оператора «--» в том, что разность списков — это не разность множеств: если какой-то элемент встречается в левом списке несколько раз, для того, чтобы полностью удалить его из левого списка, нужно столько же раз упомянуть его в правом:

```
163> [1, 1, 1, 2, 2] -- [1, 1].
[1,2,2]

164> ''switches'' -- ''s''.
''witches''
```

Пользоваться операторами «++» и «--» нужно с осторожностью: «++» полностью пробегает свой левый аргумент, так что с длинными списками в левой части будет работать медленно; производительность «--» ещё хуже.

Задача 5.6 *Определите функцию, разворачивающую список с помощью правой свёртки (не путайте эту задачу с задачей 4.8) [Абельсон 2010, 2.39]. В решении должна использоваться функция* `foldr`.

Задача 5.7 *Написать функцию, превращающую дерево в список, сохраняя порядок следования вершин слева направо. Например, для дерева с рисунка 3.2 на стр. 45 такая функция должна выдавать список:* `[5, 7, 6, 10, 3, 2, 7]`.

Задача 5.8 *Вы, скорее всего, сталкивались со знаменитой головоломкой «Ханойские башни» (рис. 5.2), но на всякий случай напомню, как она устроена. Головоломка состоит из трёх стержней и некоторого количества*

дисков разного размера. Диски уложены друг на друга в порядке убывания диаметра (чем выше, тем меньше) на одном из стержней. Диски можно перемещать только со стержня со стержень и только по одному. Бóльший диск нельзя класть на меньший. Задача — переложить по этим правилам стопку дисков с одного стержня на другой.

Пронумеруем стержни слева направо и будем считать, что диски сложены на левом стержне. Напишите функцию, возвращающую решение головоломки для заданного количества дисков в виде списка кортежей вида `{From, To}`*, где* `From` *— номер стержня, с которого снимается диск, а* `To` *— номер стержня, на который диск перемещается: например, для случая одного диска функция должна возвращать* `[{1,3}]`*, для двух дисков —* `[{1,2},{1,3},{2,3}]` *и т.д.*

Задача 5.9 ∗*Функция* `dropwhile/2` *из модуля* `lists` *принимает функцию и список, а возвращает список, из которого выкинуты все элементы до первого, для которого функция-аргумент вернула* `false`*:*

```
165> lists:dropwhile(fun(X) -> X < 5 end, [1,2,3,5,7,3,5]).
[5,7,3,5]

166> lists:dropwhile(fun(X) -> X rem 2 =:= 1 end, [7,7,7,8,7,7,7]).
[8,7,7,7]
```

Реализуйте свою версию функции `dropwhile`*, используя свёртку списков: она, хотя и может содержать дополнительные определения, должна возвращать именно результат вызова функции свёртки.*

Обе ли функции свёртки подходят для реализации `dropwhile`*? Обоснуйте ответ.*

Задача 5.10 ∗*Написать функцию, которая возвращает список всех перестановок чисел 1..n [Шень 2004, 7.3.2].*

6. От функционального программирования к параллельному

> Не стоит изучать язык, который не меняет вашего представления о программировании.
>
> А.Дж.Перлис

6.1. Азы параллельного программирования

В этой книжке мы использовали Эрланг как простой в изучении и в то же время богатый функциональный язык. Но, как уже говорилось, известен Эрланг прежде всего своими возможностями по созданию параллельных (то есть состоящих из нескольких одновременно выполняющихся процессов) и распределённых (то есть выполняющихся на нескольких компьютерах) программ. Было бы глупо совсем не затронуть эти его возможности.

Программа на Эрланге может состоять из нескольких процессов. Этих процессов может быть очень много — сотни, тысячи и даже десятки тысяч. Общаются между собой эти процессы с помощью передачи сообщений. У каждого процесса есть «почтовый ящик», в который складываются сообщения от других процессов. В момент, когда процесс готов разобрать поступившие сообщения, он может выбрать из них нужные с помощью сопоставления с образцом, а остальные оставить в почтовом ящике или отбросить.

Для того, чтобы начать создавать параллельные программы, нам нужно узнать не так много. Любой процесс в Эрланге — это просто функция. Чтобы разбирать сообщения, она должна использовать конструкцию `receive`. Устроена `receive` похоже на оператор `case` или на определение функции: в ней перечисляются образцы, с которыми сопоставляется выражение, и, после символа «—>», действия, которые надо совершить, если сопоставление с образцом прошло успешно. Образцы проверяются в том

порядке, в котором записаны, сверху вниз. Если содержание сообщения нас не интересует, достаточно написать в качестве образца несвязанную переменную, с которой успешно сопоставится любое выражение.

Например, приведённая ниже функция, получив любое сообщение, выводит на экран строку, состоящую из текста «Received: » («Получено: ») и содержания сообщения:

```
-module(output).
-export([server/0]).

server() ->
    receive
        X -> io:format(''Received: ~s~n'', [X])
    end,
    server().
```

Обратите внимание на рекурсию в функции `server`. У неё нет никакого «базового случая», и она будет выполняться бесконечно, точнее, до тех пор, пока пользователь не остановит программу. Но это не ошибка: нам как раз и нужно, чтобы процесс, в котором запущена функция `server`, постоянно крутился в памяти, ожидая наших сообщений.

Если в момент исполнения конструкции `receive` почтовый ящик будет пуст, то процесс приостановится до получения первого сообщения.

Разумеется, рекурсия, используемая в таких случаях, должна быть хвостовой (если вы не очень твёрдо помните, что это такое, самое время перечитать раздел 4.1).

Теперь давайте запустим нашу функцию в отдельном процессе и попробуем передавать ей сообщения.

Новые процессы в Эрланге запускает встроенная функция `spawn`. Она принимает ссылку на функцию и возвращает так называемый *идентификатор процесса* — специальное значение, с помощью которого можно отправлять процессу сообщения:

```
167> Pid = spawn(fun output:server/0).
<0.38.0>
```

Традиционно используемое для хранения идентификатора процессора имя переменной `Pid` — это сокращение от английских слов «Process identificator», «идентификатор процесса».

Сообщения процессам посылаются с помощью оператора «`!`»:

```
168> Pid ! ''Hello, parallel world''.
Received: Hello, parallel world
''Hello, parallel world''
```

Строка «Received: Hello, parallel world» выведена сервером, а ''Hello, parallel world'' — значение, которое возвращает применение оператора «!». То, что оператор передачи сообщения возвращает содержание сообщения, позволяет отправить сообщение сразу нескольким процессам вот так:

```
Pid1 ! Pid2 ! Pid3 ! ''Hello''.
```

(в приведённом примере сообщение ''Hello'' отправляется процессам с идентификаторами, хранящимися в переменных Pid1, Pid2 и Pid3).

Итак, мы научились передавать сообщения. Но большинство сообщений полезно только в том случае, если мы получаем на них ответ. Как и в обмене почтовыми сообщениями между людьми, чтобы процесс-получатель мог нам ответить, он должен располагать «обратным адресом» — идентификатором процесса-отправителя.

Рассмотрим, как в Эрланге работают с отправкой ответа, на примере процесса, реализующего простейший калькулятор. Он принимает сообщения вида {Client, {Action, A, B}}, где Client — идентификатор процесса-отправителя, Action — знак одного из четырёх известных калькулятору арифметических действий ('+', '-', '/', '*'), а A и B — числа, с которыми надо совершить это действие:

```
calcserver() ->
    receive
        {Client, {'+', A, B}} ->
            Client ! {ok, A + B},
            calcserver();
        {Client, {'-', A, B}} ->
            Client ! {ok, A - B},
            calcserver();
        {Client, {'*', A, B}} ->
            Client ! {ok, A * B},
            calcserver();
        {Client, {'/', A, B}} ->
            Client ! {ok, A / B},
            calcserver();
        {Client, _} ->
            Client ! {error, ''Malformed message''},
            calcserver();
        {BadMessage} ->
            io:format(''Error: Malformed message \''~s\''~n'',
                      [BadMessage]),
```

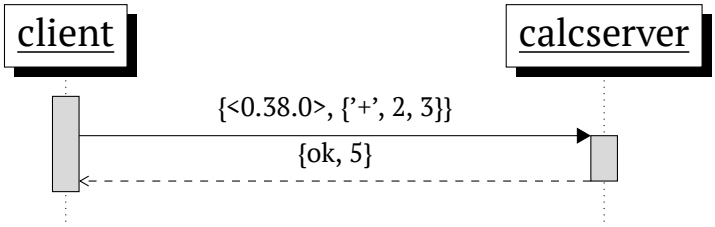


Рис. 6.1.: Обмен сообщениями между сервером-калькулятором и клиентом.

```
            calcserver()
    end.
```

Если от клиента получено «неизвестное» серверу (не заданное явно ни одним из образцов в конструкции `receive`) действие, клиенту отправляется ошибка `{error, ''Malformed message''}`. Если сообщение не содержало даже идентификатора процесса-клиента в оговорённом виде, информация об ошибке отправляется на стандартный вывод.

После отправки сообщения мы не собираемся выполнять никаких действий, кроме ожидания ответа, поэтому разумно объединить отправку сообщения и конструкцию `receive` в одну функцию:

```
call(Pid, Command) ->
    Pid ! {self(), Command},
    receive
        Answer -> Answer
    end.
```

Протестируем получившуюся связку из «клиента» и «сервера»:

```
169> c(calc).
{ok,calc}

170> Pid = spawn(fun calc:calcserver/0).
<0.38.0>

171> calc:call(Pid, {'+', 2, 3}).
{ok,5}

172> calc:call(Pid, {'-', 10, 5}).
{ok,5}

173> calc:call(Pid, {'*', 3, 5}).
{ok,15}
```

```
174> calc:call(Pid, {'/', 15, 4}).
{ok,3.75}

175> calc:call(Pid, {'%', 3, 2}).
{error,''Malformed message''}

176> Pid ! {''Not a message''}.
Error: Malformed message ''Not a message''
{''Not a message''}
```

Если очередь сообщений пуста, процесс останавливается и ждёт, когда в ней окажется хотя бы одно сообщение. Если это не то, чего мы хотим, мы можем задать время в миллисекундах, после которого процесс должен прервать ожидание и выполняться дальше. Это делается с помощью ключевого слова `after`. Например, описанную выше функцию отправки сообщения и получения ответа можно модифицировать вот так:

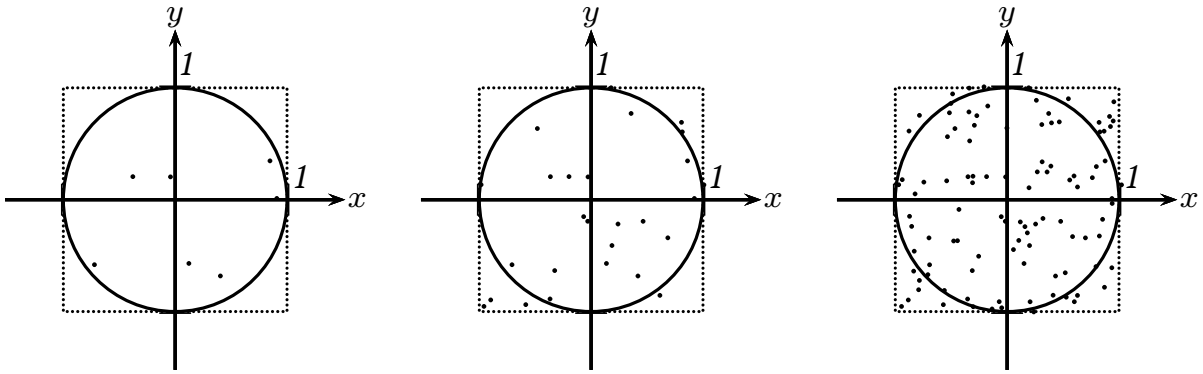
```
call(Pid, Command) ->
    Pid ! {self(), Command},
    receive
        Answer -> Answer
    after 2000 ->
        io:format(''Server␣error~n'')
    end.
```

Этот вариант функции, не получив от сервера ответа в течении двух секунд, выведет на экран сообщение «Server error».

