Разработка программ с помощью Objective Caml
Developing Applications with Objective Caml

Emmanuel Chailloux
Pascal Manoury
Bruno Pagano

6 октября 2012 г.

Оглавление

Глава 1

Как заполучить Objective CAML

Часть I

Основы языка

Глава 2

Функциональное программирование

2.1 Введение

Первый язык функционального программирования LISP появился в конце 1950, в тот же момент, что и Fortran — один из первых императивных языков. Оба этих языка существуют и по сей день, хотя они немало изменились. Область их применения: вычислительные задачи для Фортрана и символьные (symbolic) для Lisp. Интерес к функциональному программированию состоит в простоте написания программ, где под программой подразумевается функция, применённая к аргументам. Она вычисляет результат, который возвращается как вывод программы. Таким образом можно с лёгкостью комбинировать программы: вывод одной, становится входным аргументом для другой.

Функциональное программирование основывается на простой модели вычислений, состоящей из трёх конструкций: переменные, определение функции и её применение к какому–либо аргументу. Эта модель, называемая λ–исчисление, была введена Alonzo Church в 1932, ещё до появления первых компьютеров. В λ–исчислении любая функция является переменной, так что она может быть использована как входной параметр другой функции, или возвращена как результат другой. Теория λ–исчисления утверждает что все то, что вычисляемо может быть представлено этим формализмом. Однако, синтаксис этой теории слишком ограничен, чтобы его можно было использовать его как язык программирования. В связи с этим к λ–исчислению были добавлены базовые типы (например, целые числа или строки символов), операторы для этих типов, управляющие структуры и объявление позволяющие именовать переменные или

функции, и в частности рекурсивные функции.

Существует разные классификации языков функционального программирования. Мы будем различать их по двум характеристикам, которые нам кажутся наиболее важными:

- без побочных эффектов (чистые), или с побочным эффектом (не чистые): чистый язык — это тот, в котором не существует изменения состояния. Все есть вычисление, и как оно происходит, нас не интересует. Не чистые языки, такие как Caml или ML, имеют императивные особенности, такие как изменение состояния. Они позволяют писать программы в стиле близкому к Фортрану, в котором важен порядок вычисления выражений.

- язык типизирован динамически или статически: типизация необходима для проверки соответствия аргумента, переданного функции, типу формального параметра. Это проверка может быть выполнена во время выполнения программы. В этом случае типизация называется динамической. В случае ошибки программа будет остановлена, как это происходит в Lisp. В случае статической типизации проверка осуществляется во время компиляции, то есть до выполнения программы. Таким образом она (проверка) не замедлит программу во время выполнения. Эта типизация используется в ML и в его диалектах, таких как Objective CAML. Только правильно типизированные программы, то есть успешно прошедшие проверку типов, могут быть скомпилированы и затем выполнены.

2.2 План главы

В этой главе представлены базовые элементы функциональной части языка Objective CAML, а именно: синтаксис, типы и механизм исключений. После этого вводного курса мы сможем написать нашу первую программу.

В первом разделе описаны основы языка, начиная с базовых типов и функций. Затем мы рассмотрим структурные и функциональные типы. После этого мы рассмотрим управляющие структуры, а также локальные и глобальные объявления. Во втором разделе мы обсудим определение типов для создания структур и механизм сопоставления с образцом, который используется для доступа к этим структурам. В третьем разделе рассматривается выводимый тип функций и область их применения, после чего следует описание механизма исключений. Четвёртый

раздел объединяет введённые понятия в простой пример программы–калькулятора.

2.3 Функциональное ядро Objective CAML

Как любой другой язык функционального программирования, Objective CAML — это язык выражений, состоящий в основном из создания функций и их применения. Результатом вычисления одного из таких выражений является значение данного языка (value in the language) и выполнение программы заключается в вычислении всех выражений из которых она состоит.

2.3.1 Значения, функции и базовые типы

В Objective CAML определены следующие типы: целые числа, числа с плавающей запятой, символьный, строковый и логический.

Числа

Различают целые `int` и числа с плавающей запятой `float`. Objective CAML следует спецификации IEEE 754 для представления чисел с плавающей запятой двойной точности. Операции с этими числами описаны в таблице 2.3.1. Если результат целочисленной операции выходит за интервал значений типа `int`, то это не приведёт к ошибке. Результат будет находиться в интервале целых чисел, то есть все действия над целыми ограниченны операцией `modulo` с границами интервала.

Разница между целыми числами и числами с плавающей запятой.

Значения разных типов, таких как `float` и `int`, не могут сравниваться между собой напрямую. Для этого существует функции перевода одного типа в другой (`float_of_int` и `int_of_float`).

```
# 2 = 2.0;;
Error: This expression has type float but an expression was expected of
    type
         int
# 3.0 = float_of_int 3;;
- : bool = true
```

Таблица 2.1: Операции над числами

целые	плавающие
+ сложение	+. сложение
- вычитание и унарный минус	-. вычитание и унарный минус
* умножение	*. умножение
/ деление	/. деление
`mod` остаток целочисленного деления	** возведение в степень

```
# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
(* limits of the representation
   *)
(* of integers *)
# 2147483650 ;;
- : int = 2
```

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
# 1. /. 0. ;;
- : float = inf
(* limits of the representation
   *)
(* of floating-point numbers *)
# 222222222222.11111 ;;
- : float = 222222222222
```

Аналогично, операции над целыми числами и числами с плавающей запятой различны:

```
# 3 + 2;;
- : int = 5
# 3.0 +. 2.0;;
- : float = 5.
# 3.0 + 2.0;;
Error: This expression has type float but an expression was expected of
    type
         int
# sin 3.14159;;
- : float = 2.65358979335273e-06
```

Неопределённый результат, например получаемый при делении на ноль, приведет к возникновению исключения (см. **??**, стр. **??**), которое остановит вычисление. Числа с плавающей запятой имеют специальные значения для бесконечных величин (`Inf`) и для не определённого результата (`NaN` [1]). Основные операции над этими числами приведены в таблице 2.3.1

Таблица 2.2: Функции над числами с плавающей запятой

`ceil`	`cos` косинус
`floor`	`sin` синус
`sqrt` — квадратный корень	`tan` тангенс
`exp` — экспонента	`acos` арккосинус
`log` — натуральный логарифм	`asin` арксинус
`log10` — логарифм по базе 10	`atan` арктангенс

```
# ceil 3.4 ;;
- : float = 4.
# floor 3.4 ;;
- : float = 3.
# ceil (-.3.4) ;;
- : float = -3.
# floor (-.3.4) ;;
- : float = -4.
```

```
# sin 1.57078 ;;
- : float = 0.999999999866717837
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679489656
# asin 3.14 ;;
- : float = nan
```

Символы и строки

Символы, тип `char`, соответствуют целым числам в интервале от 0 до 255, первые 128 значений соответствуют кодам ASCII. Функция `char_of_int` и `int_of_char` преобразуют один тип в другой. Строки, тип `string` — это последовательность символов определенной длинны (не длиннее $2^{24} - 6$). Оператором объединения строк (конкатенации) является шапка $\wedge$. Следующие функции необходимы для перевода типов `int_of_string`, `string_of_int`, `string_of_float` и `float_of_string`.

```
# 'B' ;;
```

[1] Not a Number

```
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "est une îchane" ;;
- : string = "est une cha\195\174ne"
# (string_of_int 1987) ^ " est l'éanne de la écration de CAML" ;;
- : string = "1987 est l'ann\195\169e de la cr\195\169ation de
   CAML"
```

Если строка состоит из цифр, то мы не сможем использовать ее в численных операциях, не выполнив явного преобразования.

