Введение в язык Окамль

1.1 Структура программы

Поскольку мы только учимся, давайте несколько упрощённо скажем, что программа на языке Окамль — это последовательность выражений и описаний, указываемых через двойное двоеточие (;;). Или то же самое зададим грамматикой:

$$\langle \textit{программа} \rangle \quad ::= \quad (\langle \textit{описание} \rangle \mid \langle \textit{выражение} \rangle)\ \texttt{;;}\ [\langle \textit{программа} \rangle]$$

1.1.1 Выражение

Выражения как идея вам должны быть хорошо знакомы. Например, вы наверняка умеете строить арифметические выражения с помощью арифметических действий. Выражение 2+2 имеет в Окамле такой же смысл, как и в арифметике (вычисление суммы двух чисел).

Однако, действия (и, соответственно, выражения, которые из них составляются) бывают не только арифметические. Рассмотрим такую программу:

```
print_string "Здравствуй, мир!";;
```

Здесь выражение содержит действие `print_string` — печати строки на экране. При запуске эта программа печатает текст `Здравствуй, мир!`, после чего заканчивает работу.

Давайте напишем что-нибудь еще:

```
print_string "Здравствуй, ";;
print_string "мир!";;
```

Эта программа делает в точности то же самое, но устроена сложнее: в ней два выражения, каждое из которых печатает свой кусок текста.

В целом, возможные действия в Окамле очень разнообразны.

1.1.2 Описание

Идея описаний для вас тоже не должна быть новой. Описания подобны словам «обозначим за v скорость пешехода», часто встречающимся в решениях задач по физике и математике, они позволяют давать выражениям имена. Зачем делают описания? Чтобы разбить выражения на части, упростить их запись, сделать их понятными.

Если вы хотите дать какому-нибудь выражению имя (иначе называемое как *идентификатор*), делайте это в соответствии с грамматикой:

$$\langle \textit{описание} \rangle \quad ::= \quad \texttt{let}\ \langle \textit{идентификатор} \rangle\ \texttt{=}\ \langle \textit{выражение} \rangle$$

Рассмотрим, например, такую программу:

```
let str = "Здравствуй, мир!";;
print_string str;;
```

В данной программе в первой строке создается константа `str`, содержащая строчку `"Здравствуй мир!"` (иными словами, это описание константы `str`), которая затем печатается в выражении во второй строке.

Раз это константа, то ее можно использовать сколько угодно раз:

```
let str = "Здравствуй, мир!";;
print_string str;;
print_string str;;
```

Данная программа будет печатать строчку два раза.

Константу можно переопределить:

```
let str = "Здравствуй, мир!";;
print_string str;;
let str = "Привет, мир!";;
print_string str;;
```

При этом старое значение константы теряется. Данная программа напечатает следующий текст:

```
Здравствуй, мир!Привет, мир!
```

Идентификаторы констант строятся по следующим приавилам:

$$\begin{aligned} <\textit{идентификатор}> &::= <\textit{начальный символ}>\{<\textit{символ}>\}* \\ <\textit{начальный символ}> &::= \texttt{a..z} \mid \texttt{\_} \\ <\textit{символ}> &::= \texttt{a..z} \mid \texttt{A..Z} \mid \texttt{0..9} \mid \texttt{\_} \mid \texttt{'} \end{aligned}$$

Таким образом, идентификаторы `fAAA9_'` и `___` вполне допустимы, тогда как `0ffa`, `'abcd`, `ABCD` не являются допустимыми идентификаторами констант.

1.1.3 Пробелы, переводы строки, комментарии

Для компилятора Окамля существенно, чтобы два идентификатора отделялись всегда пробелами. Также, не смотря на то, что синтаксис языка не требует указания пробела между знаками операций и идентификаторами (скажем, мы можем написать `let x="мир";;`, мы можем их разделить их пробелом, если это нам кажется правильным: `let x = "мир";;` Везде же, где может/должен находиться хотя бы один пробел, вместе с ним (или вместо него) может также находиться произвольное количество пробелов и/или переводов строк, а также *комментариев*.

Комментарий — это произвольный текст, ограниченный символами `(*` и `*)`, который не анализируется компилятором. В комментариях обычно пишут какие-то соображения на естественном языке, комментирующие код программы.

Код программы с комментариями может выглядеть так:

```
print_string    "Здравствуй, ";; (* Здравствуй, мир! - это перевод с
                                    английского классического текста *)
print\_string                    (* hello, world *)
   "мир!"
;;
```

Ещё комментарии используются для временного исключения части кода. Перед тем, как удалить код из программы, бывает, стоит его «закомментировать». Вдруг передумаем удалять?

1.2 Элементарные типы данных: числа и строки

Из школьной программы у вас должно уже сложиться интуитивное понятие типа значения: скажем, скорость измеряется в километрах в час, а масса — в килограммах. При этом ясно, что выражение «1 км/ч» + «5 кг» в общем случае бессмысленно и никакого естественного способа его вычислить мы не знаем.

В программировании это интуитивное понятие встречается очень часто, поэтому, чтобы двигаться дальше, нам потребуется его формализовать. Данная формализация несовершенна, но подходит для целей знакомства с языком Окамль.

Определение 1.2.1. *Типом данных* называется множество значений.

Например, целочисленный тип данных в языке Окамль (при работе на 32-разрядном компьютере) составляют все целые числа в диапазоне от -2^{30} до $2^{30}-1$.

Тип может иметь имя, для целых чисел в Окамле используется имя `int`, для чисел с плавающей точкой — `float`, для строчек — `string`.

Литералом некоторого значения мы будем называть его запись в программе. Например, литералом для строчки `Здравствуй, мир!` будет `"Здравствуй, мир!"`.

1.2.1 Строки

Рассмотрим строковые литералы подробнее:

$$\begin{aligned} \textit{<строковый-литерал>} &::= \texttt{"}\textit{<символ>}*\texttt{"} \\ \textit{<символ>} &::= \textit{<печатный символ>} \mid \texttt{\}\textit{<спецсимвол>} \\ \textit{<спецсимвол>} &::= \texttt{"} \mid \texttt{\} \mid \texttt{n} \end{aligned}$$

Здесь *<печатный символ>* — это любой символ, кроме двойных кавычек (") и обратной косой черты (\).