Задача 6.1 *В практическом программировании зачастую быстро полученный приближённый вариант — это лучше, чем поздно полученный точный. Параллельное программирование позволяет легко обеспечить доступ к предварительным результатам вычисления, не прерывая подсчёт более точного значения.*

В качестве примера ресурсоёмкой задачи воспользуемся задачей вычисления числа π методом Монте-Карло[1]. Метод Монте-Карло для приблизительного расчёта площади двумерной фигуры можно реализовать, например, так: заключить фигуру в прямоугольник известной площади, вычислить некоторое количество точек со случайными координатами в пределах этого прямоугольника, а затем подсчитать, какой процент точек попадает внутрь фигуры. Этот процент будет примерно соответствовать проценту площади нашего прямоугольника, занятой фигурой, причём

[1] *Монте-Карло — это не фамилия автора метода, а название местности в княжестве Монако, в которой расположено множество казино; название подчёркивает роль случайности при расчёте.*



Внутри 7
Снаружи 0
$\pi \approx 4.0$

Внутри 22
Снаружи 5
$\pi \approx 3.259259259259259$

Внутри 80
Снаружи 23
$\pi \approx 3.1067961165048543$

Рис. 6.2.: Вычисление приближённого значения числа π методом Монте-Карло.

с ростом количества точек всё лучше (разумеется, асимптотически, из-за случайности вычисляемых координат).

Единичная окружность (то есть окружность с радиусом 1 и центром в начале координат) задаётся уравнением $x^2 + y^2 = 1$. Соответственно, для точки (X, Y), попадающей в очерченный этой окружностью круг, будет верным неравенство $X^2 + Y^2 \leqslant 1$. Этот круг вписан в квадрат с углами в координатах $(-1, -1)$, $(-1, 1)$, $(1, 1)$ и $(1, -1)$, площадью 4. Площадь самого круга равна π по формуле площади круга $S = \pi r^2$. Соответственно, отношение числа попавших внутрь круга точек к общему числу точек даст нам долю площади квадрата, занятой кругом, а умножение этого числа на площадь квадрата даст нам саму площадь круга, то есть как раз число π (см. рис. 6.2).

Реализуйте функцию, которая в отдельном процессе будет с помощью бесконечной рекурсии[2] вычислять случайные точки по описанным правилам. По сообщению извне вида `{Pid, get_current}`, где `Pid` — идентификатор пославшего сообщение процесса, она должна отправлять ответ вида `{Inside, Outside}`, где `Inside` — число точек, попавших внутрь круга, а `Outside` — число точек, в круг не попавших. Отправляющую запрос функцию разумно оформить так, чтобы она сразу вычисляла приближённое значение числа π по формуле

$$\frac{Inside}{Inside + Outside} \times 4$$

Обратите внимание, что код разбора входящих сообщений в вычисляющей функции должен предусматривать тайм-аут, после которого ожидание

[2] *Не забывайте, что бесконечная рекурсия может быть только хвостовой*

сообщений прекращается, иначе следующий шаг вычислений будет происходить не раньше, чем запрошены результаты предыдущего.

То, что для передачи сообщения надо знать идентификатор процесса, который меняется при каждом новом выполнении программы, не всегда удобно: ведь это означает, что при запуске процесса нам надо сохранить его идентификатор и передать этот идентификатор всем процессам, которым в будущем может потребоваться отправить ему сообщения.

Поэтому в Эрланге есть механизм *регистрации процессов*, позволяющий обращаться к процессу по зарезервированному имени. Работает регистрация процессов очень просто: мы вызываем функцию `register/2`, которая принимает атом, назначаемый именем, и идентификатор процесса. Если переданное имя ещё не занято, функция регистрирует на него процесс, и дальше мы можем использовать имя процесса вместо его идентификатора; `register/2` в этом случае возвращает `true`. В случае, если процесс зарегистрировать не получилось, функция `register/2` выдаст ошибку.

Зарегистрируем калькулятор со страницы 66 как процесс с именем `calcserver`:

```
177> register(calcserver, spawn(fun calc:calcserver/0)).
true
```

Функция `spawn` вернула идентификатор запущенного процесса, который был сразу передан функции `register/2`. Так как процесса с именем `calcserver` в системе зарегистрировано ещё не было, регистрация прошла успешно, о чём свидетельствует возвращённый результат — `true`.

Проверим возможность обращаться к процессу по имени:

```
178> calc:call(calcserver, '+', 2, 3).
{ok,5}
```

Второй процесс с таким же именем в систем зарегистрировать не удастся:

```
179> register(calcserver, spawn(fun calc:calcserver/0)).
** exception error: bad argument
     in function  register/2
        called as register(calcserver,<0.49.0>)
```

Узнать, зарегистрирован ли уже процесс с таким именем в системе, можно с помощью функции `whereis`:

```
180> whereis(calcserver).
<0.38.0>
```

Обсудим случай, когда удобно применять зарегистрированный процесс, а заодно изучим полезную технику ускорения работы программ.

В качестве примера возьмём функцию, возвращающую n-ное число Фибоначчи. Напомним, что числа Фибоначчи — последовательность, вычисляемая по формуле:

$$F_n = \begin{cases} 1, & n = 1, \\ 1, & n = 2, \\ F_{n-2} + F_{n-1}, & n > 2. \end{cases}$$

В разделе 3.4 мы научились переводить такие определения в определения функций на Эрланге напрямую. Соответственно, из определения чисел Фибоначчи получится такое определение функции:

```
fib(1) ->
    1;
fib(2) ->
    1;
fib(X) ->
    fib(X - 2) + fib(X - 1).
```

Эта функция красиво выглядит и хорошо читается, но, к сожалению, очень неэффективна. Дело в том, что она многократно повторяет одни и те же вычисления.

На рисунке 6.3 изображена схема вычисления 6-го числа Фибоначчи с помощью функции `fib`[3]. Можно видеть, что, например, при вызове `fib(4)` из тела функции `fib(6)` полностью повторяются все действия, выполненные при вызове `fib(4)` во время вычисления `fib(5)`.

Поэтому с ростом номера числа Фибоначчи количество шагов, совершаемых этой функцией, будет расти очень быстро[4]. На самом деле, её производительность настолько плоха, что делает её практически неприменимой (на компьютере, на котором я пишу этот текст, время вычисления 37-го числа Фибоначчи, измеренное с помощью вызова `timer:tc(fib, fib, [37]).`, составило почти две секунды, а 42-го — уже 21 секунду).

Разумеется, конкретно для вычисления чисел Фибоначчи есть существенно более эффективные алгоритмы. Но существует и общий метод

[3]Рисунок предоставлен Г.К.Бронниковым, переводчиком «Структуры и интерпретации компьютерных программ» на русский язык; я заменил нумерацию с нуля на применяемую в этой книге нумерацию с единицы.

[4]Если быть точным, экспоненциально относительно роста аргумента.

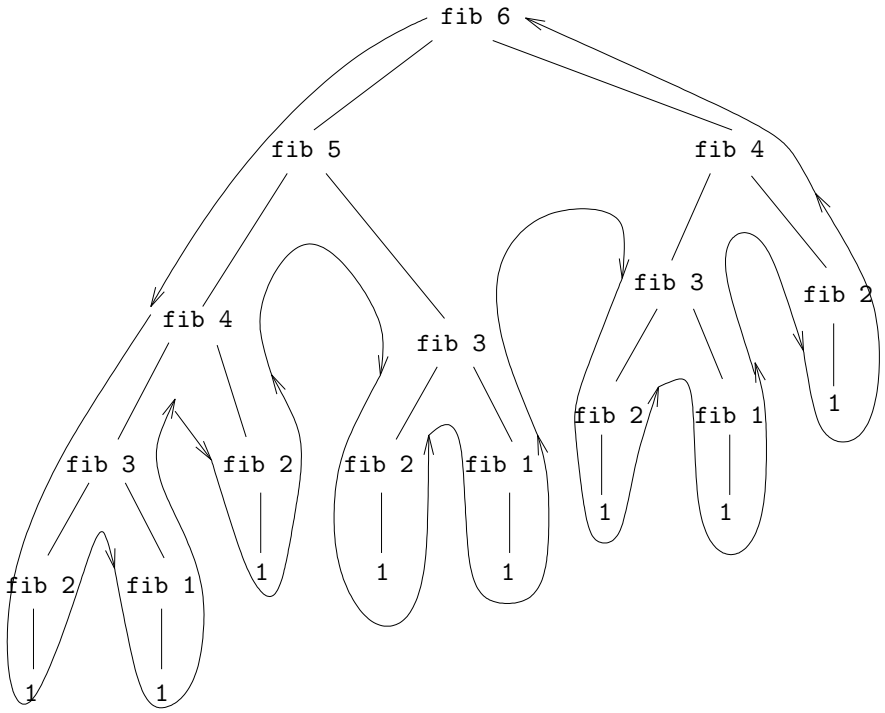


Рис. 6.3.: Вычисление `fib(6)` с помощью древовидной рекурсии (иллюстрация из книги «Структура и интерпретация компьютерных программ»[Абельсон 2010]).

ускорения работы таких алгоритмов, ради которого мы и затеяли этот разговор — он называется *мемоизацией*.

Суть этого метода в том, что каждый раз, вычисляя функцию от какого-то аргумента, мы запоминаем полученный результат. В следующий раз, когда нам нужна функция от того же аргумента, мы обнаруживаем, что он уже вычислен, и вместо повторных вычислений используем запомненное значение. Если время на вычисление значения значительно превосходит время извлечения значения из хранилища результатов (а для многих алгоритмов это так, если мы, конечно, выбрали достаточно эффективное хранилище), то выигрыш во времени может быть очень существенным.

Ассоциативный массив (словарь) обеспечивает доступ к своим элементам не по индексам, а по ключам — связанным с элементами произвольным значениям. Ключами могут быть строки, числа или, иногда, даже значения более сложных типов.

Мемоизируем вычисление чисел Фибоначчи. В качестве хранилища уже вычисленных значений используем словарь (ассоциативный массив). Ключами словаря будут номера чисел Фибоначчи, а значениями — сами числа. Испольуем ту реализацию ассоциативных массивов, которую нам предлагает модуль `dict` из стандартной библиотеки Эрланга.