```
# "1999" + 1 ;;
Error: This expression has type string but an expression was expected of
    type
         int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

В модуле `String` собрано много функций для работы со строками (стр. **??**)

Булевый тип

Значение типа `boolean` принадлежит множеству состоящему из двух элементов: `true` и `false`. Основные операторы описаны в таблице 2.3.1. По историческим причинам операторы `and` и `or` имеют две формы.

Таблица 2.3: Булевы операторы

not — отрицание	
&& логическое и	& синоним &&
|| логическое или	`or` синоним |

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
```

Операторы && и || или их синонимы, вычисляют аргумент слева и в зависимости от его значения, вычисляют правый аргумент. Они могут быть переписаны в виде условной структуры (см. **??** стр. **??**).

Таблица 2.4: Операторы сравнения и равенства

= равенство структурное	< меньше
== равенство физическое	> больше
<> отрицание =	<= меньше или равно
!= отрицание ==	>= больше или равно

Операторы сравнения и равенства описаны в таблице 2.3.1. Это полиморфные операторы, то есть они применимы как для сравнения двух целых, так и двух строк. Единственное ограничение это то что операнды должны быть одного типа (см. **??** стр. **??**).

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one" < "two" ;;
- : bool = true
# 0 < '0' ;;
Error: This expression has type char but an expression was expected of
    type
         int
```

Структурное равенство при проверке двух переменных сравнивает значение полей структуры, тогда как физическое равенство проверяет занимают ли эти переменные одно и то же место в памяти. Оба оператора возвращают одинаковый результат для простых типов: `boolean`, `char`, `int` и константные конструкторы (см. **??** стр. **??**).

Осторожно

Числа с плавающей запятой и строки рассматриваются как структурные типы.

Объединения

Тип `unit` определяет множество из всего одного элемента, обозначается: ()

```
# () ;;
- : unit = ()
```

Это значение будет широко использоваться в императивных программах (см. **??** стр. **??**), в функциях с побочным эффектом. Функции, результат которых равен (), соответствуют понятию процедуры, которое отсутствует в Objective CAML, так же как и аналог типа `void` в языке C.

Декартово произведение, кортежи

Значения разных типов могут быть сгруппированы в кортежи. Значения из которых состоит кортеж разделяются запятой. Для конструкции кортежа, используется символ `*`. `int*string` есть кортеж, в котором первый элемент целое число и второй строка.

```
# ( 12 , "October" ) ;;
- : int * string = (12, "October")
```

Иногда мы можем использовать более простую форму записи.

```
# 12 , "October" ;;
- : int * string = (12, "October")
```

Функции `fst` и `snd` дают доступ первому и второму элементу соответственно.

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

Эти обе функции полиморфные, входной аргумент может быть любого типа.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

Тип `int*char*string` — это триплет, в котором первый элемент типа `int`, второй `char`, а третий — `string`.

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = (65, 'B', "ascii")
```

Осторожно

Если аргумент функций `fst` и `snd` не пара, а другой n–кортеж, то мы получим ошибку.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Error: This expression has type int * char * string
       but an expression was expected of type 'a * 'b
```

Существует разница между парой и триплетом: тип `int*int*int` отличен от `(int*int)*int` и `int*(int*int)`. Методы доступа к элементам триплета (и других кортежей) не определены в стандартной библиотеке. В случае необходимости мы используем сопоставление с образцом (см. **??**).

Списки

Значения одного и того же типа могут быть объединены в списки. Список может быть либо пустым, либо содержать однотипные элементы.

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two" ; 3 ] ;;
Error: This expression has type string but an expression was expected of
    type
         int
```

Для того чтобы добавить элемент в начало списка существует следующая функция в виде инфиксного оператора :: — аналог `cons` в Caml.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Для объединения (конкатенации) списков существует инфиксный оператор: @.

```
# [ 1 ] @ [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ] @ [ 3 ] ;;
- : int list = [1; 2; 3]
```

Остальные функции манипуляции списками определены в библиотеке `List`. Функции `hd` и `tl` дают доступ к первому и последнему элементу списка.

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Exception: Failure "hd".
```

В последнем примере получить первый элемент пустого списка действительно «сложно», поэтому возбуждается исключение (см. **??**).

2.3.2 Структуры условного контроля

Одна из структур контроля необходимая в каждом языка программирования — условный оператор.

Синтаксис:

```
if expr1 then expr2 else expr3
```

Тип выражения `expr1` равен `bool`. Выражения `expr2` и `expr3` должны быть одного и того же типа.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
- : string = "4"
# if 3=4 then 0 else "4";;
Error: This expression has type string but an expression was expected of
    type
         int
```

Замечание

Ветка `else` может быть опущена, в этом случае будет подставлено значение по умолчанию равное `else ()`, в соответствии с этим выражение `expr2` должно быть типа `unit` (см. **??** стр. **??**).

2.3.3 Объявление значений

Определение связывает имя со значением. Различают глобальные и локальные определения. В первом случае, объявленные имена видны во всех выражениях, следуют за ним, во втором — имена доступны только в текущем выражении. Мы также можем одновременно объявить несколько пар имя-значение.

Глобальные объявления

Синтаксис:

```
let name = expr ;;
```

Глобальное объявление определяет связь имени `nom` со значением выражения `expr`, которое будет доступно всем следующим выражениям.

```
# let yr = "1999" ;;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

Одновременное глобальное объявление

Синтаксис:

```
let nom1 = expr1
and nom2 = expr2
:
and nomn = exprn;;
```

При одновременном объявлении переменные будут известны только к концу всех объявлений.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Error: Unbound value z
```

Можно сгруппировать несколько глобальных объявлений в одной фразе, вывод типов и значений произойдёт к концу фразы, отмеченной «;;». В данном случае объявления будут вычислены по порядку.

```
# let x = 2
  let y = x + 3 ;;
val x : int = 2
val y : int = 5
```

Глобальное объявление может быть скрыто локальным с тем же именем (см. ?? стр. ??).

Локальное объявление

Синтаксис:

```
let nom = expr1 in expr2;;
```

Имя `nom` связанное с выражением `expr1` известно только для вычисления `expr2`.

```
# let xl = 3 in xl * xl ;;
- : int = 9
```

Локальное объявление, которое связывает `xl` со значением `3`, существует только в ходе вычисления `xl*xl`.

```
# xl ;;
Error: Unbound value xl
```

Локальное объявление скрывает любое глобальные с тем же именем, но как только мы выходим из блока в котором была оно определено, мы находим старое значение связанное с этим именем.

```
# let x = 2 ;;
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

Локальное объявление — это обычное выражение, соответственно оно может быть использовано для построения других выражений.

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Локальные объявления так же могут быть одновременными.

```
let  name1 = expr1
and  name2 = expr2
:
and  namen = exprn
in  expr ;;
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5.
# b ;;
Error: Unbound value b
```

2.3.4 Функциональное выражение, функции

Функциональное выражение состоит из параметра и тела. Формальный параметр — это имя переменной, а тело — это выражение. Обычно говорят что формальный параметр является абстрактным, по этой причине функциональное выражение тоже называется абстракцией.

Синтаксис:

```
function p -> expr
```

Таким образом функция возведения в квадрат будет выглядеть так:

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Objective CAML сам определяет тип. Функциональный тип `int->int` это функция с параметром типа `int`, возвращающая значение типа `int`.