Зачем нужны спецсимволы? Например, чтобы задавать двойные кавычки как часть текста. Рассмотрим текст `""^""`: следует ли его понимать как соединение двух пустых строк `("")^("")` или его следует понимать как литерал строки `"^"`? Иными словами, вторые и третьи кавычки — это символы строки или ограничители строки?

Чтобы избежать путаницы, в Окамле принято считать, что двойные кавычки всегда ограничивают строку, а для указания кавычек как символа используется последовательность из двух символов: обратной косой черты и кавычек (\"). Литерал `"\"\\"` задает строчку из кавычки, за которой идет обратная косая черта, а литерал `"\\\"\\\` задает строчку `\"\"`.

Но помимо обычных символов, существуют и другие символы, которые нужно иметь возможность вставлять в строки, но которым не соответствует никакого изображения. Нам пока потребуется только один из них — символ перевода (окончания) строки. Всё,

что печатается после него, будет печататься на новой строке. Для его задания служит последовательность `\n`.

Так, программа

```
print_string "Здравствуй,\nмир!";;
```

напечатает 2 строки:

```
Здравствуй,
мир!
```

1.2.2 Целые числа

Литералы целых чисел задаются следующей грамматикой:

$$\textit{<литерал целого числа>} \;::=\; \{0..9\}^{+}$$

Пример. Следующие строчки являются литералами целых чисел:

```
2419  71000  001  0
```

1.2.3 Плавающие числа

Плавающие числа (или, более формально, *числа с плавающей запятой*) предназначены для представления дробных и вещественных чисел. К сожалению, компьютер не может представить многие вещественные числа точно, только приближённо, отсюда и иное название.

Упрощённый синтаксис литералов чисел с плавающей запятой:

$$\begin{aligned}\textit{<литерал плавающего числа>} \;::=\;& \{0..9\}^{+}.\{0..9\}^{+}\\ \mid\;& \{0..9\}^{+}.\\ \mid\;& .\{0..9\}^{+}\end{aligned}$$

Пример. Примеры литералов плавающих чисел:

```
00024.19  71.000  0.17  147.  .1717
```

Как видите, от обычной записи десятичной дроби отличий два:

- Вместо традиционной запятой в числе используется точка.
- Помимо основной формы есть сокращенные, когда ноль в целой или дробной части может быть опущен: числа `0.0`, `0.` и `.0` — это одно и то же число.

1.2.4 Арифметические выражения

Как вы, возможно, обратили внимание, литералы для значений каждого типа существенно различаются. Если значение в кавычках — это строка, если оно состоит только из цифр — это целое число, если оно содержит точку — это плавающее число. Такое разделение создано намеренно: никакое значение одного элементарного типа не может принадлежать какому-то другому элементарному типу.

Арифметические функции для работы с целыми и плавающими числами также имеют разные имена. Нельзя пользоваться целочисленной операцией для работы с плавающими числами и наоборот, плавающей операцией для работы с целыми. В таблице приводятся обозначения для четырех основных арифметических операций и примеры работы с ними.

Действие	Целые	Плавающие	Примеры	
Сложение	`+`	`+.`	`14+12`	`14.+.12.`
Вычитание	`-`	`-.`	`41-20`	`.23-.71.`
Умножение	`*`	`*.`	`24*90`	`78.42*.84.0`
Деление	`/`	`/.`	`64/16`	`0.14/.9881.`

Пример. Такие выражения содержат ошибки:

```
24+42.071  5.0*.5  20/.20
```

А такие выражения допустимы:

```
24.+.42.071  5*5  20/20
```

Точка у арифметических операций для плавающих чисел — это часть их имени, как и точка в обозначении соответствующих литералов. Если вы описываете константу, точки в ее имени *не* ставятся. Например, в коде ниже три константы для плавающих чисел: `pi`, `diameter`, `area`, ни при одной из них не указывается точка:

```
let pi = 3.1415926;;
let diameter = 4.0;; (* 4 метра *)
let area = pi *. diameter *. diameter /. 4.;; (* вычисляем площадь *)
```

1.3 Условное выражение

Вернемся к одной из задач: будем решать линейное уравнение вида $a \cdot x + b = c$. Нетрудно видеть, что это уравнение почти всегда имеет ровно одно решение, кроме случая, когда $a = 0$. В этом случае, в зависимости от констант b и c, уравнение либо имеет бесконечно много решений, либо ни одного.

Надо как-то выделить этот случай, поскольку применение классической формулы $x = (c - b)/a$ даст деление на 0, что приведет к ошибке выполнения программы. В этом нам поможет условное выражение.

1.3.1 Булевский тип

Для начала мы введем новый тип данных — *булевский (логический, истинностный)* — имеющий два значения `false` и `true` (*ложь* и *истина* соответственно). Данный тип в Окамле имеет идентификатор `bool`.

Тип назван так в честь Джорджа Буля — английского математика, предложившего использовать алгебраические методы в логических рассуждениях. Это ему мы в значительной степени обязаны идеей представлять ложь как 0, а истину как 1.

Для работы с булевскими типами определены несколько операций:

Операция	Обозначение	Пример
Логическое «И»	`&&`	`true && false`
Логическое «Или»	`||`	`false || false`
Отрицание	`not`	`not true`

1.3.2 Условное выражение

Условное выражение — это выражение, задаваемое следующей грамматикой:

<*условное выражение*> ::= `if` <*условие*> `then` <*ветка then*> [`else` <*ветка else*>]
<*условие*> ::= <*выражение*>
<*ветка then*> ::= <*выражение*>
<*ветка else*> ::= <*выражение*>

В том случае, когда ветка else не указана, предполагается что это сокращение для записи

`if`<*условие*>`then`<*ветка then*>`else ()`

Результат вычисления условного выражения зависит от условия. Если результат вычисления условия — `true`, то результат всего выражения — это результат вычисления ветки then, иначе — результат вычисления ветки else.