Создать такой словарь можно с помощью функции `new()`:

```
181> D = dict:new().
```

```
{dict,0,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
      {{[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]}}}
```

Возвращаемый этой операцией кортеж — это внутренняя структура словаря, которую мы сейчас обсуждать не будем.

Поместить значение в словарь можно с помощью функции `dict:append/3`, принимающей ключ, значение, которое будет храниться по этому ключу, и имя словаря. Это функция возвращает новый словарь, в который внесены запрошенные изменения:

```
182> D2 = dict:append(''A'', 1, D).
{dict,1,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
      {{[],[],[],[],[],[],[],[],[],[],[],[],[],[],['A',1]],[],[],[]}}

183> D3 = dict:append('B', 2, D2).
{dict,2,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
      {{[],
        ['B',2]],
        [],[],[],[],[],[],[],[],[],[],
        ['A',1]],
        [],[],[]}}}
```

По одному ключу можно хранить более одного значения, но сейчас нам эта возможность не понадобится.

Функция `find/2` осуществляет поиск в словаре и возвращает кортеж из атома `ok` и списка результатов в случае успеха или атом `error` в случае неудачи:

```
184> dict:find(''A'', D3).
{ok,[1]}

185> dict:find(''C'', D3).
error
```

Функция `fetch/2` выдаёт список хранящихся по заданному ключу значений; в случае, если такой ключ в словаре отсутствует, она завершается с ошибкой.

```
186> dict:fetch('B', D3).
[2]
```

Мемоизированная с помощью словаря из модуля `dict` функция вычисления чисел Фибоначчи будет выглядеть следующим образом:

```
fibmem(1, Cache) ->
    {1, Cache};
fibmem(2, Cache) ->
    {1, Cache};
fibmem(N, Cache) ->
    case dict:find(N, Cache) of
        {ok, [Res]} ->
            {Res, Cache};
        error ->
            {X, TmpCache} = fibmem(N - 2, Cache),
            {Y, _} = fibmem(N - 1, TmpCache),
            Res = X + Y,
            NewCache = dict:append(N, Res, TmpCache),
            {Res, NewCache}
    end.
```

Разберём приведённый код.

Функция теперь принимает два аргумента — номер вычисляемого числа Фибоначчи и словарь, содержащий уже вычисленные значения. Первое и второе числа Фибоначчи по-прежнему заданы явно. А вот в рекурсивном варианте функции мы теперь первым делом проверяем, не содержится ли уже в словаре число Фибоначчи с заданным номером. В случае, если да (первый образец в операторе `case`), мы просто возвращаем результат вместе с текущим состоянием словаря. Если такого числа в словаре ещё нет (второй образец в операторе `case`), мы вычисляем это число и помещаем его в словарь, а потом возвращаем вычисленное значение и новое состояние словаря. Обратите внимание на анонимную переменную во втором рекурсивном вызове — так как второе используемое число меньше первого, мы знаем, что при его вычислении в словаре не появится новых значений. В других случаях древовидной рекурсии это, разумеется, может быть не так.

Измерим получившийся выигрыш во времени. Всё на том же компьютере, на котором измерялась производительность «наивной» реализации, вычисление 37-го числа Фибоначчи с исходно чистым словарём заняло 515 микросекунд, а 42-го — 673 микросекунды. Если же при вычислении 42-го числа Фибоначчи использовать словарь, образовавшийся при вычислении 37-го, функция выполняется всего за 71 микросекунду. Повторный вызов с тем же аргументом, использующий уже хранящееся в словаре значение, занял всего 4 микросекунды.

Впечатляющее улучшение! Однако, у мемоизированных таким спосо-

бом функций есть и недостаток: при их использовании нам всё время придётся помнить о хранилище уже вычисленных результатов — передавать его как дополнительный аргумент, запоминать его новое состояние, где-то хранить между вызовами нашей функции и т.д. Это затрудняет чтение кода и является потенциальным источником ошибок, не говоря уже о том, что с такими функциями трудно использовать некоторые удобные возможности языка, например, списковые включения.

Можно ли сделать мемоизацию незаметной для использующего функцию программиста? Если бы мы могли использовать присваивание, проблема решалась бы глобальной переменной, в которой уже вычисленные значения хранились бы между вызовами функций. В объектном языке ту же переменную можно было бы «спрятать» внутри класса.

В Эрланге возможно другое решение: выделить работу с таблицей уже вычисленных значений в отдельный процесс[5]. Всю работу по мемоизации мы при этом можем спрятать в определении функции, избавив программиста от необходимости думать об этом при её вызове.

Тут-то нам и пригодится возможность регистрировать процессы: вместо того, чтобы как-то хранить идентификатор процесса, хранящего таблицу вычисленных значений (что приносило бы те же проблемы, что и хранение ссылки на саму таблицу), мы будем обращаться к процессу по заранее известному имени.

Задача 6.2 *Реализуйте мемоизированную версию функции, вычисляющей числа Фибоначчи. Она должна принимать только один аргумент (номер числа в последовательности), а словарь с вычисленными значениями должен сохраняться между вызовами. Для этого работу с ним выделите в отдельный процесс.*

6.2. Азы распределённого программирования

Мы научились передавать сообщения между процессами. Но все эти процессы запускались внутри одного экземпляра Эрланг-машины. Мы знаем, как сократить и упростить текст программы, разбив её на несколько процессов, и, возможно, эти несколько процессов справятся с задачей чуть быстрее, чем один (за счёт исполнения на разных ядрах процессора). Но Эрланг может гораздо больше — он позволяет нам писать программы, которые будут работать на многих компьютерах, почти так же легко, как программы из нескольких процессов, работающих на одной машине.

[5] В некоторых функциональных языках, скажем, в Haskell, существуют другие решения этой проблемы, позволяющие мемоизировать функцию без нарушения функциональной чистоты. Но в Эрланге нет средств, которые эти решения используют.

Разработчикам языка хотелось, чтобы программы на Эрланге обладали двумя свойствами:

- *масштабируемостью* (чтобы увеличить производительность программы можно было, просто увеличив количество компьютеров, на которых она запущена) и
- *отказоустойчивостью* (чтобы программа продолжала работать, даже если часть компьютеров, на которых она запущена, вышла из строя).

Разумеется, Эрланг — не единственный язык, на котором можно писать такие программы. Но разработчики сделали всё, чтобы на Эрланге такие программы писались настолько просто и приятно, насколько это вообще возможно. Давайте посмотрим, какие механизмы они для этого предусмотрели.

Распределённая система на Эрланге строится из отдельных экземпляров Эрланг-машины, которые называются узлами. Начнём с того, что запустим два узла на одной и той же машине. Для этого надо дать команду `erl` в двух эмуляторах терминала, запущенных, например, в соседних окошках. Чтобы узлы знали, как обращаться друг к другу, надо передать команде *имя узла* с помощью ключа `-sname`:

Первое окно	Второе окно
`erl -sname hog`	`erl -sname pig`

В этом примере мы дали запущенному в первом окне узлу имя `hog`, а запущенному во втором — имя `pig`. В результате мы получим знакомое нам приглашение, в котором будет указано имя узла — для узла `hog` это будет:

```
(hog@localhost)1>
```

а для узла `pig`:

```
(pig@localhost)1>
```

Если машине задано сетевое имя, вместо `localhost` вы можете увидеть это имя.

Имя узла — это атом, поэтому должно состоять из латинских букв в нижнем регистре или из любых символов, окружённых апострофами.

Для того, чтобы отправить сообщение процессу на другом узле, применяется тот же оператор «`!`», только вместо идентификатора или имени

процесса указывается кортеж из имени или идентификатора и узла, на котором процесс запущен. Проиллюстрируем это на примере сервера-калькулятора, описанного на странице 66.

Запустим сервер (функцию `calcserver`) под именем `calcserver` на узле `hog`:

```
1> register(calcserver, spawn(fun calc:calcserver/0)).
true
```

Теперь обратимся к нему с узла `pig`:

```
(pig@localhost)187> calc:call({calcserver, hog@localhost}, {'+',
2, 3}).
{ok,5}
```

Обращаться можно не только к зарегистрированным процессам — идентификаторы безымянных процессов можно передавать между узлами как сообщения. Хотя, конечно, начать обмен идентификаторами без какого-нибудь именованного процесса не получится.

В качестве ещё одного примера обменяемся сообщениями между самими оболочками интерпретаторов на узлах `hog` и `pig`. Сначала воспользуемся тем, что оболочка — такой же процесс Эрланга, как и любой другой, и может быть поименована с помощью команды `register`. Перейдём в оболочку узла `hog` и выполним команду:

```
(hog@localhost)188> register(hogshell, self()).
true
```

Теперь прямо из оболочки выполним конструкцию `receive`, которая просто вернёт любое полученное сообщение, и запомним результат в переменной `Pid`:

```
(hog@localhost)189> Pid = receive
(hog@localhost)189>           X -> X
(hog@localhost)189>       end.
```

Оболочка приостановит выполнение до попадания сообщений в почтовый ящик процесса. Теперь перейдём в оболочку узла `pig`. Оболочка узла `hog` зарегистрирована на своём узле как `hogshell`, так что мы можем сразу отправить ей сообщение. Передадим ей идентификатор процесса оболочки узла `pig`:

```
(pig@localhost)190> {hogshell, hog@pterodactyl} ! self().
<0.40.0>
```

Перейдём в окно оболочки узла `hog` и увидим, что сообщение получено:

```
(hog@localhost)191> Pid = receive
(hog@localhost)191>           X -> X
(hog@localhost)191>       end.
<6832.40.0>
```

Строковое представление идентификатора процесса отличается на разных узлах, это нормально. Отправим оболочке узла `pig` что-нибудь в ответ. Но для этого надо запустить `receive` в этой оболочке:

```
(pig@localhost)192> Answer = receive
(pig@localhost)192>              X -> X
(pig@localhost)192>          end.
```

Вернёмся в оболочку узла `hog` и отправим сообщение:

```
(hog@localhost)193> Pid ! ''Hello, distributed world!''.
''Hello, distributed world!''
```

Обратите внимание, что если у нас есть идентификатор процесса, запущенного на другом узле, имя узла нам не нужно — идентификатор уже содержит эту информацию.

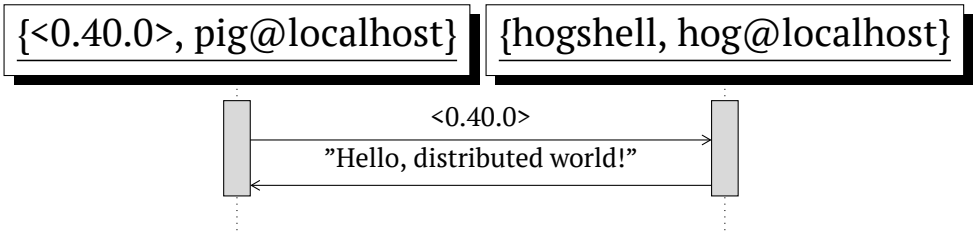


Рис. 6.4.: Оболочки на узлах `pig@localhost` и `hog@localhost` общаются друг с другом.