Функция с одним аргументом пишется как функция и аргумент следующий за ней.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Вычисление функции состоит в вычисление её тела, в данном случае `x*x`, где формальный параметр `x`, заменён значением аргумента (эффективным параметром), здесь он равен 5.

При конструкции функционального выражения `expr` может быть любым выражением, в частности функциональным.

```
# function x -> (function y -> 3*x + y) ;;
- : int -> int -> int = <fun>
```

Скобки не обязательны, мы можем писать просто:

```
# function x -> function y -> 3*x + y ;;
- : int -> int -> int = <fun>
```

В простом случае мы скажем, что функция ожидает два аргумента целого типа на входе и возвращает значение целого типа. Но когда речь идёт о функциональном языке, таком как Objective CAML, то скорее это соответствует типу функции с входным аргументом типа `int` и возвращающая функциональное значение типа `int->int`:

```
# (function x -> function y -> 3*x + y) 5 ;;
- : int -> int = <fun>
```

Естественно, мы можем применить это функциональное выражение к двум аргументам. Для этого напишем:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Когда мы пишем `f a b`, подразумевается применение `(f a) к b`.

Давайте подробно рассмотрим выражение

```
(function x -> function y -> 3*x + y) 4 5
```

Для того чтобы вычислить это выражение, необходимо сначала вычислить значение

```
(function x -> function y -> 3*x + y) 4
```

что есть *функциональное выражение* равное

```
function y -> 3*4 + y
```

в котором `x` заменён на 4 в выражении `3 * x + y`. Применяя это значение (являющееся функцией) к 5, мы получаем конечное значение `3 * 4 + 5 = 17`:

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Арность функции

Арностью функции называется число аргументов функции. По правилам, унаследованным из математики, аргументы функции задаются в скобках после имени функции. Мы пишем: `f(4,5)`. Как было указанно

ранее, в Objective CAML мы чаще используем следующий синтаксис: `f 4 5`. Естественно, можно написать функциональное выражение применимое к `(4,5)`:

```
# function (x,y) -> 3*x + y ;;
- : int * int -> int = <fun>
```

Но в данном случае, функция ожидает не два аргумента, а один; тип которого пара целых. Попытка применить два аргумента к функции ожидающей пару или наоборот, передать пару функции для двух аргументов приведёт к ошибке:

```
# (function (x,y) -> 3*x + y) 4 5 ;;
Error: This function is applied to too many arguments;
maybe you forgot a ';'
# (function x -> function y -> 3*x + y) (4, 5) ;;
Error: This expression has type int * int
       but an expression was expected of type int
```

Альтернативный синтаксис

Существует более компактная форма записи функций с несколькими аргументами, которая дошла к нам из старых версий *Caml*. Выглядит она так:

Синтаксис

```
fun p1 ... pn -> expr
```

Это позволяет не повторять слово `function` и стрелки. Данная запись эквивалентна

```
function p1 -> -> function pn -> expr
# fun x y -> 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y -> 3*x + y) 4 5 ;;
- : int = 17
```

Эту форму можно часто встретить в библиотеках идущих в поставку с Objective CAML.

Замыкание

Objective CAML рассматривает функциональное выражение также как любое другое. Значение возвращаемое функциональным выражением на-

зывается замыканием. Каждое выражение Objective CAML вычисляется в окружении состоящем из соответствий имя–значение, которые были объявлены до вычисляемого выражения. Замыкание может быть описано как триплет, состоящий из имени формального параметра, тела функции и окружения выражения. Нам необходимо хранить это окружение, поскольку в теле функции кроме формальных параметров могут использоваться другие переменные. В функциональном выражении эти переменные называются свободными, нам понадобится их значение в момент применения функционального выражения.

```
# let m = 3 ;;
val m : int = 3
# function x -> x + m ;;
- : int -> int = <fun>
# (function x -> x + m) 5 ;;
- : int = 8
```

В случае когда применение замыкания к аргументу возвращает новое замыкание, оно (новое замыкание) получает в своё окружение все необходимые связи для следующего применения. Раздел **??** подробно рассматривает эти понятия. В главе **??**, а так же в главе **??** мы вернёмся к тому как замыкание представляется в памяти.

До сих пор рассмотренные функциональные выражения были *анонимными*, однако мы можем дать им имя. Объявление функциональных значений

Объявление функциональных значений

Функциональное значение объявляется так же как и другие, при помощи конструктора `let`

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x -> function y -> 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Для упрощения записи, можно использовать следующий синтаксис:
Синтаксис:

```
let nom p1 ... pn = expr
```

что эквивалентно:

```
let nom=function p1−>−>function pn−> expr
```

Следующие объявления `succ` и g эквивалентны предыдущим:

```
# let succ x = x + 1 ;;
val succ : int −> int = <fun>
# let g x y = 2∗x + 3∗y ;;
val g : int −> int −> int = <fun>
```

В следующем примере демонстрируется функциональная сторона Objective CAML, где функция `h1` получена применением `g` к аргументу. В данном случае мы имеем дело с частичным применением.

```
# let h1 = g 1 ;;
val h1 : int −> int = <fun>
# h1 2 ;;
− : int = 8
```

С помощью `g` мы можем определить другую функцию `h2` фиксируя значение второго параметра `y`:

```
# let h2 = function x −> g x 2 ;;
val h2 : int −> int = <fun>
# h2 1 ;;
− : int = 8
```

Объявление инфиксных функций

Некоторые бинарные функции могут быть использованы в инфиксной форме. Например при сложении двух целых мы пишем `3 + 5` для применения `+` к `3` и `5`. Для того чтобы использовать символ `+` как классическое функциональное значение, необходимо указать это, окружая символ скобками (`op`).

В следующем примере определяется функция `succ` используя `(+)`:

```
# (+) ;;
− : int −> int −> int = <fun>
# let succ = (+) 1 ;;
val succ : int −> int = <fun>
# succ 3 ;;
```

```
- : int = 4
```

Таким образом мы можем определить новые операторы, что мы и сделаем определив `++` для сложения двух пар целых.

```
# let (++) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ( ++ ) : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = (2, 3)
# c ++ c ;;
- : int * int = (4, 6)
```

Существуют однако ограничения на определение новых операторов, они должны содержать только *символы* (такие как `*,+,$`, etc.), исключая буквы и цифры. Следующие функции являются исключением:

```
or mod land lor lxor lsr asr
```

Функции высшего порядка

Функциональное значение (замыкание) может быть возвращено как результат, а так же передано как аргумент функции. Такие функции, берущие на входе или возвращающие функциональные значения, называются функциями высшего порядка.

```
# let h = function f -> function y -> (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Замечание Выражения группируются справа налево, но функциональные типы объединяются слева направо. Таким образом тип функции `h` может быть написан:

```
(int -> int) -> int -> int или(int -> int) -> (int -> int)
```

При помощи функций высшего порядка можно элегантно обрабатывать списки. К примеру функция `List.map` применяет какую-нибудь функцию ко всем элементам списка и возвращает список результатов.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

Другой пример — функция `List.for_all` проверяет соответствуют ли элементы списка определённому критерию.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n -> n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n -> n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Видимость переменных

Для вычисления выражения, необходимо чтобы все используемые им переменные были определены, как, например, выражение `e` в определении

```
let p=e
```

Переменная `p` не известна в этом выражении, она может быть использована только в случае если `p` была объявлена ранее.