Например, результатом выражения ниже будет 3:

```
if 2 > 1 then 3 else 44
```

1.3.3 Операции сравнения

Определим сравнения в Окамле подробнее:

Операция	Обозначение	Пример
равно	`=`	`1 = 1`
не равно	`<>`	`"строка" <> "еще одна строка"`
больше	`>`	`2 > 1`
меньше	`<`	`1.0 < 2.0`
больше или равно	`>=`	`1 >= 1`
меньше или равно	`<=`	`false <= true`

Все приведенные выше примеры выдают значение `true`. Напротив, выражения

```
1=0  2<>2  "я"<"a"  2.0>2.0  1>=2  true<=false
```

все имеют значение `false`.

Обратите внимание, что операции сравнения, в отличие от арифметических, берут аргументы произвольного типа. Единственным условием является то, что тип левого аргумента и тип правого аргумента должен быть одинаков. Такие операции называют *полиморфными*.

1.3.4 Пример

Итак, напишем программу, решающую линейное уравнение:

```
print_string "x:\n";;
if a <> 0. then
  print_float ((c -. b) /. a)
else
  print_string "либо ни одного, либо бесконечно много";;
```

Но ответ «либо ни одного, либо бесконечно много» слишком неопределенный, ведь мы можем ответить на вопрос точнее. Добавим второе условное выражение, разделяющее случаи $b = c$ и $b \neq c$.

```
print_string "x:\n";;
if a <> 0. then
  print_float ((c -. b) /. a)
else (if b = c then
  print_string "любое"
else
  print_string "решений нет");;
```

Обратите внимание на скобки вокруг второго условного выражения. С условными выражениями можно обращаться так же, как и с арифметическими, в том числе заключать в скобки. Впрочем, в данном конкретном случае мы можем написать это же чуть красивее:

```
print_string "x:\n";;
if a <> 0. then
  print_float ((c -. b) /. a)
else if b = c then
  print_string "любое"
else
  print_string "решений нет";;
```

1.4 Функции

Мы уже довольно давно знакомы с функциями (напомним, первая программа состояла из вызова функции `print_string`), но только со стороны пользователя. Сейчас мы научимся функции описывать.

Существует математические определения функции, как отображения, ставящего в соответствие каждому элементу одного множества (называемого множеством отправления), ровно один элемент другого множества (называемого множеством прибытия).

Это определение сохраняет некоторый смысл и в случае функциональных языков программирования - скажем, функция `sqrt`, вычисляющая квадратный корень, действительно очень близка к математическому определению; такая функция называется *чистой*. Но существует и отличие. В языках программирования возникает понятие *побочного эффекта* — любое действие может, помимо вычисления требуемого значения, как-то повлиять на остальной мир.

Например, как вы думаете, функция `format_hard_disk_drive`, берущая в качестве аргумента целое число и возвращающая строчку `"винчестер отформатирован"` — действительно ли это только функция из множества целых чисел в множество строк?

Впрочем и другие, менее разрушительные функции, также могут иметь побочный эффект. Например, функция печати строки на экране - ценность ее только в побочном эффекте. Поэтому мы будем придерживаться другого, более утилитарного, взгляда. Функция — это просто фрагмент кода программы. Функция может быть *вызвана*, то есть ее код может быть исполнен.

1.4.1 Определение функций

Функцию можно *определить* с помощью следующей конструкции:

<определение функции> ::= `let` [`rec`] *<идентификатор>параметры* = *<выражение>*`;;`
<параметры> ::= *<идентификатор>*$^+$ | `()`

Отличие определения функции от определения константы — наличие формальных параметров, указываемых через пробел после идентификатора функции перед знаком равенства. Нетерминал *<идентификатор>* — это тот же нетерминал, что и в определении констант.

Пример. `let average a b = (a+b)/2;;`

Эта функция берет два аргумента и возвращает их полусумму. Константы `a` и `b` называются формальными параметрами — формальными потому, что при вызове функции вместо них подставляются фактические параметры:

```
print_int (average 3 9);;
```

Вычисление значения выражения мы можем представить так: вместо букв `a` и `b` в описании функции подставляются числа 3 и 9:

```
let average 3 9 = (3+9)/2;;
```

Выражение справа от равенства (тело) функции вычисляется и результат (6) подставляется в место вызова функции:

```
print_int (6);;
```

В тех случаях, когда функция не имеет параметров, мы должны по правилам языка Окамль указать какой-нибудь фиктивный параметр — иначе функция превратится в константу. Обычно такое нужно из-за побочного эффекта функции, поскольку чистая функция, не зависящая от аргументов — это и есть константа.

Существует специальный тип, который используется для указания фиктивного значения там, где оно нужно - это тип `unit`. Он имеет ровно одно значение, записываемое как `()`.

Для примера покажем, как описать функцию, печатающую приветствие:

```
let print_hello () = print_string "Привет!";;
```

Здесь значение `()` исполняет роль формального формального параметра. Вызов выглядит уже знакомо:

```
print_hello ();;
```

Здесь значение `()` исполняет роль формального фактического параметра.

1.4.2 Возвращать или печатать?

Существенным вопросом, касающимся функций, является вопрос о результате их работы. Предположим, у нас стоит задача написать функцию, вычисляющую площадь треугольника.

У нас есть 2 варианта:

```
let triangle_area1 a h = a *. h /. 2.;;
let triangle_arge2 a h = print_float (a *. h /. 2.);;
```

Первый вариант функции *возвращает* значение площади. Функция `triangle_area1` берет 2 аргумента с плавающей точкой и возвращает плавающий результат. Второй же вариант печатает площадь треугольника на экране — а в вызвавшее функцию выражение возвращает значение типа `unit`.

Не смотря на то, что разница между этими вариантами может казаться незначительной — площадь же все равно вычисляется — тем не менее, в чуть более сложных задачах этот вопрос может стать принципиальным.