Перейдём в оболочку узла `pig` и убедимся, что сообщение принято:

```
(pig@localhost)194> Answer = receive
(pig@localhost)194>              X -> X
(pig@localhost)194>          end.
''Hello, distributed world!''

(pig@localhost)195> Answer.
''Hello, distributed world!''
```

Задача 6.3 *Напишите два процесса, отслеживающие состояние друг друга: один из них должен посылать другому сообщение с заданным интервалом, а второй — посылать ответ. Если отправитель за минуту не дождался ответа, а получатель уже минуту не получал сообщений, на стандартный вывод должно отправляться сообщение об ошибке. Протестируйте вашу программу на одном узле и на разных.*

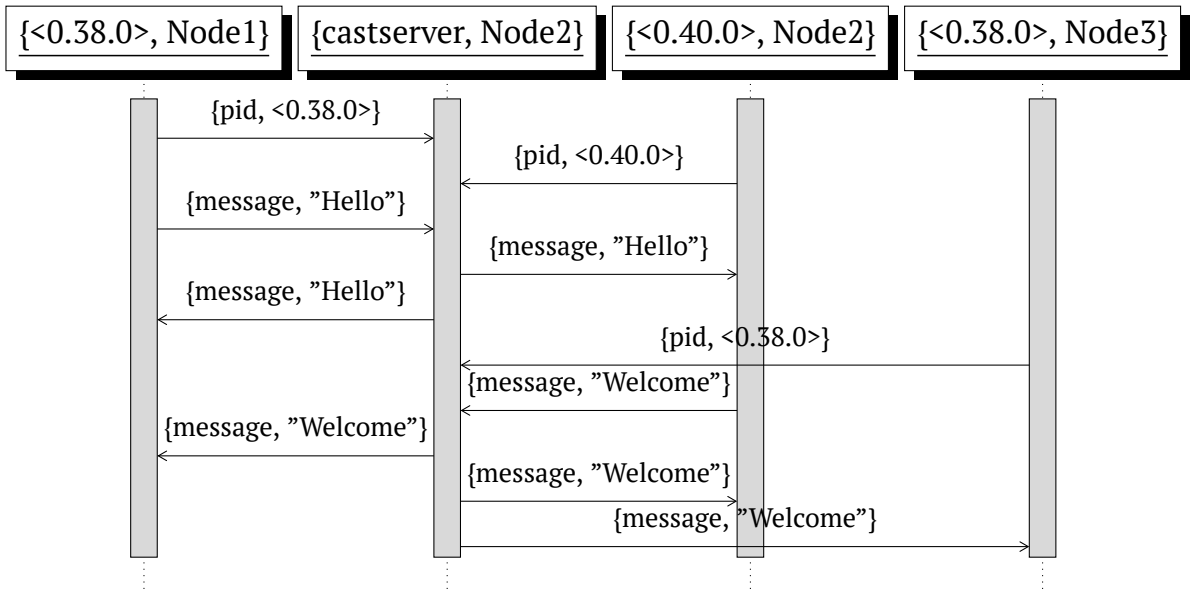


Рис. 6.5.: «Ретранслятор» из задачи 6.4 в работе.

Задача 6.4 *Напишите сервер-«ретранслятор», умеющий принимать два типа сообщений. Сообщение вида* `{pid, Pid}`*, где* `Pid` *— идентификатор процесса, добавляет процесс с таким идентификатором в список получателей. Сообщение вида* `{message, Message}` *приводит к тому, что* `Message` *рассылается всем адресатам из списка.*

7. Что дальше?

7.1. Что мы узнали о функциональном программировании

Сформулируем главные черты функциональных языков, которые мы узнали из этого курса.

В разделе 2.3 на странице 14 мы узнали, что в функциональных языках нет операции присваивания в привычном нам смысле: единожды связав переменную со значением, изменить его мы не можем.

В разделе 3.4 на странице 36 выяснилось, как функциональные языки обходятся без циклов типа `for` для описания повторяющихся действий: с помощью рекурсивных функций. А в разделе 4.1 мы разобрали, как рекурсия превращается в полноценную замену циклам с помощью оптимизации хвостовых вызовов.

Наконец, в разделах 4.2 и 4.3 показано свойство функциональных языков, без которого писать на них было бы очень тяжело — функции в них это «полноправные граждане», такие же значения, как символы или числа.

Итак, функциональный язык — это язык, в котором

- нет присваивания, то есть возможности изменять значение переменной;
- есть хвостовая рекурсия;
- функции являются полноправными значениями.

Такие свойства языка обеспечивают множество выгод. Но главная из них — возможность создавать короткие и понятные программы, которые легко читать и изменять. Надеюсь, что примеры и задачи вполне убедили в этом читателя.

7.2. Какие ещё бывают функциональные языки

На рис. 7.1 изображена родословная функциональных языков. Их, конечно, гораздо больше, изображены только те, которые очень популярны

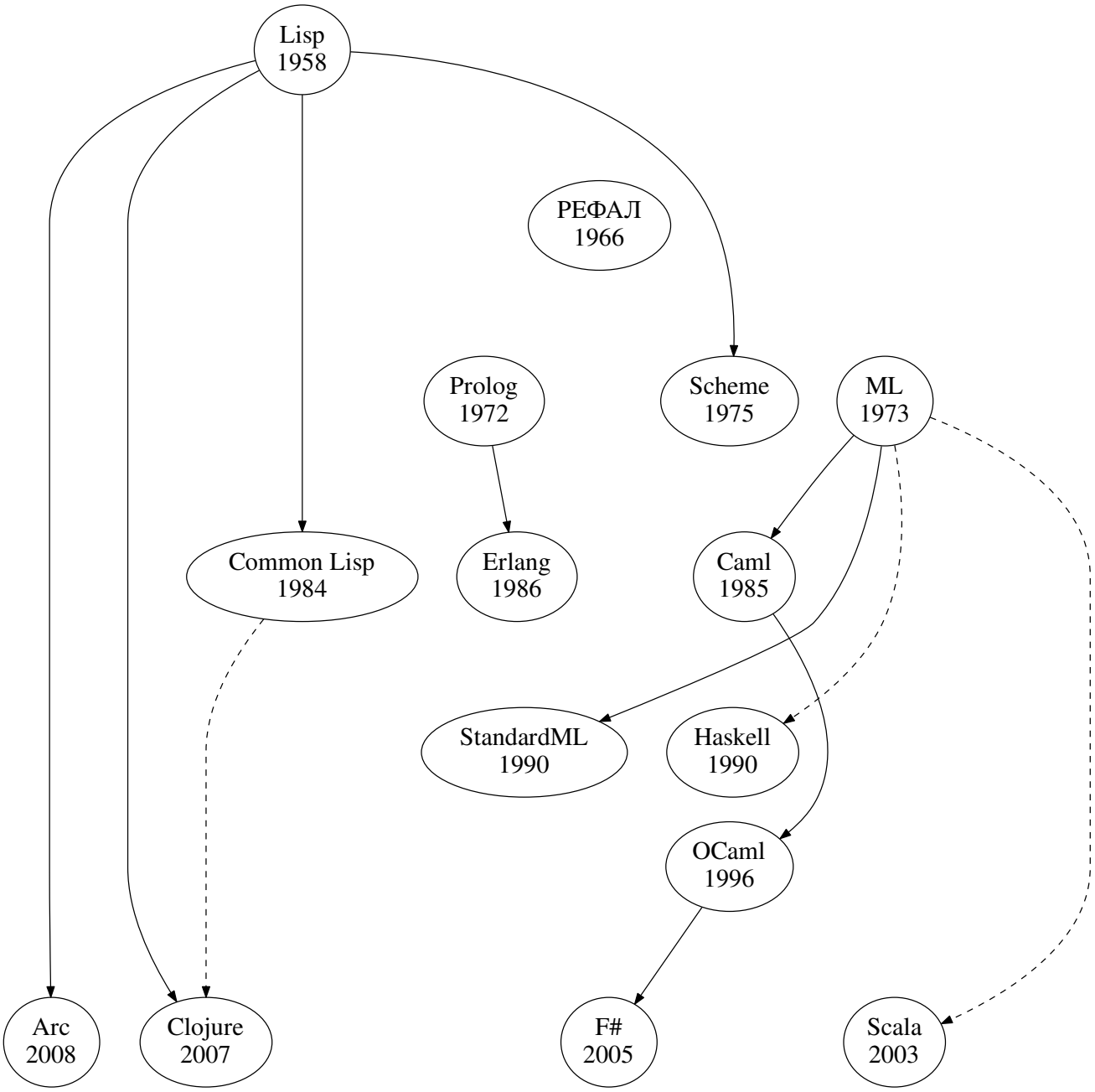


Рис. 7.1.: Родословное древо функциональных языков

или оказали на развитие функционального программирования большое влияние.

Как уже упоминалось в разделе 1.2, функциональные языки существуют довольно давно. Первый из них, Лисп, создан в 1958 году, всего на год позже первого императивного языка — Фортрана. Ещё один известный функциональный язык, РЕФАЛ, создан В.Ф.Турчиным в 1966 году. Но если РЕФАЛ стал широко известен довольно поздно (возможно, по причинам, далёким от программирования: он был создан в СССР в эпоху Холодной войны, а ФП развивалось в то время в основном по другую сторону Железного занавеса), то распространение Лиспа продолжалось

всё время его существования. Уже в двухтысячных годах появились два новых популярных диалекта этого языка: Arc и Clojure.

Наиболее известные диалекты Лиспа — Common Lisp и Scheme. Первый чаще других вариантов Лиспа применяется на практике, а вторая используется в основном для обучения программированию. К сожалению, непоследовательная политика стандартизации сильно замедлила распространение Scheme как языка практического программирования.

Изучение Лиспа (скорее всего, Scheme) практически неизбежно для человека, интересущегося функциональным программированием, так как многие важные работы на эту тему используют именно Scheme в качестве языка примеров. Среди возможностей Scheme, отсутствующих в Эрланге, особенно интересны продолжения (continuations) — способ сохранять как значение состояние программы в определённый момент.

Второй после Лиспа по влиятельности и многочисленности «потомства» функциональный язык это ML. В нём впервые была реализована очень мощная техника — *вывод типов*, то есть автоматическое определение компилятором типа выражения по контексту, в котором оно используется. Вывод типов избавляет программиста от целого класса неприятных ошибок. В этой книжке я использовал язык без вывода типов[1] только потому, что это сильно усложнило и удлинило бы курс.

Наиболее известные потомки ML это OCaml, F# (построенный на базе OCaml язык для платформы .Net) и Haskell.

Именно Haskell считается «передним краем» исследований в области функционального программирования. В практическом программировании он применяется не так часто, возможно, потому, что его авторы очень жёстко отказываются вносить в язык полезные для индустрии, но «некрасивые» с точки зрения теории возможности — полушутливый девиз Haskell это «Avoid success at all costs» («Избегать успеха любой ценой»). Пожалуй, Haskell — это второй после Scheme язык, изучения которого интересующемуся ФП человеку вряд ли удастся избежать. Изучить этот язык стоит хотя бы для того, чтобы познакомиться с такими интереснейшими концепциями, как ленивые вычисления или монады.

К сожалению, малопопулярен другой замечательный диалект ML — StandardML, очень красивый, гораздо более компактный, чем Haskell (и, на мой взгляд, более логичный синтаксически, чем OCaml и F#) язык.