```
# let p = p ^ "-suffixe" ;;
Error: Unbound value p
# let p = "éprfixe" ;;
val p : string = "pr\195\169fixe"
# let p = p ^ "-suffixe" ;;
val p : string = "pr\195\169fixe-suffixe"
```

В Objective CAML переменные связаны статически. При применении замыкания используется окружение в момент её (замыкания) объявления (статическая видимость), а не в момент её применения (динамическая видимость)

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = (10, 10, 20)
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = (1000, 10, 1010)
```

В функции `k` имеется свободная переменная `p`, которая была определена в глобальном окружении, поэтому определение `k` принято. Связь между именем `p` и значением 10 в окружении замыкания `k` статическая, то есть не зависит от последнего определения `p`.

Рекурсивное объявление

Объявление переменной называется рекурсивным, если оно использует свой собственный идентификатор в своём определении. Эта возможность часто используется для определения рекурсивных функций. Как мы видели ранее, `let` не позволяет делать это, поэтому необходимо использовать специальный синтаксис:

```
let rec nom = expr ;;
```

Другой способ записи для функции с аргументами:

```
let rec nom p1 ... pn = expr ;;
```

Определим функцию `sigma` вычисляющую сумму целых от `0` до значения указанного аргументом:

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
# sigma 10 ;;
- : int = 55
```

Как заметил читатель, эта функция рискует «не закончится» если входной аргумент меньше 0.

Обычно, рекурсивное значение — это функция, компилятор не принимает некоторые рекурсивные объявления, значения которых не функциональные.

```
# let rec x = x + 1 ;;
Error: This kind of expression is not allowed as right-hand side of 'let
    rec'
```

Как мы увидим позднее, такие определения все таки возможны в некоторых случаях (см. **??** стр. **??**).

Объявление `let rec` может быть скомбинировано с `and`. В этом случае функции определённые на одном и том же уровне, видны всем остальным. Это может быть полезно при декларации взаимно рекурсивных функций.

```
# let rec pair n = (n<>1) && ((n=0) or (impair (n-1))) and impair n
    = (n<>0) &&
```

```
((n=1) or (pair (n−1)));;
val pair : int −> bool = <fun>
val impair : int −> bool = <fun>
# pair 4 ;;
− : bool = true
# impair 5 ;;
− : bool = true
```

По тому же принципу, локальные функции могут быть рекурсивными. Новое определение функции `sigma` проверяет корректность входного аргумента, перед тем как посчитать сумму локальной функцией `sigma_rec`.

```
# let sigma x = let rec sigma_rec x = if x = 0 then 0 else x +
    sigma_rec (x−1)
in if (x<0) then "erreur : argument negatif" else "sigma = " ^ (
    string_of_int
(sigma_rec x)) ;;
val sigma : int −> string = <fun>
```

Замечание

Мы вынужденны были определить возвращаемый тип как `string`, поскольку необходимо чтобы он был один и тот же, независимо от входного аргумента, отрицательного или положительного. Какое значение должна вернуть `sigma` если аргумент больше нуля? Далее, мы увидим правильный способ решения этой проблемы (см. **??** стр. **??**).

2.3.5 Полиморфизм и ограничение типа

Некоторые функции выполняют одни и те же инструкции независимо от типа аргументов. К примеру, для создание пары из двух значений нет смысла определять функции для каждого известного типа. Другой пример, доступ к первому полю пары не зависит от того, какого типа это поле.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a −> 'b −> 'a * 'b = <fun>
# let p = make_pair "papier" 451 ;;
val p : string * int = ("papier", 451)
# let a = make_pair 'B' 65 ;;
val a : char * int = ('B', 65)
# fst p ;;
− : string = "papier"
```

```
# fst a ;;
- : char = 'B'
```

Функция, для которой не нужно указывать тип входного аргумента или возвращаемого значения называется полиморфной. Синтезатор типов, включённый в компилятор Objective CAML находит наиболее общий тип для каждого выражения. В этом случае Objective CAML использует переменные, здесь они обозначены как `'a` и `'b`, для указания общих типов. Эти переменные конкретизируются типом аргумента в момент применения функции.

При помощи полиморфных функций, мы получаем возможность написания универсального кода для любого типа переменных, сохраняя при этом надёжность статической типизации. Действительно, несмотря на то что `make_paire` полиморфная, значение созданное (`make_paire '6' 65`) имеет строго определённый тип, который отличен от (`make_paire "paire"451`). Проверка типов реализуется в момент компиляции, таким образом универсальность кода никак не сказывается на эффективности программы.

Пример функций и полиморфных значений

В следующем примере приведена полиморфная функция с входным параметром функционального типа.

```
# let app = function f -> function x -> f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

Мы можем применить её к функции `impaire`, которая была определена ранее.

```
# app impair 2;;
- : bool = false
```

Функция тождества (`id`) возвращает полученный аргумент.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id 1 ;;
- : int = 1
```

Следующая функция, `compose` принимает на входе две функции и ещё один аргумент, к которому применяет две первые.

```
# let compose f g x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
# let add1 x = x+1 and mul5 x = x*5 in compose mul5 add1 9 ;;
- : int = 50
```

Как мы видим, тип результата возвращаемого `g` должен быть таким же как и тип входного аргумента `f`.

Не только функциональные значения могут быть полиморфными, проиллюстрируем это на примере пустого списка.

```
# let l = [] ;;
val l : 'a list = []
```

Следующий пример иллюстрирует тот факт, что создание типов основывается на разрешении ограничений, накладываемых применением функций, а вовсе не на значениях, полученных в процессе выполнения.

```
# let q = List.tl [2] ;;
val q : int list = []
```

Здесь тип равен `List.tl 'a list->'a list`, то есть эта функция применяется к списку целых и возвращает список целых. Тот факт что в момент выполнения возвращён пустой список, не влияет на его тип.

Objective CAML создаёт параметризованные типы для каждой функции, которые не зависят от её аргументов. Этот вид полиморфизма называется *параметрическим полиморфизмом* [2].

Ограничение типа

Синтезатор типа Objective CAML образует самый общий тип и иногда бывает необходимо уточнить тип выражения.

Синтаксис ограничения типа следующий:

```
(expr:t)
```

В этом случае синтезатор типа воспользуется этим ограничением при конструкции типа выражения. Использование ограничения типа позволяет

- сделать видимым тип параметра функции
- запретить использование функции вне свой области применения

[2] Некоторые встроенные функции не подчиняются этому правилу, особенно функция структурного равенства (`=`), которая является полиморфной (её тип `'a -> 'a -> bool`), но она исследует структуру своих аргументов для проверки равенства.

- уточнить тип выражения, это окажется необходимым в случае физически изменяемых значений (см. **??** стр. **??**).

Рассмотрим использование ограничения типа.

```
# let add (x:int) (y:int) = x + y ;;
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int -> int) (g : int -> int) (x:int) = compose f
    g x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <
   fun>
# let nil = ([] : string list);;
val nil : string list = []
# 'H'::nil;;
Error: This expression has type string list
       but an expression was expected of type char list
```

Это ограничение полиморфизма позволяет лучше контролировать тип выражений, тем самым ограничивая тип определённый системой. В таких выражениях можно использовать любой определённый тип.

```
# let llnil = ([] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

`llint` является списком списков любого типа.

В данном случае подразумевается ограничение типа, а не явная типизация заменяющая тип, который определил Objective CAML. В частности, мы не можем обобщить тип, более чем это позволяет вывод типов Objective CAML.

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Ограничения типа будут использованы в интерфейсах модулей **??**, а так же в декларации классов **??**

2.3.6 Примеры

В этом параграфе мы приведём несколько примеров функций. Большинство из них уже определены в Objective CAML, мы делаем это только в «педагогических» целях.

Тест остановки рекурсивных функций реализован при помощи проверки, имеющей стиль более близкий к `Lisp`. Мы увидим как это сделать в стиле ML (см. **??**).

Размер списка

Начнём с функции проверяющей пустой список или нет.