Дело в том, что значение функции может быть использовано в дальнейших вычислениях, и тем самым появляется много дополнительных случаев для ее применения. Функция, печатающая вычисленное значение на экране, такой возможности лишена. Поэтому везде в задачах, где явно не указано иное, предполагается, что от вас требуется написание функции, к которой при необходимости прилагается отдельный код, печатающий ее результат:

```
let a = read_float ();;
let h = read_float ();;
let triangle_area a h = a *. h /. 2;;
print\_float (triangle_area a h);;
```

1.5 Алгебраические типы данных

Представим, что мы пишем функцию для вычисления корня линейного уравнения

$$a \cdot x + b = 0$$

Если $a \neq 0$, то результат функции — плавающее число $-\frac{b}{a}$. Однако, функция может вернуть *не только* плавающее число. Например, если $a = 0$ и $b = 0$, то решением уравнения будет любое число, и никакое конкретное число не будет являться полным ответом. Также, если $a = 0$ и $b \neq 0$, то решений у уравнения нет вообще.

Таким образом, функция может вернуть один из трёх различных вариантов ответа: «один корень» (в этом случае мы уточняем ответ, указывая этот корень), «любое плавающее число — корень» и «корни отсутствуют».

Для таких ситуаций — а также для многих других, когда у значения есть несколько вариантов, — в языке Окамль предусмотрены *алгебраические типы*, которым и посвящён этот раздел.

1.5.1 Определение алгебраического типа данных

Давайте определим пользовательский тип данных, способный хранить корни линейного уравнения.

Мы уже поняли, что корни уравнения — это один из трёх вариантов: «один корень» (с указанием этого корня), «любое плавающее число — корень» и «корни отсутствуют». В Окамле необходимо называть варианты одним словом, начинающимся с большой буквы:

пусть они называются `One`, `Any` и `None`. Названия были выбраны произвольно, на основании английских слов, значащих «один», «любой» и «никакой» соответственно — мы могли выбрать любые другие. А теперь составим определение типа:

```
type roots = One of float | Any | None;;
```

Тут задано четыре имени: имя типа (`roots`) и три имени *конструкторов типа* — вариантов значений (`One`, `Any`, `None`).

Пара слов об имени типа. Каждый тип имеет своё имя, мы уже знакомы с некоторыми (`int`, `string` и т.п.), и алгебраических типов нам тоже, весьма вероятно, потребуется не один. Чтобы типы различать, им нужно давать уникальные имена.

Обратите внимание на первый вариант: мы потребовали, чтобы с вариантом `One` указывалось ещё и число типа `float`.

Несколько примеров значений:

```
let a = One 1.4;; (* Единственный корень, равный 1.4 *)
let b = Any;;     (* Корень - любое плавающее число *)
let c = None;;    (* Корни отсутствуют *)
```

Теперь мы можем даже привести функцию, вычисляющую решение линейного уравнения, и возвращающую его решение через значение алгебраического типа roots.

```
type roots = One of float | Any | None;;

let solve a b c =
  if a <> 0. then One ((c -. b) /. a)
  else if b = c then Any
  else None;;
```

1.5.2 Сопоставление с образцом

Теперь мы умеем создавать значения алгебраических типов, надо научиться их использовать. Например, печатать.

С вариантами без параметров можно справиться и с имеющимися конструкциями, с использованием условного оператора:

```
if x = Any then print_string "Любое число"
```

Но как напечатать корень уравнения, который удалось найти, в случае `One`?

Для этого существует сопоставление с образцом.

Образец — это одно из следующего:

1. литерал, например, `23`, `932.4`, `()`, `true`, `"math"`;
2. имя переменной, например, `x`, `v923`, `_`;
3. кортеж (упорядоченная n-ка) других образцов: $(p_1, p_2, \ldots, p_n)$ — образец, если $p_1, \ldots p_n$ — образцы; например, `(23, x)`, `(23, (x, 17))`;
4. конструктор алгебраического типа без параметра, например `None` или `Any`;
5. конструктор алгебраического типа с параметром: Kp — образец, если K — конструктор, а p — образец; например, `One x`.

Сопоставление с образцом в Окамле используется в разных конструкциях, посмотрим, как это работает в `match`.

1.5.3 Конструкция match

Рассмотрим, скажем, следующий пример:

```
match s with
    0 -> print_string "Ноль"
  | x -> print_string "Не ноль, "; print_int x;;
```

В конструкции match слева от стрелки пишут образцы, а справа — соответствующие им выражения. Первый образец (0) сопоставится только, если $s = 0$. В остальных случаях сопоставится второй образец (x), при этом, как и в конструкции let, переменная x получит нужное для сопоставления значение.

Теперь мы можем легко написать функцию, печатающую значение типа roots:

```
let print_roots v =
  match v with
    Any    -> print_string "Любое число"
  | None   -> print_string "Корней нет"
  | One f  -> print_float f;;
```

Существует специальный идентификатор для констант, значения которых нас не интересуют — символ подчеркивания (_). При сопоставлении это — обычная константа и, например, сопоставление с образцом `One _` происходит так же, как и с образцом `One f`. Но в случае удачного сопоставления значение константе s_ не присваивается. Наиболее типичный пример использования этой константы выглядит так:

```
match v with
  One f -> print_float f               (* печатаем корень *)
| _ -> print_string "Одного корня нет";; (* а здесь все остальные,
неинтересные нам случаи *)
```

В приведенном выше примере мы печатали результат и значение функции нас не интересовало. Но все может быть и иначе:

```
let finite_roots v = (* функция возвращает true, если ее аргумент -
не более чем конечное количество корней *)
  match v with
    One _ -> true
  | None  -> true
  | Any   -> false;;
```

1.6 Списки

1.6.1 Определение

Слово список всем хорошо знакомо: список дел, список класса, список оценок. В программировании так называют структуру данных, позволяющую хранить упорядоченный набор значений, при этом важно, что мы умеем быстро добавлять элемент в начало списка и быстро убирать начальный элемент.

Определение, данное выше, весьма размытое, но с помощью Окамля мы можем его формализовать.

Списком назовём алгебраический тип, заданный следующим образом:

```
type 'a list = Nil | Cons of 'a * ('a list)
```

Напомним, что значит эта запись: список есть либо константа `Nil`, либо конструктор типа `Cons`, имеющий аргументом пару из значения и списка.