Ну и, наконец, не надо забывать, что Эрланг мы тоже изучили не до конца. Если вам понравился Эрланг, стоит всерьёз заняться изучением его возможностей. На самом деле, в этом курсе вообще не используется одно из главных преимуществ Эрланга — библиотека OTP, Open Telecom

[1] На самом деле, похожие инструменты для Эрланга реализованы, но они гораздо слабее своих аналогов из языков-потомков ML.

Platform, предоставляющая готовые решения для задач параллельного и распределённого программирования. «За кадром» остались также поведения (behaviours), обработка исключительных ситуаций, макросы и многое другое.

7.3. Что ещё можно почитать про ФП

Функциональному программированию посвящено довольно много книг, ресурсов в Интернете и журналов. Хорошей стартовой точкой для дальнейшего знакомства с миром ФП может стать журнал «Практика функционального программирования», свободно доступный на сайте `http://fprog.ru/`. Кстати, в первом номере журнала опубликован обзор литературы по функциональному программированию, сделанный Алексеем Оттом [Отт 2009].

Ещё одна хорошая обзорная статья — «Функциональное программирование для всех» В.Ахмечета [Ахмечет 2006], опубликованная в журнале RSDN Magazine.

Среди книг прежде всего надо упомянуть «Структуру и интерпретацию компьютерных программ» Х.Абельсона и Дж.Сассмана [Абельсон 2010], которая по праву считается классической. Это учебник по программированию вообще, не только функциональному, но значительная часть материала изложена именно с помощью функциональных решений.

Большинство работ по алгоритмам и структурам данных расчитано на императивные языки. Эту ситуацию пытается исправить Крис Окасаки, автор книги «Чисто функциональные структуры данных» [Okasaki 1999]. Ещё одна серьёзная работа по теоретическим аспектам программирования, ориентированная на ФП — «Типы в языках программирования» Б.Пирса [Пирс 2012].

Очень хороший пример практического программирования на функциональных языках приводится в статье Д.Астапова «Давно не брал я в руки шашек» [Астапов 2009]. Интересующимся продвинутыми программистскими техниками будет, вероятно, интересно прочитать статьи А.Вознюка о продолжениях [Вознюк 2011] и И.Ключникова о суперкомпиляции [Ключников 2011]. Впрочем, как уже говорилось, журнал «Практика функционального программирования», в котором опубликованы все эти статьи, вообще рекомендуется к ознакомлению при наличии интереса к теме.

Замечательные учебники по языку Scheme и функциональному программированию — книги «Little Schemer» [Friedman 1996] и «Seasoned Schemer» [Friedman 1996,2], к сожалению, в отличие от SICP не переведены на русский язык.

Хороший вводный материал по языку Haskell — цикл статей Алексея Бешенова [Бешенов 2010] и книга «Изучай Haskell во имя добра!» [Липовача 2012], а если говорить об источниках на английском языке — «Real World Haskell» [O'Sullivan 2008]. По Стандартному ML на русском издана книга Р.Харпера «Введение в Стандартный ML» [Харпер 1996].

Про возможности OTP, концепцию «поведений» (behaviours) можно прочитать в книге [Чезарини 2012]. Из важных книг по Эрлангу, пока не переведённых на русский язык, надо упомянуть учебник Дж.Армстронга (создателя языка) [Armstrong 2007] и книгу «Erlang and OTP in action» [Logan 2011]. Хороший вводный курс, правда, во многом дублирующий уже изложенное здесь — «Learn You Some Erlang for Great Good» [LYSE].

Решения задач

Азбука ФП

Задача 3.1

```
-module(logic).
-export([nand/2]).

nand(true, true) -> false;
nand(false, true) -> true;
nand(true, false) -> true;
nand(false, false) -> true.
```

Вообще говоря, функцию `nand` можно было бы определить гораздо проще:

```
nand(true, true) -> false;
nand(_, _) -> true.
```

Но этот вариант функции `nand` не будет выдавать ошибку, если передать ему в качестве аргументов что-то помимо атомов `true` и `false`, что может затруднить поиск источника проблемы в неправильно написанной программе. Поэтому лучше перечислить все корректные варианты аргументов явно.

Помните, в Эрланге принят подход «Let it fail» — «Пусть падает».

Задача 3.2

```
-module(logic).
-export([nand/2, my_not/1, my_and/2, my_or/2]).

nand(true, true) -> false;
nand(false, true) -> true;
nand(true, false) -> true;
nand(false, false) -> true.

my_not(X) -> nand(X, X).
```

```
my_and(X, Y) -> my_not(nand(X, Y)).

my_or(X, Y) -> nand(my_not(X), my_not(Y)).
```

Задача 3.3

```
-module(author).
-export([hello/0]).

hello() ->
    {ok, [Username]} = io:fread("Enter your name: ", "~s"),
    case Username of
        "Yuriy" -> io:format("Hello, author!~n");
        _ -> io:format("Hello, ~s!~n", [Username])
    end.
```

Задача 3.4

```
-module(sign).
-export([sign/1]).

sign(N) ->
    if
        N <   0 -> -1;
        N =:= 0 -> 0;
        N >   0 -> 1
    end.
```

Задача 3.5

```
-module(divisor).
-export([is_divisor/2]).

is_divisor(N, M) when N rem M =:= 0 -> true;
is_divisor(_, _) -> false.
```

Задача 3.6

```
-module(summ).
-export([summ/0]).

summ(Summ, 0) -> Summ;
summ(Summ, Value) ->
```

```
    NewSumm = Summ + Value,
    {ok, [NewValue]} = io:fread(''Enter a number: '', ''~d''),
    summ(NewSumm, NewValue).

summ() ->
    {ok, [Value]} = io:fread(''Enter a number: '', ''~d''),
    summ(0, Value).
```

Задача 3.7

```
-module(divisors).
-export([divisors/1]).

divisors(N, Limit, Limit) ->
    io:format(''~w~n'', [N]);
divisors(N, Limit, Current) when N rem Current =:= 0 ->
    io:format(''~w '', [Current]),
    divisors(N, Limit, Current+1);
divisors(N, Limit, Current) ->
    divisors(N, Limit, Current+1).

divisors(N) ->
    Limit = (N div 2) + 1,
    divisors(N, Limit, 1).
```

Задача 3.8

```
-module(remainder).
-export([remainder/2]).

remainder(N, M) when N >= M -> remainder(N - M, M);
remainder(N, _) -> N.
```

Задача 3.9

```
-module(number_input).
-export([read_numbers/0]).

read_numbers() ->
    {ok, Number} = io:read(''> ''),
    case Number of
        0 -> ok;
        _ -> read_numbers()
```

```
    end,
    io:format("~w~n", [Number]).
```

Эту задачу можно решить и иначе — например, вот так:

```
-module(number_input).
-export([read_numbers/0]).

read_numbers(0) -> ok;
read_numbers(Previous) ->
    {ok, Next} = io:read("> "),
    read_numbers(Next),
    io:format("~w~n", [Previous]).

read_numbers() ->
    {ok, Next} = io:read("> "),
    read_numbers(Next).
```

Идея решения та же самая, а как её оформить — вопрос вкуса. В таком варианте приодится дублировать оператор ввода, зато вместо `case` используется сопоставление с образцом, что, по идее, должно делать код более понятным.

Задача 3.10

```
-module(rember).
-export([rember/2]).

rember([], _) -> [];
rember([X|Tail], X) -> rember(Tail, X);
rember([Head|Tail], X) -> [Head|rember(Tail, X)].
```

Задача 3.11

```
-module(listrev).
-export([listrev/1]).

listrev([], Res) -> Res;
listrev([H|T], Res) -> listrev(T, [H|Res]).

listrev(L) -> listrev(L, []).
```

Задача 3.12

```
-module(slice).
```

```
-export([slice/3]).

slice([H|_T], _From, To, To) ->
    [H];
slice([_H|T], From, To, Cur) when Cur < From ->
    slice(T, From, To, Cur+1);
slice([H|T], From, To, Cur) ->
    [H|slice(T, From, To, Cur+1)].

slice(L, From, To) -> slice(L, From, To, 0).
```

Задача 3.13

Алгоритм решения этой задачи можно коротко описать так: мы создаём временный список для хранения текущего разбираемого элемента (второй аргумент функции `flatten/2`). Поначалу он пуст (это тот второй аргумент, который передаёт функции `flatten/2` функция `flatten/1`).

Если исходный список тоже пуст, то работа закончена (первый вариант функции).

Если исходный список не пуст, то мы берём его голову и помещаем во временный список для разбора (второе тело функции).

Если текущий разбираемый элемент это тоже список, то мы берём его голову, а хвост возвращаем в исходный список, где он будет дожидаться своей очереди (третье тело функции).

Наконец, если ни одно из предыдущих сопоставлений с образцом не увенчалось успехом, надо полагать, что хранилище разбираемых элементов не пусто (иначе бы сработали бы образцы 1 или 2) и то, что в нём лежит — не список (иначе сработал бы образец 3). Следовательно, это одиночное значение и мы можем смело помещать его в голову итогового списка.

```
-module(flatten).
-export([flatten/1]).

flatten([], []) -> [];
flatten([H|T], []) -> flatten(T, H);
flatten(L, [H|T]) -> flatten([T|L], H);
flatten(L, Value) -> [Value|flatten(L, [])].

flatten(L) -> flatten(L, []).
```

Задача 3.14

```
-module(palindrome).
-export([is_palindrome/1]).

is_palindrome(L) -> L =:= lists:reverse(L).
```

Задача 3.15

Функция `divisors`, составляющая список делителей числа N, лежащих в интервале от 1 до M включительно — это попросту индуктивное определение такого списка: если он заканчивается единицей, то только единица в него и войдёт, а если он должен состоять из чисел в интервале от единицы до некоего числа M, большего, чем единица, то в него войдут делители числа N в интервале от 1 до $M-1$, а также само M, если оно тоже является делителем N.

Случай с единицей мы обрабатываем в `divisors/1`, чтобы избежать лишнего вызова функции.

Функция `lists:sum` из модуля `lists`, подсчитывающая сумму списка чисел, ощутимо упрощает решение этой задачи.