```
# let null l = (l=[]);;
val null : 'a list -> bool = <fun>
```

Определим функцию `size` вычисления размера списка (т.е. число элементов).

```
# let rec size l = if null l then 0 else 1+(size(List.tl l));;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```

Функция `size` проверяет список: если он пуст возвращает 0, иначе прибавляет 1 к длине остатка списка.

Итерация композиций (Iteration of composition)

Выражение `iterate n f` вычисляет f^n, соответствующее применение функции `f n` раз.

```
# let rec iterate n f =
    if n = 0 then (function x -> x)
    else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

Функция `iterate` проверяет аргумент n на равенство нулю, если аргумент равен, то возвращаем функцию идентичности, иначе возвращаем композицию `f` с итерацией `f n-1` раз.

Используя `iterate` можно определить операцию возведения в степень, как итерацию умножения.

```
# let rec power i n =
  let i_times = ( * ) i in
    iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
```

```
- : int = 256
```

Функция `power` повторяет n раз функциональное выражение `i_times`, затем применяет этот результат к 1, таким образом мы получаем n-ю степень целого числа.

Таблица умножения

Напишем функцию `multab`, которая вычисляет ряд таблицы умножения соответствующую целому числу переданному в аргументе.

Для начала определим функцию `apply_fun_list`. Пусть `f_list` список функций, тогда вызов `apply_fun_list x f_list` возвращает список результатов применения каждого элемента списка `f_list` к `x`.

```
# let rec apply_fun_list x f_list =
  if null f_list then []
  else ((List.hd f_list) x)::(apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [(+) 1;(+) 2;(+) 3] ;;
- : int list = [2; 3; 4]
```

Функция `mk_mult_fun_list` возвращает список функций умножающих их аргумент на i, `0<=i<=n`.

```
# let mk_mult_fun_list n =
    let rec mmfl_aux p =
      if p = n then [ ( * ) n ]
      else (( * ) p) :: (mmfl_aux (p+1))
    in (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

Подсчитаем ряд для 7

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Итерация в списке

Вызов функции `fold_left f a [e1; e2; ...; en]` возвращает `f...(f (f a e1) e2) ... en`, значит получаем n применений.

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f ( f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

С помощью функции `fold_left` можно компактно определить функцию вычисления суммы элементов списка целых чисел.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Или, например, конкатенация элементов списка строк.

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

2.4 Объявления типов и сопоставление с образцом

С помощью типов определённых в Objective CAML мы можем определять структурные типы, при помощи кортежей или списков. Но в некоторых случаях бывает необходимо определить новые типы для описания специальных структур. В Objective CAML, объявления типов рекурсивны и могут быть параметризованы переменными типа, как в случае `'a list` который мы уже обсуждали. Вывод типов принимает во внимание новые объявления для определения типов выражений. Конструкция значений новых типов использует конструктор описанный определением типа. Особой возможностью языков семейства ML является сопоставление с образцом, которое обеспечивает простой метод доступа к компонентам (полям) структур данных. Определение функции соответствует чаще всего сопоставлению с образцом по одному из ее параметров, что позволяет определять функции для различных случаев.

Для начала, продемонстрируем механизм сопоставления на существующих типах и затем опишем различные объявления структурных типов, конструкций таких переменных и доступ к компонентам по сопоставлению с образцом.

2.4.1 Сопоставление с образцом

Образец (по английски `pattern`) строго говоря не совсем выражение Objective CAML. Речь скорее идёт о корректной компоновке (синтаксической и с точки зрения типов) элементов, таких как константы базовых типов (`int, bool, char` ...), переменные, конструкторы и символ `_`, называемый универсальным образцом. Другие символы служат для записи шаблона, которые мы опишем по ходу.

Сопоставление с образцом применяется к значениям, оно служит для распознания формы этих значений и позволяет определять порядок вычислений. С каждым образцом связано выражение для вычисления.

Синтаксис:

```
match expr with
| p1 -> expr1
:
| pn -> exprn
```

Выражение `expr` последовательно сопоставляется (фильтруется) с разными образцами `p1, ..., pn`. Если один из образцов (например `pi`) соответствует значению `expr`, то соответствующее ответвление будет вычислено (`expri`). Образцы `pi` одинакового типа, так же как и `expri`. Вертикальная черта перед первым образцом не обязательна.

Пример

Приведём два способа, при помощи сопоставления с образцом, определения функции `imply` типа `(bool * bool)->bool`, реализующую логическую импликацию. Образец для пар — `(,)`.

Первая версия перечисляет все возможности, как таблица истинности.

```
# let imply v = match v with
      (true,true) -> true
    | (true,false) -> false
    | (false,true) -> true
    | (false,false) -> true;;
val imply : bool * bool -> bool = <fun>
```

Используя переменные группирующие несколько случаев, мы получаем более компактное определение.

```
# let imply v = match v with
      (true,x) -> x
```

```
    | (false,x) -> true;;
val imply : bool * bool -> bool = <fun>
```

Обе версии `imply` выполняют одну и ту же функцию, то есть они возвращают одинаковые значения, для одинаковых входных аргументов.

Линейный образец

Образец обязательно должен быть линейным, то есть определённая переменная не может быть использована дважды. Мы могли бы написать:

```
# let equal c = match c with
      (x,x) -> true
    | (x,y) -> false;;
Error: Variable x is bound several times in this matching
```

Но для этого компилятор должен уметь реализовывать тесты на равенство, что приводит к множеству проблем. Если мы ограничимся физическим значением переменных, то мы получим слишком «слабую» систему, не в состоянии, например, распознать равенство между двумя списками `[1; 2]`. Если же мы будет проверять на структурное равенство, то рискуем бесконечно проверять циклические структуры. Рекурсивные функции, например, это циклические структуры, но мы так же можем определить рекурсивные значения не являющиеся функциями (см.**??**).

Универсальный образец

Символ `_` совпадает со всеми возможными значениями — он называется универсальным. Этот образец может быть использован для сопоставления сложных типов. Воспользуемся им для определения ещё одной версии функции `imply`:

```
# let imply v = match v with
          (true,false) -> false
        | _ -> true;;
val imply : bool * bool -> bool = <fun>
```

Сопоставление должно обрабатывать все случаи, иначе компилятор выводит сообщение:

```
# let is_zero n = match n with 0 -> true ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
```

```
val is_zero : int -> bool = <fun>
```

В случае если аргумент не равен 0, функция не знает какое значение должно быть возвращено. Для исправления добавим универсальный образец:

```
# let is_zero n = match n with
    0 -> true
  | _ -> false ;;
val is_zero : int -> bool = <fun>
```

Если во время выполнения ни один образец не выбран, возбуждается исключение:

```
# let f x = match x with 1 -> 3 ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val f : int -> int = <fun>
# f 1 ;;
- : int = 3
# f 4 ;;
Exception: Match_failure ("", 77, -33).
```

Исключение `Match_Failure` возбуждено при вызове `f 4` и если оно не обработано, то вычисление останавливается (см. **??**).