Поясним, почему `Cons (3, Cons (5, Nil))` — список. Начнём с `Nil`, который является списком по первому правилу из определения. Раз так, значит, `Cons (5, Nil)` — тоже список, поскольку у `Cons` аргументом как раз и является пара из значения (`5`) и списка (`Nil`). А раз `Cons (5, Nil)` — список, то и всё выражение — тоже список.

Поскольку списки — одни из основных конструкций Окамля, то для них предопределён особый синтаксис: вместо `Nil` используются квадратные скобки (`[]`), а вместо `Cons` — двойное двоеточие (`::`). Таким образом, `Cons (3, Cons (5, Nil))` записывается как `3 :: (5 :: [])`.

Также, список можно задавать явным перечислением значений в квадратных скобках через точку с запятой: `[3;5]`.

Ещё определим несколько полезных слов. *Пустой* список — список, состоящий только из `[]`, т.е. список, не содержащий никаких значений. В противоположность ему, любой непустой список a_1`::`$(a_2$`::`$\ldots$`::`$(a_n$`::[]`$))$ можно разбить на *голову* и *хвост*: голова — это a_1, а хвост — это остающийся после удаления головы список a_2`::`$\ldots$`::`$(a_n$`::[]`$)$. Или иначе: если рассмотреть конструктор типа `::`, то левый его аргумент — это голова, а правый — хвост (*голова* `::` *хвост*).

1.6.2 Доступ к элементам списка

Как мы уже проходили, доступ к элементам алгебраического типа в первую очередь осуществляется с помощью сопоставления с образцом и конструкции `match`, и списки — не исключение.

Классический, самый прямой, вариант применения этой конструкции покажет функция вычисления длины списка:

```
let rec length l = match l with
    [] -> 0
  | h::t -> 1 + length t;;
```

Два конструктора в определении списка — значит, два варианта в `match`. Первый случай — когда список пуст, второй — когда список собран из некоторых головы и . Самой по себе головой мы не интересуемся, ведь, чтобы посчитать количество элементов в списке, вникать в сами эти элементы не обязательно. Поэтому значение `h` не указано справа от стрелки, только слева — а справа мы всегда прибавляем 1 к длине хвоста.

В самом деле: пусть есть непустой список `l`. Раз он непустой, то он образован конструктором «двойное двоеточие» из какого-то элемента `h` и какого-то ещё списка `t`: `l = h::t`. Понятно, что длина `t` на один элемент меньше, чем длина всего списка.

1.6.3 Пример: красивая печать списка

Всё, что дальше будет излагаться про списки, ничего принципиально нового не содержит: это будут примеры на использование уже рассказанных выше конструкций. Однако, данные примеры могут быть поучительны.

В этом подразделе рассмотрим задачу печати список целых чисел. Мы можем сделать это в целом следуя идее, использованной при вычислении длины.

```
let rec print_int_list l = match l with
    [] -> ()
  | h::t -> print_int h; print_int_list t;;
```

Отличия от функции `length` здесь только в том, что мы не возвращаем длину списка, а печатаем его — возвращаем же мы `()`. Напомним, что данное значение принадлежит типу `unit` и используется там, где формально указать значение нужно, а фактически написать нам нечего — ведь весь смысл данной функции напечатать текст на экране; новых значений с её помощью мы не создаём.

Итак — в случае пустого списка мы возвращаем «ничего» — то есть `()`, а в случае наличия головы и хвоста печатаем голову, после чего рекурсивно вызываем печать для хвоста.

Однако, вызов `print_int_list [1;2;3]` выдаст нам `123` — напечатает числа без пробелов. Это можно исправить, добавив разделителей между числами, например, так:

```
let rec print_int_list l = match l with
    [] -> ()
  | h::t -> print_int h; print_string ","; print_int_list t;;
```

Однако, результат не станет идеальным: мы получим `1,2,3,`. Каша из цифр исчезла, но появилась заключительная запятая, которая будет сбивать с толку. Причина её появления очевидна: мы *всегда* печатаем запятую после числа, однако, в красивой записи запятая разделяет два числа, и за последним числом не нужна. Поэтому нам нужно две версии печати числа: с запятой в конце или без неё.

Идея, как мы могли бы отличить эти два случая, проста — у последнего двойного двоеточия списка пустой хвост: $a_1 :: (a_2 :: \ldots :: (a_n :: []))$. Иными словами, когда последнее число списка a_n окажется в переменной `h` образца, в переменной `t` окажется пустой список. Это наблюдение приводит к следующей программе:

```
let rec print_int_list l = match l with
    [] -> ()
  | h::[] -> print_int h
  | h::t -> print_int h; print_string ","; print_int_list t;;
```

Обратите внимание, что мы вставили вариант печати последнего элемента списка перед общим случаем. Конструкция `match` перебирает варианты строго сверху вниз, и выбирает первый подходящий образец. Если бы мы поставили обработку последнего элемента ниже общего случая, соответствующий код никогда бы не вызывался.

Последний штрих: данная функция печатает только целочисленные списки. А что делать, если нам нужно печатать списки других типов, например, состоящие из строк? Окамль в общем случае не знает, как печатать значение произовольного типа. Для этой цели давайте функции печати списка передавать функцию печати элемента:

```
let rec print_list f l = match l with
    [] -> ()
  | h::[] -> f h
  | h::t -> f h; print_string ","; print_list f t;;
```

Вызывать теперь её нужно, правда, чуть хитрее:

```
print_list print_int [1;2;3]
```

Функция `print_list` теперь берёт на вход два параметра, и первый — это функция `print_int`, которая печатает элементы. Если нам в другой ситуации потребуется печать списка строк, мы без труда сделаем это: `print_list print_string ["a";"b";"c"]`.

1.6.4 Пример: соединение двух списков

Определим функцию `append`, соединяющую два списка — скажем, чтобы из списков `[1;2]` и `[3;4]` получался бы список `[1;2;3;4]`. Главная сложность здесь — увидеть, как организовать рекурсию.