Обратите внимание, что функция `divisors` небезопасна и будет зацикливаться при значениях M меньше 1. Здесь было разумным пожертвовать безопасностью в ущерб ясности; в реальных программах часто полезно делать наоборот.

```
-module(perfect).
-export([is_perfect/1]).

divisors(_, 1) -> [1];
divisors(N, M) when N rem M =:= 0 ->
    [M|divisors(N, M-1)];
divisors(N, M) ->
    divisors(N, M-1).

divisors(1) -> [1];
divisors(N) -> divisors(N, N div 2).

is_perfect(N) -> N =:= lists:sum(divisors(N)).
```

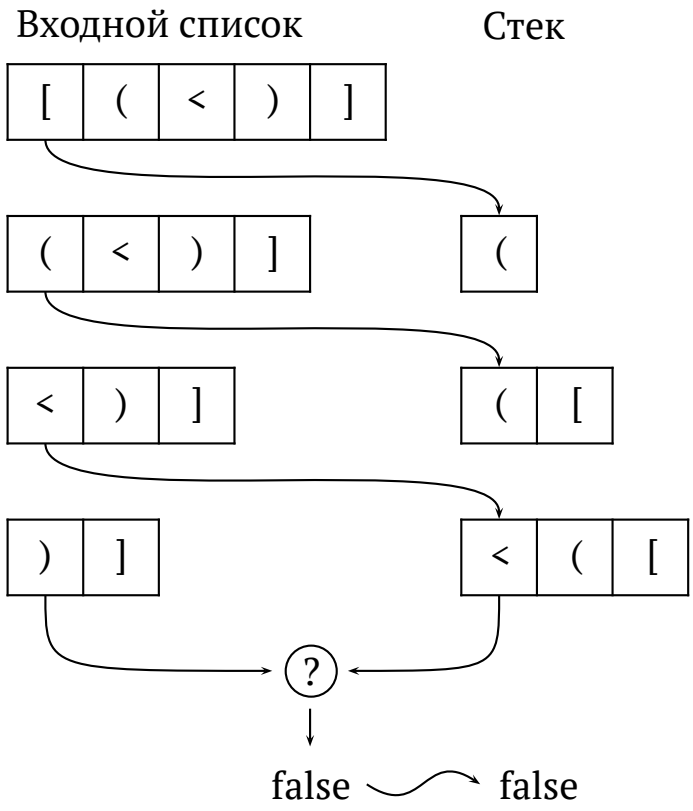


Рис. 7.2.: Функция `is_valid_string/2` проверяет скобочную структуру «[(<)]» на правильность. Слева — входной список (первый аргумент функции), справа — стек (второй аргумент функции).

Задача 3.16

В этой задаче можно представлять строку для разбора либо как обычный эрланговский список, либо как битовую строку. Решение с битовыми строками чуть более громоздко, но в реальной жизни битовые строки гораздо полезнее «строк», хранящихся как списки ASCII-кодов.

Алгоритм же в обоих решениях — один и тот же: сделаем стек, в который будем класть все открывающие скобки. Встречая закрывающую скобку, будем выпихивать со стека верхнее значение. Если тип закрывающей скобки из текущей позиции строки и открывающей скобки с вершины стека не совпали (или стек пуст), порядок скобок неправильный. Если после того, как вся строка разобрана, в стеке что-то осталось, порядок скобок тоже неправильный. Если же нам удалось подойти к концу строки с пустым стеком (или строка просто пустая), значит, порядок скобок — правильный.

```
-module(brackets_str).
-export([is_valid_string/1]).
```

```
is_valid_string("", []) -> true;
is_valid_string("", _) -> false;
is_valid_string([$(|StrT], Stack) ->
    is_valid_string(StrT, [$(|Stack]);
is_valid_string([$[|StrT], Stack) ->
    is_valid_string(StrT, [$[|Stack]);
is_valid_string([$<|StrT], Stack) ->
    is_valid_string(StrT, [$<|Stack]);
is_valid_string([$)|StrT], [$(|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string([$]|StrT], [$[|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string([$>|StrT], [$<|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string(_, _) -> false.

is_valid_string(S) -> is_valid_string(S, []).
```

```
-module(brackets).
-export([is_valid_string/1]).

is_valid_string(<<>>, []) -> true;
is_valid_string(<<>>, _) -> false;
is_valid_string(<<"(", StrT/binary>>, Stack) ->
    is_valid_string(StrT, ["("|Stack]);
is_valid_string(<<"[", StrT/binary>>, Stack) ->
    is_valid_string(StrT, ["["|Stack]);
is_valid_string(<<"<", StrT/binary>>, Stack) ->
    is_valid_string(StrT, ["<"|Stack]);
is_valid_string(<<")", StrT/binary>>, ["("|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string(<<"]", StrT/binary>>, ["["|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string(<<">", StrT/binary>>, ["<"|StackT]) ->
    is_valid_string(StrT, StackT);
is_valid_string(_, _) -> false.

is_valid_string(S) -> is_valid_string(list_to_binary(S), []).
```

Задача 3.17

Идея решения вот в чём: реализуем очередь как два списка, входной и выходной. При помещении значения в очередь будем добавлять его в голову входного списка. При извлечении значения из очереди проверим, пуст ли выходной список. Если да, проверим, пуст ли входной. Если пуст и он, очередь пуста (второй вариант тела функции). А если входной список не пуст, развернём его, отдадим голову получившегося списка как выходное значение, а хвост поместим в выходной список (входной список, соответственно, становится пустым). Эти действия описаны в первом варианте тела функции. Ну а если выходной список не пуст, просто отдаём его голову, т.к. значения в нём уже стоят в правильном порядке (третий вариант тела функции).

Работа с организованной таким образом очередью будет выглядеть примерно так.

Компилируем модуль, загружаем определение записи:

```
196> c(listqueue).
{ok,listqueue}

197> rr(''queue.hrl'').
[queue]
```

Создаём новую очередь:

```
198> Q1 = #queue{}.
#queue{in = [],out = []}
```

Помещаем в очередь значение 1:

```
199> Q2 = listqueue:enqueue(Q1, 1).
#queue{in = [1],out = []}
```

Помещаем в очередь значение 2:

```
200> Q3 = listqueue:enqueue(Q2, 2).
#queue{in = [2,1],out = []}
```

Помещаем в очередь значение 3:

```
201> Q4 = listqueue:enqueue(Q3, 3).
#queue{in = [3,2,1],out = []}
```

Извлекаем из очереди первое значение (1) — в этот момент входной список будет развёрнут и перемещён в выходной:

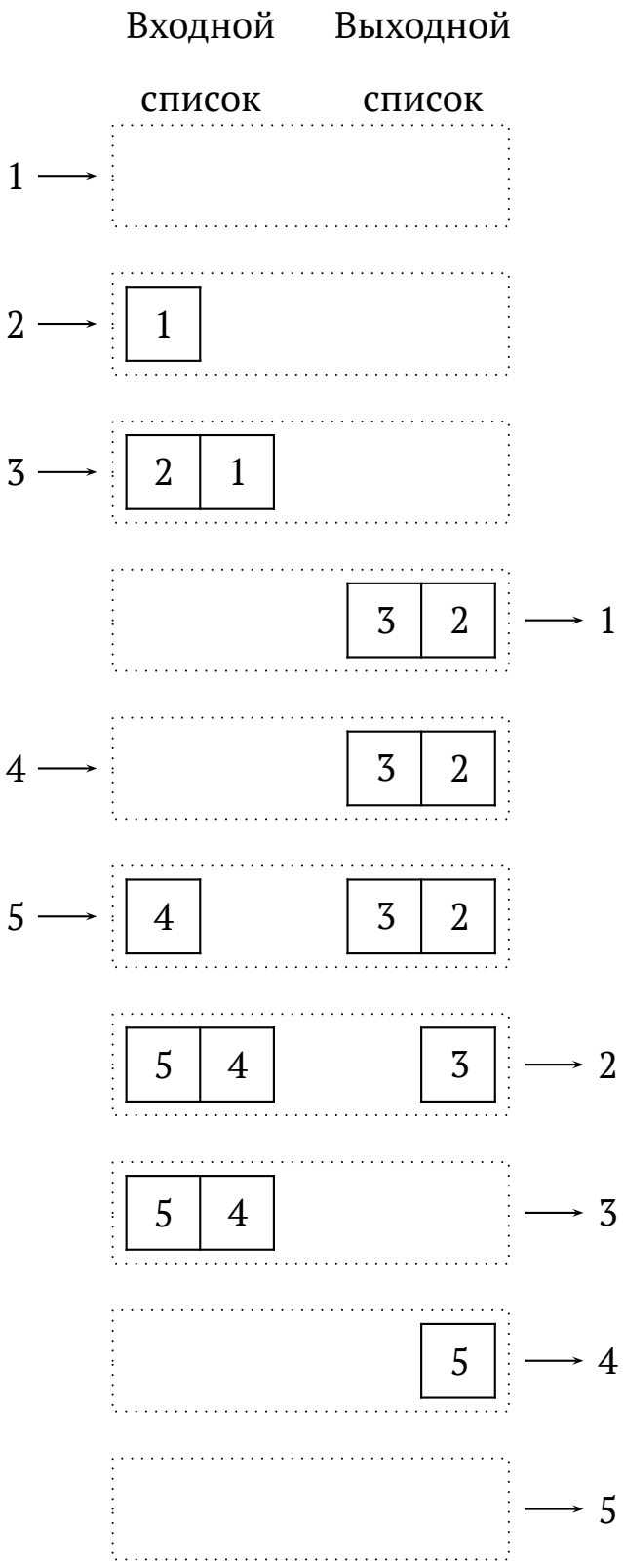


Рис. 7.3.: Организация очереди из двух списков.

```
202> {ok, Out1, Q5} = listqueue:dequeue(Q4).
{ok,1,#queue{in = [],out = [2,3]}}

203> Out1.
1
```

Добавляем в очередь значение 4 (с этого момента формируется новый входной список):

```
204> Q6 = listqueue:enqueue(Q5, 4).
#queue{in = [4],out = [2,3]}
```

Добавляем в очередь значение 5:

```
205> Q7 = listqueue:enqueue(Q6, 5).
#queue{in = [5,4],out = [2,3]}
```

Извлекаем из очереди второе значение (2):

```
206> {ok, Out2, Q8} = listqueue:dequeue(Q7).
{ok,2,#queue{in = [5,4],out = [3]}}
```

Извлекаем из очереди третье значение (3):

```
207> {ok, Out3, Q9} = listqueue:dequeue(Q8).
{ok,3,#queue{in = [5,4],out = []}}
```

Извлекаем из очереди четвёртое значение (4) — выходной список пуст, снова разворачиваем входной список и перемещаем его на место опустевшего выходного:

```
208> {ok, Out4, Q10} = listqueue:dequeue(Q9).
{ok,4,#queue{in = [],out = [5]}}
```

Извлекаем из очереди пятое значение (5):

```
209> {ok, Out5, Q11} = listqueue:dequeue(Q10).
{ok,5,#queue{in = [],out = []}}
```

Очередь пуста, при попытке извлечения дальнейших значений будет возвращена информация об ошибке:

```
210> listqueue:dequeue(Q11).
{error,''Queue is empty''}
```

Несмотря на наличие в этом алгоритме «тяжёлой» операции разворачивания списка, требующей N шагов для списка из N элементов, условиям задачи он отвечает, т.к. каждый проходящий через очередь элемент участвует в этой операции только один раз.

```
-module(listqueue).
-export([enqueue/2, dequeue/1]).

-record(queue, {in=[], out=[]}).

enqueue(Q, New) ->
    #queue{in=[New|Q#queue.in], out=Q#queue.out}.

dequeue(Q) when (Q#queue.out =:= []) and
                (Q#queue.in =/= []) ->
    [Value|Rest] = lists:reverse(Q#queue.in),
    {
      ok,
      Value,
      #queue{out=Rest, in=[]}
    };
dequeue(Q) when Q#queue.out =:= [] ->
    {
      error,
      ''Queue is empty''
    };
dequeue(Q) ->
    [Value|Rest] = Q#queue.out,
    {
      ok,
      Value,
      #queue{out=Rest, in=Q#queue.in}
    }.
```

Задача 3.18

Обратите внимание на то, как функции `trim/2` и `trim/3` рекурсивно вызывают друг друга. Это полезный приём, который позволяет упростить отладку и тестирование, а также улучшить читаемость кода за счёт избавления от громоздких операторов ветвления.

```
-module(trimtree).
```

```
-export([trim/2]).

-record(node, {value, left=nil, right=nil}).

trim(_Node, P, Fate) when Fate < P -> nil;
trim(Node, P, _Fate) ->
    #node {
      value = Node#node.value,
      left = trim(Node#node.left, P),
      right = trim(Node#node.right, P)
    }.

trim(nil, _P) -> nil;
trim(Node, P) ->
    Fate = random:uniform(),
    trim(Node, P, Fate).
```

Задача 3.19

```
-module(treemeter).
-export([measure/2]).

-record(node, {value, left=nil, right=nil}).

measure(nil,   _N, _Cur) -> 0;
measure(_Node, N,  N)    -> 1;
measure(Node,  N,  Cur)  ->
    measure(Node#node.left, N, Cur+1) +
        measure(Node#node.right, N, Cur+1).

measure(T, N) -> measure(T, N, 1).
```

Полезные приёмы

Задача 4.1

Хвостовая рекурсия уже используется в решениях задач 3.6, 3.7, 3.8, 3.11, 3.16.