Комбинация образцов

Комбинация образцов позволяет получить новый образец, который может разложить значение в соответствии с одним или другим из начальных шаблонов.

Синтаксис

$$p_1 | \ldots | p_n$$

Эта форма создаёт новый образец из комбинации мотивов $p_1, \ldots, p_n$, с единственным ограничением — каждый из этих образцов должен содержать константные значения либо универсальный образец.

Следующий пример показывает как проверить является ли входной символ гласной буквой.

```
# let is_a_vowel c = match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
```

```
  | _ -> false ;;
val is_a_vowel : char -> bool = <fun>
# is_a_vowel 'i' ;;
- : bool = true
# is_a_vowel 'j' ;;
- : bool = false
```

Параметризованное сопоставление

Параметризованное сопоставление используется в основном для определения функций выбора. Для облегчения записи подобных определений, конструктор `function` разрешает следующее сопоставление одного параметра.

Синтаксис

```
function | p1 -> expr1
    | p2 -> expr2
    :
    | pn -> exprn
```

Вертикальная черта перед первым образцом не обязательна. Каждый раз при определении функции, мы используем сопоставление с образцом. Действительно, конструкция `function x -> expression` является определением сопоставления по уникальному образцу одной переменной. Можно использовать эту особенность в простых шаблонах:

```
# let f = function (x,y) -> 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Форма

```
function p1 -> expr1 | ... | pn -> exprn
```

эквивалентна

```
function expr -> match expr with p1 -> expr1 | ... | pn -> exprn
```

Используя схожесть определения на (см. 2.3.4), мы бы написали:

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

Но подобная запись возможна лишь в случае если фильтруемое значение принадлежит к типу с единственным конструктором, иначе сопоставление не является исчерпывающим:

```
# let is_zero 0 = true ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Присвоение имени фильтруемому значению

Иногда при сопоставлении с образцом, бывает необходимо дать имя всему образцу или лишь его части. Следующая синтаксическая форма вводит ключевое слово as, которое ассоциирует имя образцу.

Синтаксис

```
(p as nom)
```

Это бывает полезно когда необходимо разбить значение на части, сохраняя при этом его целостность. В следующем примере функция возвращает наименьшее рациональное число из пары. Каждое рациональное число представлено числителем и знаменателем.

```
# let min_rat pr = match pr with
    ((_,0),p2) -> p2
  | (p1,(_,0)) -> p1
  | (((n1,d1) as r1), ((n2,d2) as r2)) ->
            if (n1 * d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

Для сравнения двух рациональных чисел, необходимо разбить их для именования числителя и знаменателя `(n1, n2, d1 и d2)`, но так же для собирания в одно целое начальные пары (`r1` или `r2`). Конструкция позволяет дать имена частям значения — это избавляет от реконструкции рационального числа при возвращении результата.

Сопоставление с ограничением

Сопоставление с ограничением соответствует вычислению условного выражения сразу после сопоставления с образцом. Если это выражение возвращает значение `true`, то выражение соответствующее этому образцу будет вычислено, иначе сопоставление будет продолжаться дальше.

Синтаксис:

```
match expr with
```

```
:
| pi when condi -> expri
:
```

В следующем примере используется два ограничения для проверки равенства двух рациональных чисел.

```
# let eq_rat cr = match cr with
    ((_,0),(_,0)) -> true
  | ((_,0),_) -> false
  | (_,(_,0)) -> false
  | ((n1,1), (n2,1)) when n1 = n2 -> true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) -> true
  | _ -> false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

Если при фильтрации четвёртого образца ограничение не выполнится, то сопоставление продолжится по пятому образцу.

Предупреждение

При проверке исчерпываемости сопоставления Objective CAML предполагает что условное выражение может быть ложным. В следствии, верификатор не учитывает этот образец, так как не возможно знать до выполнения сработает ограничение или нет. Исчерпываемость следующего фильтра не может быть определена.

```
# let f = function x when x = x -> true;;
Warning 25: bad style, all clauses in this pattern-matching are guarded.
val f : 'a -> bool = <fun>
```

Образец интервала символов

При сопоставлении символов, неудобно описывать комбинацию всех образцов, которая соответствует интервалу символов. Действительно, для того чтобы проверить является ли символ буквой, необходимо написать как минимум 26 образцов и скомбинировать их. Для символьного типа разрешается запись следующего шаблона:

Синтаксис

```
'c1' .. 'cn'
```

что соответствует `'c1' | 'c2' | ...| 'cn'`

К примеру образец

```
'0' .. '9'
```

соответствует образу

```
'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'.
```

Первая форма быстрее и проще воспринимается при чтении.

Предупреждение

Это особенность является расширением языка и может измениться в следующих версиях.

Используя комбинированные образцы и интервалы, напишем функцию по определению символа по нескольким критериям.

```
# let char_discriminate c = match c with
      'a' | 'e' | 'i' | 'o' | 'u' | 'y'
    | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y' -> "Vowel"
    | 'a'..'z' | 'A'..'Z' -> "Consonant"
    | '0'..'9' -> "Digit"
    | _ -> "Other" ;;
val char_discriminate : char -> string = <fun>
```

Заметим, что порядок образцов важен, второе множество содержит первое, но оно проверяется только после неудачи первого теста.

Образцы списков

Как мы уже видели список может быть:

- либо пуст (в форме `[]`)
- либо состоять из первого элемента (заголовок) и под–списка (хвост). В этом случае он имеет форму `t::q`.

Обе эти записи могут быть использованы в образце при фильтрации списка.

```
# let rec size x = match x with
      [] -> 0
    | _::tail_x -> 1 + (size tail_x) ;;
val size : 'a list -> int = <fun>
# size [];;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

Перепишем раннее показанный пример (см. 2.3.6) используя сопоставление с образом для итерации списков.

```
let rec fold_left f a = function
    [] -> a
  | head::tail -> fold_left f (f a head) tail ;;
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

Объявление значения через сопоставлением с образцом

Объявление значений само по себе использует сопоставление. `let x = 18` сопоставляет образцы `x` со значением `18`. Любой образец принимается как фильтр объявления, переменные образца ассоциируются со значениями которые они сопоставляют.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

Видимость переменных фильтра та же самая что у локальных переменных. Здесь `c` ассоциировано с `'A'`.

```
# a + (int_of_char c);;
- : int = 66
```

Как и любой фильтр, определение значения может быть не исчерпывающим.

```
# let [x;y;z] = [1;2;3];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Exception: Match_failure ("", 98, -39).
```

Принимается любой образец, включая конструктор, универсальный и комбинированный.

```
# let head :: 2 :: _ = [1; 2; 3] ;;
Warning 8: this pattern−matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : int = 1
# let _ = 3. +. 0.14 in "PI" ;;
− : string = "PI"
```

Последний пример мало интересен для функционального программирования, поскольку значение 3.14 не именовано, и соответственно будет потерянно.

2.4.2 Декларация типов

Объявление типа — это одна из других возможных фраз синтаксиса Objective CAML. Она позволяет определить новые типы соответствующие обычным структурам данных используемым в программах. Существует два больших семейства типов: тип–произведение для кортежей или записей или тип–сумма для `union`.

Синтаксис:

```
type nom = typedef ;;
```

В отличии от объявления переменных, объявление типов по умолчанию рекурсивно. То есть объявления типов, когда они скомбинированы, могут объявлять взаимно рекурсивные типы.