Возьмём первый список и разберём возможные случаи. Если первый список пуст, результатом функции, очевидно, будет второй: добавление пустого списка не изменяет результат. Если же первый список содержит голову a_1 и хвост $[a_2; \ldots; a_n]$, мы можем рекурсивно соединить хвост со вторым списком, а затем добавить голову к результату:

$$append[a_1; a_2; \ldots a_n][b_1; b_2; \ldots b_n] = append(a_1 :: [a_2; \ldots a_n])[b_1; \ldots; b_n] =$$

$$a_1 :: append[a_2; \ldots a_n][b_1; b_2; \ldots b_n]$$

Ведь всё равно элемент a_1 должен быть первым в итоговом списке — поэтому мы его можем сразу от первого списка отцепить, и прицепить уже к результату рекурсивного вызова.

Это даёт нам следующую программу:

```
let rec append a b = match a with
    [] -> b
  | h::t -> h::(append t b)
```

Поскольку данная операция очень часто нужна при работе со списками, она встроена в язык и у неё специальное имя — символ «коммерческое „эт“» (`@`), он же неформально называется «собака». Соответственно, следующее странное на первый взгляд равенство в Окамле вполне осмысленно: `[1;2] @ [3;4] = [1;2;3;4]`

1.6.5 Пример: разворот списка

И следующий пример, который мы разберём — функция разворота, делающая из списка $[a_1; a_2; \ldots; a_n]$ список $[a_n; a_{n-1}; \ldots; a_2; a_1]$.

Прямолинейное решение, `match` по аргументу, вполне сработает. Для пустого списка его разворот также пуст, для непустого — развернём хвост, а голову списка присоединим к концу:

```
let rec reverse l = match l with
    [] -> []
  | h::t -> reverse t @ [h]
```

Сразу обращаем внимание, что код `h::t -> reverse t :: h` был бы некорректен. Напомним, что двойное двоеточие *всегда* требует слева элемент (голову), справа список (хвост). В данном же случае получается, что слева указан хвост, а справа — голова. Это неизбежно приведёт к ошибке компиляции.

Также, возможно, кого-то озадачит код `[h]` — но это всего лишь список, содержащий `h` в качестве единственного своего элемента. Для Окамля нет никакой разницы между записями `[5]`, `[1+1*4]` или даже `let x = 5 in [x]`, внутри квадратных скобок может стоять любое выражение.

Однако, можно предложить и другое решение:

```
let rec revh l a = match l with
    [] -> a
  | h::t -> revh t (h::a);;

let reverse l = revh l [];;
```

Проследим за тем, как данный код развернёт трёхэлементный список `[2;3;4]` (мы напишем только цепочку равенств, более подробный анализ исполнения кода оставим читателю):

```
revh [2;3;4] [] = revh [3;4] [2] = revh [4] [3;2] = revh [] [4;3;2] = [4;3;2]
```

Функция `rehv` на каждом рекурсивном вызове перекидывает по элементу из исходного списка в итоговый, пока не развернёт его целиком.

1.7 Кортежи (упорядоченные n-ки)

Бывают ситуации, когда нужно запомнить (передать) два числа одновременно.

Давайте напишем функцию, которая возвращает все целые числа, отличающиеся от данного целого числа на 1. Скажем, для 4 функция должна вернуть два числа: 3 и 5. Аналогично, для -18 — числа -17 и -19.

Содержательно задача тривиальная (по числу x вернуть $x+1$ и $x-1$), но пока что мы писали функции, возвращающие не более одного числа. Мы можем написать две функции: одна будет прибавлять 1, а другая — вычитать, но есть другой способ. , называемый кортежем или упорядоченной n-кой.

Сперва научимся создавать значения этого типа. Для этого нужно перечислить значения, которые вы хотите положить в кортеж, через запятую и окружить скобками:

Примеры:

```
let a = (1,2);;
let b = (1,3,"asdf",9.);;
let c = ((1,2),(3,4));;
```

Заметим, что константа c хранит кортеж из двух других кортежей. Это, естественно, вполне допустимо.

Кортежи также можно указывать в сопоставлении с образцом:

```
type roots = One of float | Any | None

let roots_info t =
  match t with
    (Any, _) -> "бесконечное количество корней"
  | (_, Any) -> "бесконечное количество корней"
  | _        -> "конечное количество корней";;

print_string (roots_info (One 1., Any));;
```

И, конечно, кортежи можно указывать как параметры в алгебраических типах:

```
type roots2 = Two of float*float | One of float | Any | None
```

1.7.1 Формальная грамматика

$$
\begin{aligned}
\textit{<сигнатура-кортежа>} &::= \textit{<сигнатура-типа>}\{\texttt{*}\textit{<сигнатура-типа>}\}^{+} \\
\textit{<литерал-кортежа>} &::= \texttt{(}\textit{<выражение>}\{\texttt{,}\textit{<выражение>}\}^{+}\texttt{)} \\
\textit{<образец-кортежа>} &::= \texttt{(}\textit{<образец>}\{\texttt{,}\textit{<образец>}\}^{+}\texttt{)}
\end{aligned}
$$

1.8 Рекурсия

Что можно сказать про такой тип? type a = S of a | Nil;;

Отличие его от обычного алгебраического типа - использование своего имени внутри описания. Выглядит, возможно, довольно непривычно для вас, но если будем действовать по порядку, все получится. Воспользуемся нашим упрощенным определением (тип - это множество значений), и попробуем найти значения, которые этому типу соответствуют.

- Давайте сперва посмотрим на конструктор `Nil`. У `Nil` нет параметров, поэтому значение `Nil`, очевидно, принадлежит типу `a`.
- Теперь будем разбираться с конструктором `S`. Этот конструкторв имеет параметр типа `a`, значит, если `x` - это значение типа `a`, то `S(x)` - это тоже значение типа `a`.
- Но раз так, то и `S(S(Nil)` – типа `a`. И вообще, любое выражение вида `S(S(...S(Nil)...))` – типа `a`. Но, поскольку кроме рассмотренных конструкторов `S` и `Nil` никаких других способов образовать тип a в его описании не указано, у него нет и других значений. Таким образом, мы исчерпывающе описали тип a.