Задача 4.2

Функцию `rember/2` из задачи 3.10 можно переписать следующим образом:

```
-module(tailrember).
-export([rember/2]).

rember([], _M, Acc) -> lists:reverse(Acc);
rember([M|T], M, Acc) -> rember(T, M, Acc);
rember([H|T], M, Acc) -> rember(T, M, [H|Acc]).

rember(L, M) -> rember(L, M, []).
```

Проблема в том, что такой способ обработки списка разворачивает его, и нам приходится вызывать `lists:reverse`, чтобы вернуть исходный порядок следования элементов. Поэтому применение здесь хвостовой рекурсии не очень оправдано. В решениях других задач применение хвостовой рекурсии принесёт ещё больше сложностей.

Не надо применять хвостовую рекурсию по любому поводу. Во-первых, она не всегда эффективнее обычной, во-вторых, помните: сначала стабильная работа программы, потом оптимизация.

Задача 4.5

```
-module(perfect).
-export([is_perfect/1, list_of_perfects/2]).

divisors(_, 1) -> [1];
divisors(N, M) when N rem M =:= 0 ->
    [M|divisors(N, M-1)];
divisors(N, M) ->
    divisors(N, M-1).

divisors(1) -> [1];
divisors(N) -> divisors(N, N div 2).

is_perfect(N) -> N =:= lists:sum(divisors(N)).

list_of_perfects(N, M) ->
    lists:filter(fun is_perfect/1, lists:seq(N, M)).
```

Задача 4.6

```
-module(maptree).
-export([maptree/2]).

-record(node, {value, left=nil, right=nil}).

maptree(nil, _) -> nil;
maptree(Node, F) ->
    #node {
         value = F(Node#node.value),
         left = maptree(Node#node.left, F),
         right = maptree(Node#node.right, F)
    }.
```

Задача 4.7

```
lists:foldl(fun(_, Length) -> Length + 1 end, 0, L).
```

Задача 4.8

```
-module(foldrev).
-export([rev/1]).

rev(L) ->
    lists:foldl(fun(T, H) -> [T|H] end, [], L).
```

Задача 4.9

```
-module(mean).
-export([listmean/1]).

listmean(L) ->
    {Num, Summ} = lists:foldl(
                    fun(Next, {N, S}) -> {N+1, S+Next} end,
                    {0, 0},
                    L
                  ),
    Summ / Num.
```

Задача 4.10

```
-module(mylists).
```

```
-export([foldr1/2, foldl1/2]).

foldr1(_, [X]) ->
    X;
foldr1(F, [H|T]) ->
    F(H, foldr1(F, T)).

foldl1(_, Acc, []) ->
    Acc;
foldl1(F, Acc, [H|T]) ->
    foldl1(F, F(Acc, H), T).

foldl1(F, [H|T]) ->
    foldl1(F, H, T).
```

Задача 4.11

```
-module(myseq).
-export([foldseq/3]).

foldseq(F, Acc, Finish, Finish) ->
    F(Acc, Finish);
foldseq(F, Acc, Cur, Finish) ->
    foldseq(F, F(Acc, Cur), Cur+1, Finish).

foldseq(F, Start, Finish) ->
    foldseq(F, Start, Start+1, Finish).
```

Задача 4.12

```
-module(ismemberof).
-export([ismemberof/1]).

ismemberof(L) ->
    fun(X) -> lists:member(X, L) end.
```

Приятные мелочи

Задача 5.1

```
-module(perfect).
-export([is_perfect/1]).
```

```
divisors(X) ->
    [Y || Y <- lists:seq(1, X div 2), X rem Y =:= 0 ].
is_perfect(X) -> lists:sum(divisors(X)) =:= X.
```

Задача 5.2

```
-module(common_members).
-export([common_members/2]).

common_members(L1, L2) ->
    [X || X <- L1, lists:member(X, L2)].
```

Задача 5.3

```
-module(multiplication).
-export([table/0]).

table() ->
    [io:format(''~w␣*␣~w␣=␣~w~n'', [X, Y, X*Y]) ||
     X <- lists:seq(2, 9), Y <- lists:seq(2, 9)].
```

Задача 5.4

```
-module(common2).
-export([common/2]).

common(X, Y) -> [M1 || M1 <- X, M2 <- Y, M1 =:= M2].
```

Задача 5.5

```
-module(queens).
-export([queens/0]).

safe([], _Ver, _Hor, _Cur) ->
    true;
safe([Hor|_T], _Ver, Hor, _Cur) ->
    false;
safe([H|_T], Ver, Hor, Cur) when abs(Hor-H) =:= abs(Ver-Cur) ->
    false;
safe([_H|T], Ver, Hor, Cur) ->
    safe(T, Ver, Hor, Cur-1).

safe(Queens, Ver, Hor) -> safe(Queens, Ver, Hor, Ver-1).
```

```
safeCells(Queens, Ver) ->
    lists:filter(fun(X) -> safe(Queens, Ver, X) end,
                 lists:seq(1, 8)).

queens(Queens, 9) -> Queens;
queens(Queens, Ver) ->
    queens([[Pos|Prev] ||
            Prev <- Queens,
            Pos <- safeCells(Prev, Ver)], Ver+1).

queens() -> queens([[]], 1).
```

Задача 5.6

```
-module(foldrrev).
-export([rev/1]).

rev(L) ->
    lists:foldr(fun(X, Y) -> Y ++ [X] end, [], L).
```

Задача 5.7

```
-module(trees).
-export([tree2list/1]).

-record(node, {value, left=nil, right=nil}).

tree2list(nil) -> [];
tree2list(Node) ->
    tree2list(Node#node.left) ++
        [Node#node.value] ++
        tree2list(Node#node.right).
```

Задача 5.8

Сформулируем принципы перекладывания стопки из N дисков.

Базовый случай в этой задаче — $N = 0$, т.е. ситуация, когда нам ничего перекладывать не нужно. Большее количество дисков перекладывается по следующему алгоритму (стержень, с которого перемещаем диски, будем называть исходным, стержень, на который перемещаем диски — целевым, а третий — вспомогательным):

1. Взять $N - 1$ дисков и переместить их с исходного стержня на вспомогательный.

2. Переместить оставшийся на исходном стержне диск на целевой стержень.

3. Переместить $N-1$ дисков со вспомогательного стержня на целевой.

Первый и третий шаги предполагают перекладывание стопки дисков, то есть рекурсивный вызов процедуры перекладывания. Но так как в каждом следующем вызове N гарантированно уменьшается, мы можем быть уверены, что рано или поздно рекурсия закончится.

```
-module(hanoi).
-export([solve/1]).

solve(0, _From, _To, _Aux) -> [];
solve(N, From, To, Aux) ->
    solve(N-1, From, Aux, To) ++
        [{From, To}] ++
        solve(N-1, Aux, To, From).

solve(N) -> solve(N, 1, 3, 2).
```

Задача 5.9

Сложность с реализацией `dropwhile` в том, что реакция на элемент, для которого переданная функция возвращает `true`, должна отличаться в зависимости от того, встретился ли нам уже элемент, для которого эта функция вернула `false`. Так как мы имеем дело со свёрткой, мы не можем ввести какой-то дополнительный параметр, указывающий, так ли это — ведь тогда свёртка вернёт его вместе с итоговым списком, а это не то, что нам нужно.

Но на самом деле при левой свёртке таким параметром является сам итоговый список: если он пуст, значит, не отвечающий критерию элемент нам ещё не встречался и мы должны проверить текущий элемент с помощью функции-аргумента. Если список не пуст, мы должны безо всяких проверок поместить в него текущий элемент.

```
-module(dropwhile).
-export([dropwhile/2]).

dropwhile(F, L) ->
```

```
    Test = fun(Elem, []) ->
                   case F(Elem) of
                       true -> [];
                       false -> [Elem]
                   end;
               (Elem, Acc) ->
                       Acc ++ [Elem]
           end,
    lists:foldl(Test, [], L).
```

Реализовать `dropwhile` с помощью правой свёртки, строго говоря, невозможно — правая свёртка начинается с конца списка, что не позволяет нам найти первое с начала списка вхождение элемента с её помощью. Но динамическая типизация Эрланга позволяет нам реализовать обходное решение, жульническое по сути (реальная работа делается вне свёртки), но позволяющее «обмануть» функцию `foldr`, подсунув ей функцию, которая формально удовлетворяет условиям задачи:

```
-module(rdropwhile).
-export([dropwhile/2]).

findpos(_F, [], _Pos) ->
    0;
findpos(F, [H|T], Pos) ->
    case F(H) of
        true -> Pos;
        false -> findpos(F, T, Pos+1)
    end.

findpos(F, L) -> findpos(F, L, 1).

dropwhile(F, L) ->
    Pos = findpos(fun(X) -> not F(X) end, L),
    Trim = fun(_Elem, {[], _Cur, 0}) ->
                   [];
               (Elem, {Acc, First, First}) ->
                   [Elem|Acc];
               (Elem, {Acc, Cur, First}) ->
                   {[Elem|Acc], Cur-1, First};
               (_Elem, Acc) ->
                   Acc
           end,
```

```
    lists:foldr(Trim, {[], length(L), Pos}, L).
```

Идея этого решения в следующем: мы заранее находим первый элемент в списке, для которого функция `F` вернёт `false` и запоминаем его позицию (далее будем называть её «линией отреза». После этого мы определяем функцию `Trim`, у которой есть два варианта списка аргументов: либо текущий элемент списка и кортеж вида *{Накопленный хвост списка, Позиция текущего элемента, Линия отреза}*, либо текущий элемент и список.

`Trim` будет добавлять к накопленному хвосту списка элемент за элементом, пока позиция текущего элемента не совпадёт с линией отреза. В этот момент она оставит в аргументе-накопителе только список. Следующие вызовы, обнаружив во втором аргументе список вместо кортежа, просто передадут аргумент-накопитель дальше.