Синтаксис:

```
type name1 = typedef1
and name2 = typedef2
      :
and namen = typedefn ;;
```

Объявления типов могут быть параметризованы переменной типа. Имя переменной типа начинается всегда с апострофа (символ ').

Синтаксис

```
type 'a nom = typedef ;;
```

В случае когда таких переменных несколько, параметры типа декларируются как кортеж перед именем типа.

Синтаксис

```
type ('a1 ...'an) nom = typedef ;;
```

Только те параметры, которые определены в левой части декларации, могут появится в её правой части.

Замечание Во время вывода на экран Objective CAML переименует параметры типа, первый в `'a`, второй в `'b` и так далее. Мы всегда можем объявить новый тип используя ранее объявленные типы.

Синтаксис:

```
type name = type expression
```

Это полезно в том случае, когда мы хотим ограничить слишком общий тип:

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
```

Без ограничения, выводится наиболее общий тип:

```
# let x = (3, 3.14) ;;
val x : int * float = (3, 3.14)
```

Но мы можем использовать ограничитель типа для того, чтобы получить желаемое имя:

```
# let (x:specific_pair) = (3, 3.14) ;;
val x : specific_pair = (3, 3.14)
```

2.4.3 Записи

Запись — это кортеж, в котором каждому полю присваивается имя на подобии `record` в Pascal или `struct` в C. Запись всегда соответствует объявлению нового типа. Для определения записи необходимо указать её имя, а так же имя и тип для каждого поля записи.

Синтаксис:

```
type nom = { nom1 : t1; ...; nomn : tn } ;;
```

Определим тип комплексного числа следующим образом.

```
# type complex = { re:float; im:float } ;;
type complex = { re : float; im : float; }
```

Для того чтобы создать значение типа запись, нужно присвоить каждому полю значение (в каком угодно порядке).

Синтаксис:

```
{ nomi1 = expri1; ...; nomin = exprin } ;;
```

В следующем примере мы создадим комплексное число, в котором реальная часть равна 2 и мнимая 3:

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re = 2.; im = 3.}
# c = {im=3.;re=2.} ;;
- : bool = true
```

В случае если не хватает некоторых полей, произойдёт следующая ошибка:

```
# let d = { im=4. } ;;
Error: Some record field labels are undefined: re
```

Доступ к полям возможен двумя способами: синтаксис с точкой, либо сопоставлением некоторых полей.

Синтаксис с точкой заключается в следующем:

Синтаксис:

```
expr.nom
```

Выражение `expr` должно иметь тип «запись с полем nom».

Сопоставление записи позволяет получить значение связанное с определённым полям. Синтаксис образца для записи следующий.

Синтаксис

```
{ nomi = pi ; ...; nomj = pj }
```

Образцы находятся справа от символа = `(pi, ..., pj)`. Необязательно перечислять все поля записи в образце.

Функция `add_complex` обращается к полям с помощью «точки», тогда как функция `mult_complex` обращается при помощи фильтра.

```
# let add_complex c1 c2 = {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re = 4.; im = 6.}
# let mult_complex c1 c2 = match (c1,c2) with
   ({re=x1;im=y1},{re=x2;im=y2}) -> {re=x1*.x2-.y1*.y2;im=x1*.y2
       +.x2*.y1} ;;
```

```
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re = -5.; im = 12.}
```

Преимущество записей, по сравнению с кортежами как минимум двойное:

- более информативное описание, благодаря именам полей — это в частности позволяет облегчить образцы фильтра;
- идентичное обращение по имени, порядок значения не имеет — главное указать имя.

```
# let a = (1,2,3) ;;
val a : int * int * int = (1, 2, 3)
# let f tr = match tr with x,_,_ -> x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1 : int; x2 : int; x3 : int; }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1 = 1; x2 = 2; x3 = 3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
# g b ;;
- : int = 1
```

При сопоставлении с образцом не обязательно перечислять все поля записи, тип будет вычислен по последней описанной записи, которая содержит указанные поля.

```
# let h tr = match tr with {x1=x} -> x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

Существует конструкция позволяющая создать идентичную запись, с разницей в несколько полей, что удобно для записей с большим числом полей.

Синтаксис:

```
{ name with namei= expri ; ...; namej=expr_j}
```

```
# let c = {b with x1=0} ;;
val c : triplet = {x1 = 0; x2 = 2; x3 = 3}
```

Новая копия значения `b` отличается только лишь значением поля `x1`.

Предупреждение

Это особенность является расширением языка и может измениться в следующих версиях.

2.4.4 Тип сумма (sum)

В отличии от кортежей или записей, которые соответствуют декартову произведению, тип сумма соответствует объединению множеств (`union`). В один тип мы группируем несколько разных типов (например, целые числа и строки). Различные члены суммы определяются конструкторами, которые с одной стороны позволяют сконструировать значение этого типа и с другой получить доступ к компонентам этих значений при помощи сопоставления с образцом. Применить конструктор к аргументу, значит указать что возвращаемое значение будет этого нового типа.

Тип сумма описывается указыванием имени конструкторов и типа их возможного аргумента.

Синтаксис:

```
type name = ...
    | Namei ...
    | Namej of tj ...
    | Namek of tk * ...* tl ...;;
```

Имя конструктора это специальный идентификатор.

Замечание

Имя конструктора должно всегда начинаться с заглавной буквы.

Константный конструктор

Мы называем константным, конструктор без аргументов. Такие конструктора могут затем использоваться как значения языка, в качестве констант.

(Орел Решка)

```
# type coin = Heads | Tails;;
type coin = Heads | Tails
# Tails;;
- : coin = Tails
```

Тип может быть определён подобным образом.

Конструктор с аргументами

Конструктора могут иметь аргументы, ключевое слово `of` указывает тип аргумента. Это позволяет сгруппировать под одним типом объекты разных типов, имеющих разные конструктора. Определим типы `couleur` и `carte` следующим образом.

```
# type couleur = Pique | Coeur | Carreau | Trefle ;;
type couleur = Pique | Coeur | Carreau | Trefle
# type carte =
      Roi of couleur
    | Dame of couleur
    | Cavalier of couleur
    | Valet of couleur
    | Petite_carte of couleur * int
    | Atout of int
    | Excuse ;;
type carte =
    Roi of couleur
  | Dame of couleur
  | Cavalier of couleur
  | Valet of couleur
  | Petite_carte of couleur * int
  | Atout of int
  | Excuse
```

Создание значения типа `carte` получается применением конструктора к значению с нужным типом.

```
# Roi Pique ;;
- : carte = Roi Pique
# Petite_carte(Coeur, 10) ;;
- : carte = Petite_carte (Coeur, 10)
# Atout 21 ;;
- : carte = Atout 21
```

Следующая функция, `toutes_les_cartes`, создаёт список всех карт цвета указанного аргументом.