Теперь разберемся, как нам, например, напечатать значение этого типа. Первая идея, приходящая в голову, не сработает:

```
let print_a v =
  match v with
    Nil       -> print_string "Nil"
  | S(Nil)    -> print_string "S(Nil)"
  | S(S(Nil)) -> print_string "S(S(Nil))"
  ...
```

поскольку значений типа бесконечно много.

Поэтому нам потребуется новое понятие - рекурсивная функция. Как видно из названия, это функция, которая вызывает сама себя. Вот как будет выглядеть рекурсивная функция печати этого типа:

```
let rec string_of_a v =
  match v with
    Nil  -> "Nil"
  | S(x) -> "S(" ^ string_of_a x ^ ")";;

let print_a v = print_string (string_of_a v);;
```

Обратим внимание на две тонкости в описании `string_of_a`.

Во-первых, после ключевого слова `let` появилось ранее не встечавшееся ключевое слово `rec`. Это слово необходимо писать при описании всякой рекурсивной функции. Во-вторых, в четвертой строке функции `string_of_a` происходит ее рекурсивный вызов — вызов описываемой функции.

Возможно, вам непонятно, что значит, что функция вызывает себя саму. Действительно, внешне это немного напоминает барона Мюнгхаузена, вытаскивающего себя за волосы из болота. Поступим в точности как со значениями — будем разбираться по отдельности.

Для удобства повторим код еще раз:

```
let rec string_of_a v =
  match v with
    Nil  -> "Nil"
  | S(x) -> "S(" ^ string_of_a x ^ ")";;
```

1. Чему равно `string_of_a Nil`? Ответ: `"Nil"`, поскольку эта ветвь вычислений не использует рекурсии. Давайте запомним это и более не будем думать об этом случае. Для четкости отразим это в таблице:

Выражение	Результат вычисления
`string_of_a Nil`	`"Nil"`

2. Чему равно `string_of_a (S(Nil))`? Проследив выполнение конструкции `match`, видим, что это будет `"S("^ string_of_a x ^ ")"`.

   Мы столкнулись с рекурсивным вызовом, но из таблицы мы ведь знаем результат конкретно этого рекурсивного вызова, ведь `(string_of_a Nil)` - это `"Nil"`. Поэтому мы можем этот результат подставить: `"S("^ "Nil"^ ")"`, то есть `"S(Nil)"`. Отразим и этот факт в таблице:

Выражение	Результат вычисления
`string_of_a Nil`	`"Nil"`
`string_of_a (S(Nil))`	`"S(Nil)"`

3. Чему равно `string_of_a (S(S(Nil)))`? Повторим рассуждение: это `"S("^ string_of_a (S(Nil)) ^ ")"`, то есть `"S(S(Nil))"`, что мы снова можем запомнить:

Выражение	Результат вычисления
`string_of_a Nil`	`"Nil"`
`string_of_a (S(Nil))`	`"S(Nil)"`
`string_of_a (S(S(Nil)))`	`"S(S(Nil))"`

Легко видеть, что так мы разберем все возможные случаи. Ведь каждое значение типа `a` имеет конечное число вложенных значений `S`, после которых, в самой глубине, хранится значение `Nil`. Наше же рассуждение, идя в обратную сторону, неизбежно дойдет до значения с любым конечным количеством букв `S`.

Конечно, эти рассуждения не полностью отражают реальный процесс вычислений, компьютер не запоминает результат функций, но для первого знакомства с рекурсивными функциями такого понимания должно быть вполне достаточно.

1.8.1 Натуральные числа

Напомним, как в аксиоматике Пеано представляются натуральные числа: постулируется существование константы 0 и операции прибавления 1 (*инкремента*). В формулах мы будем обозначать инкремент штрихами: число a'' — это число a, увеличенное на 1 два раза (то есть $a + 2$), а число 3 представимо как $0'''$. Что интересно, этих простых операций достаточно, чтобы определить все операции целочисленной арифметики (сложение, вычитание, умножение).

Операция	Определение
Инктремент	$inc(a) = a'$
Сложение	$plus(0, b) = b;\ plus(a', b) = (plus(a, b))'$
Декремент	$dec(a') = a;\ dec(0) = 0$
Вычитание	$minus(a', b') = minus(a, b);\ minus(0, a) = 0;\ minus(a, 0) = a$
Умножение	$mul(a', b) = plus(b, mul(a, b));\ mul(0, b) = 0$

Пример. Давайте вычислим $3 \cdot 2$ с помощью этих определений. Заметим, что 3 — это $0'''$, а 2 — это $0''$. Тогда вычисление $mul(0''', 0'')$ будет состоять из следующих преобразований:

Выражение в арифметике Пеано	Обычная запись
$mul(0''', 0'') =$	$3 \cdot 2$
$plus(0'', mul(0'', 0'')) =$	$2 + (2 \cdot 2)$
$plus(0'', plus(0'', mul(0', 0''))) =$	$2 + (2 + (1 \cdot 2))$
$plus(0'', plus(0'', plus(0'', mul(0, 0')))) =$	$2 + (2 + (2 + (0 \cdot 2)))$
$plus(0'', plus(0'', plus(0'', 0))) =$	$2 + (2 + (2 + 0))$
$plus(0'', plus(0'', (plus(0', 0))')) =$	$2 + (2 + ((1 + 0) + 1)$
$plus(0'', plus(0'', (plus(0, 0))'')) =$	$2 + (2 + ((0 + 0)) + 2)$
$plus(0'', plus(0'', 0'')) =$	$2 + (2 + 2)$
$plus(0'', (plus(0', 0''))') =$	$2 + ((1 + 2) + 1)$
$plus(0'', (plus(0, 0''))'') =$	$2 + ((0 + 2) + 2)$
$plus(0'', 0'''') =$	$2 + 4$
$(plus(0', 0''''))' =$	$(1 + 4) + 1$
$(plus(0, 0''''))'' =$	$(0 + 4) + 2$
$0'''''$	6

Математическую сторону этих определений (доказательство корректности и т.п.) мы опустим, а вот на программистскую как раз и обратим внимание. Ведь тип данных `a`, описанный выше, и является как раз типом данных, представляющим натуральные числа «в стиле аксиоматики Пеано».