Обратите внимание, что ситуация, когда элемент, для которого `F` возвращает `false`, так и не был найден, требует особой обработки.

Надо ещё раз подчеркнуть, что это решение является хаком в самом плохом смысле этого слова и непереносимо на многие функциональные языки с более строгим контролем типов, вроде Стандартного ML и Haskell (решения, построенные на аналогичной идее, пожалуй, возможны и там, но ценой ещё более уродливых ухищрений).

Задача 5.10

Идея решения этой задачи вот в чём. В список перестановок списка длины 1 входит единственный элемент — он сам. А перестановки списка большей длины можно получить так: по очереди удалять из списка каждый элемент, получать все перестановки оставшихся и в начало каждой из них подставлять удалённый элемент.

```
-module(permutations).
-export([first_n/1]).

permutations(_, [], Accum) -> Accum;
permutations(L, [RH|RT], Accum) ->
    CurPerm = [[RH|X] || X <- permutations(L ++ RT)],
    permutations([RH|L], RT, CurPerm ++ Accum).

permutations([H|[]]) -> [[H]];
permutations(L) -> permutations([], L, []).

first_n(To) -> permutations(lists:seq(1, To)).
```

От функционального программирования к параллельному

Задача 6.1

```
-module(montecarlo).
-export([area_of_circle/0, get_current/1]).

next_step(Inside, Outside) ->
    receive
        {Pid, get_current} -> Pid ! {Inside, Outside};
        _ -> io:format("Malformed message~n")
    after 50 ->
        true
    end,
    % Random numbers from -1.0 to 1.0
    X = (random:uniform() * 2.0) - 1.0,
    Y = (random:uniform() * 2.0) - 1.0,
    if
        (X * X) + (Y * Y) =< 1 ->
            next_step(Inside+1, Outside);
        true ->
            next_step(Inside, Outside+1)
    end.

area_of_circle() ->
    next_step(0, 0).

get_current(Pid) ->
    Pid ! {self(), get_current},
    receive
        {Inside, Outside} ->
            {ok, (Inside / (Inside + Outside)) * 4};
        Other ->
            {error, Other}
    end.
```

Задача 6.2

```
-module(fibmem).
-export([fib/1]).
```

```
memserver(Cache) ->
    receive
        {add, Arg, Res} ->
            NewCache = dict:append(Arg, Res, Cache),
            memserver(NewCache);
        {get, Pid, Arg} ->
            Result = dict:find(Arg, Cache),
            Pid ! Result,
            memserver(Cache)
    end.

add_value(Arg, Res) ->
    memserver ! {add, Arg, Res}.

get_value(Arg) ->
    memserver ! {get, self(), Arg},
    receive
        Res -> Res
    end.

fib(1) ->
    1;
fib(2) ->
    1;
fib(X) ->
    case whereis(memserver) of
        undefined ->
            register(memserver,
                     spawn(fun() ->
                                 memserver(dict:new())
                           end)),
            fib(X);
        _Pid ->
            case get_value(X) of
                {ok, [Res]} -> Res;
                error ->
                    Res = fib(X - 2) + fib(X - 1),
                    add_value(X, Res),
                    Res
            end
    end.
```

Литература

[Абельсон 2010] Абельсон Х., Сассман Дж. Дж. Структура и интерпретация компьютерных программ. М.: «Добросвет», «КДУ», 2010.

[Астапов 2009] Астапов Д. Давно не брал я в руки шашек // Практика функционального программирования, №1, 2009.

[Ахмечет 2006] Ахмечет В. Функциональное программирование для всех. // RSDN Magazine, №2, 2006.

[Бешенов 2010] Бешенов А. Функциональное программирование на Haskell.
Часть 1. Введение.
Часть 2.Основные типы и классы.
Часть 3. Определение функций.
Часть 4. Свертки списков.

[Вознюк 2011] Вознюк А. Продолжения в практике. // Практика функционального программирования, №7, 2011.

[Златопольский 2007] Златопольский Д.М. Программирование: типовые задачи, алгоритмы, методы. М., 2007.

[Ключников 2011] Ключников И. Суперкомпиляция: идеи и методы. // Практика функционального программирования, №7, 2011.

[Липовача 2012] Липовача М. Изучай Haskell во имя добра! М.: «ДМК Пресс», 2012.

[Отт 2009] Отт А. Обзор литературы о функциональном программировании. // Практика функционального программирования, №1, 2009.

[Пирс 2012] Пирс Б., Типы в языках программирования. М.: «Лямбда пресс», «Добросвет», 2012.

[Харпер 1996] Харпер Р. Введение в Стандартный ML. М., 1996.

[Чезарини 2012] Чезарини Ф., Томпсон С. Программирование в Эрланг. М.: «ДМК Пресс», 2012.

[Шень 2004] Шень А. Программирование: теоремы и задачи. М.: МЦНМО, 2004.

[99 problems] 99 Problems на выбранном языке (точно есть для Haskell, Scala, Prolog, OCaml, Common Lisp).

[Armstrong 2007] Armstrong J. Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf, 2007.

[Friedman 1996] Friedman D.P., Fellsen M. The Little Schemer. MIT Press, 1996.

[Friedman 1996,2] Friedman D.P., Fellsen M. The Seasoned Schemer. MIT Press, 1996.

[LYSE] Learn You Some Erlang for Great Good, `http://learnyousomeerlang.com/`.

[Logan 2011] Logan M., Merritt E., Carlsson R. Erlang and OTP in action. Manning Publications, 2011.

[Miller] Evan Miller. The Joy of Erlang; Or, How To Ride A Toruk, `http://www.evanmiller.org/joy-of-erlang.html`.

[Okasaki 1999] Okasaki C. Purely Functional Data Structures. Cambridge University Press, 1999.

[O'Sullivan 2008] O'Sullivan B., Goerzen J., D.B. Stewart. Real World Haskell. Code You Can Believe In. O'Reilly, 2008.

Приложение А.

Краткий справочник по используемым функциям

Ввод-вывод

Здесь приводятся только те функции, которые требуются для той или иной задачи. С полной документацией на упомянутые модули можно ознакомиться на сайте языка: `http://erlang.org/doc/`.

`io:format/1`	Принимает строку с командами формата и выводит её на экран.
`io:format/2`	Принимает строку с командами формата и список подставляемых значений; выводит на экран строку, в которую подставлены значения в соответствии с командами формата.
`io:fread/2`	Принимает подсказку и строку формата; читает со стандартного ввода выражения и пытается привести их к типам, заданным строкой формата; в случае успеха возвращает кортеж из атома `ok` и списка прочитанных значений, в случае ошибки — кортеж из атома `error` и описания ошибки.
`io:read/1`	Принимает подсказку для ввода; читает со стандартного ввода выражение Эрланга и возвращает кортеж из атома `ok` и прочитанного выражения в случае успеха либо кортеж из атома `error` и описания ошибки в случае ошибки.

Математика

`abs/1`	Возвращает модуль числа.
`math:sqrt/1`	Возвращает квадратный корень своего аргумента.
`random:uniform/0`	Возвращает случайное вещественное число в интервале от 0.0 до 1.0.
`random:uniform/1`	Возвращает случайное целое число в интервале от 1 до своего аргумента.

Обработка списков

`length/1`	Принимает список; возвращает длину этого списка.
`lists:dropwhile/2`	Принимает функцию-тест и список; возвращает список, из которого выброшены все элементы до первого, для которого функция-тест вернула `false`.
`lists:filter/2`	Принимает функцию-тест и список; возвращает список, в котором оставлены только те элементы, для которых функция-тест вернула `true`.
`lists:foldl/3`	Левая свёртка: принимает функцию, начальное значение накопителя и список. Функция-аргумент должна принимать два параметра: текущее значение накопителя и текущий элемент списка, а возвращать новое значение накопителя. Применяет функцию-аргумент ко всем элементам списка слева направо (от первого до последнего). Возвращает итоговое значение накопителя.

`lists:foldr/3`	Правая свёртка: принимает функцию, начальное значение накопителя и список. Функция-аргумент должна принимать два параметра: текущее значение накопителя и текущий элемент списка, а возвращать новое значение накопителя. Применяет функцию-аргумент ко всем элементам списка справа налево (от последнего до первого). Возвращает итоговое значение накопителя.
`lists:map/2`	Принимает функцию от одного аргумента и список; возвращает список результатов применения переданной функции к элементам списка.
`lists:max/1`	Принимает список; возвращает самый большой элемент списка.
`lists:member/2`	Принимает значение и список; возвращает `true`, если значение содержится в списке, и `false`, если нет.
`lists:reverse/1`	Принимает список; возвращает список, элементы которого идут в обратном относительно исходного списка порядке.
`lists:seq/2`	Принимает два целых числа; возвращает идущие подряд целые числа от первого до второго аргумента включительно.
`lists:sort/1`	Принимает список; возвращает список с элементами исходного списка, отсортированными по возрастанию.
`lists:sort/2`	Принимает функцию сравнения и список; возвращает список, отсортированный по возрастанию в соответствии с функцией сравнения. Функция сравнения должна принимать два аргумента и возвращать `true`, если первый аргумент меньше второго или равен ему, и `false` в противном случае.
`lists:sum/1`	Принимает список чисел; возвращает сумму его элементов.

Структуры данных

`dict:new/0`	Возвращает пустой словарь.
`dict:append/3`	Принимает ключ, значение и словарь; возвращает новый словарь, в котором к значениям по переданному ключу добавилось переданное значение.
`dict:find/2`	Принимает ключ и словарь. Если в словаре содержатся значения по этому ключу, возвращает кортеж `{ok, Values}`, где `Values` — список найденных значений; в противном случае возвращает атом `error`.

Служебные функции Эрланга

`register/2`	Принимает атом и идентификатор процесса; если атом ещё не используется в качестве имени процесса, регистрирует процесс под этим именем и возвращает `true`.
`self/0`	Возвращает идентификатор процесса, из которого вызвана.
`spawn/1`	Принимает функцию; запускает её в отдельном процессе и возвращает идентификатор этого процесса.
`timer:tc/3`	Принимает имя модуля, имя функции и список аргументов; возвращает время в микросекундах, затраченное на вычисление переданной функции от переданных аргументов.

Приложение В.

Команды формата

Вывод

`s`	Вывести строку.
`w`	Вывести выражение Эрланга.
`p`	Вывести выражение Эрланга с аккуратными переносами длинных строк и отступами. По умолчанию считает строку равной 80 символам. Принимает аргумент, задающий длину строки: так, команда формата «`~62p`» будет считать длину строки равной 62 символам.
`B`	Вывести число в заданной системе счисления с основанием от 2 до 36, по умолчанию используется основание 10.
`n`	Перейти на новую строку.

Ввод

`d`	Прочитать десятичное целое.
`f`	Прочитать вещественное число.
`s`	Прочитать строку. Принимает числовой аргумент — ограничение длины строки. Также принимает аргумент `t`, включающий поддержку символов Unicode (нужно, если вы собираетесь вводить, например, русские буквы).

Здесь приведены не все команды формата. Полное описание можно прочитать в документации модуля `io` (`http://erlang.org/doc/man/io.html`).

Предметный указатель