Покажем, например, как реализовать операцию инкремента, это очень просто:

```
let inc x = S x;; (* дописываем одну букву S - прибавляем один *)
```

Чуть посложнее операция сложения, там в определении есть два случая. Первый случай — когда первое слагаемое равно 0, и второй случай, когда оно содержит применение как минимум одной операции прибавления 1 ($'$). Впрочем, и здесь мы можем впрямую следовать определению.

```
let rec plus a b =
  match a with
    Nil -> b
  | S a -> S (plus a b);;
```

А так реализуется вычитание:

```
let rec minus a1 b1 =
  match (a1, b1) with  (* разбор случаев из определения *)
    (S a, S b) -> minus a b
  | (Nil, a) -> Nil
  | (a, Nil) -> a;;
```

И завершит этот пример функция преобразования значений типа `a` в значения типа `int`:

```
let rec int_of_a v =
  match v with
    Nil -> 0
  | S(x) -> 1 + int_of_a x;;
```

Здесь мы действуем впрямую по определению: если значение типа `a` имеет вид `S(x)` — прибавление 1 к x — то надо вычислить значение для `x`, а затем прибавить к нему 1.

1.8.2 Списки

Мы уже познакомились с одним частным случаем списка — типом `a` из предыдущего параграфа. Теперь настала пора познакомиться с ними в общем случае.

Вообще, список можно было бы описать примерно так:

```
type 'a list = Cons of 'a * 'a list | Nil
```

Как видите, это тип похож на тип `a`, главное его отличие — этом непонятном символе `'a` перед именем `list`, а также в наличии дополнительного значения в каждом элементе `Cons`. Естественно, значения тоже похожи:

```
Cons (1, Cons (2, Cons (3, Nil)))
```

В силу частоты использования, тип список встроен в язык, и конструкторы типа `Cons` и `Nil` имеют специальные имена: `::` и `[]` соответственно, а вместо `Cons (1, Cons (2, Cons (3, Nil)))` надо писать `1 :: 2 :: 3 :: []` Также, есть еще один вариант записи этого списка: `[1;2;3]`,

Естественно, эти сокращения так же применяются и в сопоставлении с образцом.

Важно! Конструктор типа `::` несимметричен. Слева от него всегда должен быть указан элемент, а справа — список, например так: `1 :: []`. Выражение `[] :: 1` некорректно.

Но при этом `[1]::[[1]]`, так же, как и `[1]::[]` — корректные списки, состоящие из списков целых чисел.

Пример (подсчет длины списка, то есть количества `::`):

```
let rec list_length l =
  match l with
    [] -> 0
  | l1::ls -> 1 + list_length ls;;
```

В том случае, если список состоит только из `[]`, он имеет длину 0. Если список - это `l1::[]`, то длина равна $1 +$ `list_length []`, то есть 1. Если список - это `l2::l1::[]`, то длина - $1 +$ `list_length l1::[]`, то есть 2. И в общем случае, если есть список `l1::ls`, то его длина равна $1 +$ `list_length ls`.

Список может хранить элементы любого типа, но этот тип должен быть для всех элементов один и тот же. Тип списка `[1;2;3]` — `int list`, тип списка `["1";"a";"c"]` — `string list`, список `[[1];[2];[]]` имеет тип `int list list` (для ясности можно поставить скобки: `(int list) list`), но списка `[1;"2"]` быть не может.

1.8.3 Сигнатура типа

Сигнатурой типа мы будем называть его обозначение на языке Окамль. Сигнатуры элементарного типа нам уже знакомы:

Тип	Сигнатура типа
строчки	string
целые числа	int
плавающие числа	float
булевские (логические) значения	bool
одноэлементный тип	unit

Но мы уже умеем строить значения не только этих простых типов, но и более сложных конструкций: речь идет о функциях.

Рассмотрим следующую функцию:

```
let x a b = if a > 0 then b else 1.;;
```

Эта функция берет два аргумента (целочисленный и плавающий) и возвращает плавающее число. Соответственно, сигнатура этой функции имеет следующий вид:

```
x: int -> float -> float
```

Первая из стрелок `->`, указанная в сигнатуре, разделяет сигнатуры типов аргументов функции, а вторая отделяет тип аргумента от типа результата функции.

Как можно догадаться, какой тип у функции? Когда мы пишем функцию, мы уже должны иметь представление, аргументы каких типов ей передаются и какой тип значений она должна возвращать. Ведь мы же знаем, что она должна делать. Тем не менее компилятор не знает о наших предположениях, он судит по тому коду, который мы написали — и, основываясь на нем, автоматически выводит типы аргументов и результата. Попробуем сделать это и мы.

Это как головоломка. Вот есть такой код:

```
let z x y z = if (x + y) / 2 > 0 then "false" else z;;
```

Что можно по нему заключить?

1. Формальные параметры `x` и `y` складываются между собой операцией `+`, которая требует, чтобы левый и правый операнды имели тип `int`. Значит, `x` и `y` могут иметь только тип `int` — иначе код содержит ошибку.

2. Тип результата — `string`, поскольку результат вычисления операции `if` является результатом всей функции — а одна из ветвей возвращает значение `"false"`, имеющего тип `string`.

3. И наконец ветви операции `if` должны иметь одинаковый тип — значит, тип результата совпадает с типом параметра `z`.

4. Итого, аргументы `x` и `y` типа `int`, аргумент `z` типа `string`, и результат функции — тоже типа `string`. Значит, сигнатура типа функции `z` такова: `int -> int -> string -> string`.

Рассмотрим еще несколько примеров:

Значение	Сигнатура типа
`let v () = "это строка";;`	`v: unit -> string`
`let w () = print_string "Привет";;`	`w: unit -> unit`
`let x a b = (a = 0) || (b = 0.);;`	`x: int -> float -> bool`
`let y a b c = (a = "") || (b = 0) || c;;`	`y: string -> int -> bool -> bool`
`let z x = if x = "4" then 1 else -1;;`	`z: string -> int`