```
# let rec interval a b = if a = b then [b] else a::(interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
```

```
# let toutes_les_cartes s =
    let les_figures = [ Valet s; Cavalier s; Dame s; Roi s ]
    and les_autres = List.map (function n -> Petite_carte(s,n)) (
        interval 1 10)
    in les_figures @ les_autres ;;
val toutes_les_cartes : couleur -> carte list = <fun>
# toutes_les_cartes Coeur ;;
- : carte list =
[Valet Coeur; Cavalier Coeur; Dame Coeur; Roi Coeur; Petite_carte (
    Coeur, 1);
 Petite_carte (Coeur, 2); Petite_carte (Coeur, 3); Petite_carte (Coeur, 4);
 Petite_carte (Coeur, 5); Petite_carte (Coeur, 6); Petite_carte (Coeur, 7);
 Petite_carte (Coeur, 8); Petite_carte (Coeur, 9); Petite_carte (Coeur,
     10)]
```

Для манипуляции значений типа сумма, мы используем сопоставление с образцом. В следующем примере опишем функцию преобразующую значения типа `couleur` и типа `carte` в строку (тип `string`):

```
# let string_of_couleur = function
        Pique -> "pique"
    | Carreau -> "carreau"
    | Coeur -> "coeur"
    | Trefle -> "ètrfle" ;;
val string_of_couleur : couleur -> string = <fun>
# let string_of_carte = function
        Roi c -> "roi de " ^ (string_of_couleur c)
    | Dame c -> "dame de " ^ (string_of_couleur c)
    | Valet c -> "valet de " ^ (string_of_couleur c)
    | Cavalier c -> "cavalier de " ^ (string_of_couleur c)
    | Petite_carte (c, n) -> (string_of_int n) ^ " de "^(
        string_of_couleur c)
    | Atout n -> (string_of_int n) ^ " d'atout"
    | Excuse -> "excuse" ;;
val string_of_carte : carte -> string = <fun>
```

Конструктор `Petite_carte` бинарный, для сопоставления такого значения мы должны дать имя обоим компонентам.

```
# let est_petite_carte c = match c with
        Petite_carte v -> true
    | _ -> false;;
Error: The constructor Petite_carte expects 2 argument(s),
```

```
        but is applied here to 1 argument(s)
```

Чтобы не давать имя каждой компоненте конструктора, объявим его с одним аргументом, заключив в скобки тип ассоциированного кортежа. Сопоставление двух следующих конструкторов отличается.

```
# type t =
        C of int * bool
    | D of (int * bool) ;;
type t = C of int * bool | D of (int * bool)
# let acces v = match v with
        C (i, b) -> i,b
    | D x -> x;;
val acces : t -> int * bool = <fun>
```

2.4.5 Рекурсивный тип

Определение рекурсивного типа необходимо в любом языке программирования, с его помощью мы можем определить такие типы как список, стек, дерево и так далее. В отличии от объявления `let`, `type` всегда рекурсивный в Objective CAML.

Типу список, определённому в Objective CAML, необходим один единственный аргумент. Для того чтобы хранить значения двух разных типов, как например `int` или `char` мы напишем:

```
# type int_or_char_list =
        Nil
    | Int_cons of int * int_or_char_list
    | Char_cons of char * int_or_char_list ;;
type int_or_char_list =
    Nil
  | Int_cons of int * int_or_char_list
  | Char_cons of char * int_or_char_list
# let l1 = Char_cons ( '=', Int_cons(5, Nil) ) in
    Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) ) ;;
- : int_or_char_list =
Int_cons (2,
 Char_cons ('+', Int_cons (3, Char_cons ('=', Int_cons (5, Nil)))))
```

2.4.6 Параметризованный тип

Мы также можем определить тип с параметрами, что позволит нам обобщить предыдущий список для двух любых типов.

```
# type ('a, 'b) list2 =
        Nil
    | Acons of 'a * ('a, 'b) list2
    | Bcons of 'b * ('a, 'b) list2 ;;
type ('a, 'b) list2 =
    Nil
  | Acons of 'a * ('a, 'b) list2
  | Bcons of 'b * ('a, 'b) list2
# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil))))) ;;
- : (int, char) list2 =
Acons (2, Bcons ('+', Acons (3, Bcons ('=', Acons (5, Nil)))))
```

Естественно, оба параметра `'a` и `'b` могут быть одного типа:

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil)))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bcons (4, Nil))))
```

Как в предыдущем примере, мы можем использовать тип `list2` чтобы пометить чётные и нечётные числа. Что бы создать обычный список, достаточно извлечь под–список чётных чисел.

```
# let rec extract_odd = function
      Nil -> []
    | Acons(_, x) -> extract_odd x
    | Bcons(n, x) -> n::(extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

Определение этой функции ничего не говорит о свойстве значений хранящихся в структуре, по этой причине её тип параметризован.

2.4.7 Видимость описания

На имена конструкторов распространяются те же правила, что и на глобальные объявления. Переопределение скрывает своего предшественника. Скрытые значения всегда существуют, они никуда не делись. Однако цикл не сможет различить эти оба типа, поэтому мы получим не совсем понятное сообщение об ошибке.

В следующем примере, константный конструктор `Nil` типа `int_of_char` скрывается объявлением конструктора `('a, 'b) list2`.

```
# Int_cons(0, Nil) ;;
Error: This expression has type ('a, 'b) list2
       but an expression was expected of type int_or_char_list
```

Во втором примере мы получим совсем бестолковое сообщение, по крайней мере на первый взгляд. Пусть мы имеем следующую программу:

```
# type t1 = Vide | Plein;;
type t1 = Vide | Plein
# let vide_t1 x = match x with Vide -> true | Plein -> false ;;
val vide_t1 : t1 -> bool = <fun>
# vide_t1 Vide;;
- : bool = true
```

Затем переопределим тип `t1`:

```
# type t1 = {u : int; v : int} ;;
type t1 = { u : int; v : int; }
# let y = { u=2; v=3 } ;;
val y : t1 = {u = 2; v = 3}
```

Теперь если мы применим функцию `empty_t1` к значению с новым типом `t1`, то получим следующее сообщение.

```
# vide_t1 y;;
Error: This expression has type t1/1468
       but an expression was expected of type t1/1461
```

Первое упоминание типа `t1` соответствует ранее определённому типу, тогда как второе — последнему.

2.4.8 Функциональные типы

Тип аргумента конструктора может быть каким угодно, и в частности он может быть функциональным. Следующий тип создаёт список состоящий из функциональных значений, кроме последнего элемента.

```
# type 'a listf =
    Val of 'a
  | Fun of ('a -> 'a) * 'a listf ;;
type 'a listf = Val of 'a | Fun of ('a -> 'a) * 'a listf
```

Поскольку функциональные значения входят во множество значений которые обрабатываются языком, то мы можем создать значения типа `listf`:

```
# let huit_div = (/) 8 ;;
val huit_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (huit_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

функция сопоставляющая подобные значения:

```
# let rec compute = function
    Val v -> v
  | Fun(f, x) -> f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3
```

2.4.9 Пример: реализация деревьев

Деревья — одна из часто встречающихся структур в программировании. Рекурсивные типы позволяют с лёгкостью определять подобные вещи. В этой части мы приведём два примера с такими структурами.

Бинарные деревья

Определим дерево, в котором узлы обозначены значениями одного и того же типа:

```
# type 'a arbre_bin =
    Empty
  | Node of 'a arbre_bin * 'a * 'a arbre_bin ;;
type 'a arbre_bin = Empty | Node of 'a arbre_bin * 'a * 'a arbre_bin
```

Этой структурой мы воспользуемся в программе сортировки бинарных деревьев. Бинарное дерево имеет следующее свойство: значение левого дочернего узла меньше корневого и всех значений правых дочерних узлов. Рис. 2.1 показывает пример такой структуры с целыми числами. Пустые узлы (конструктор `Empty`) представлены небольшими прямоугольниками; остальные (конструктор `Node`) представлены окружностями, в которых показаны значения.

Извлечём значения узлов в виде сортированного списка при помощи инфиксного просмотра следующей функцией:

```
# let rec list_of_tree = function
    Empty -> []
```

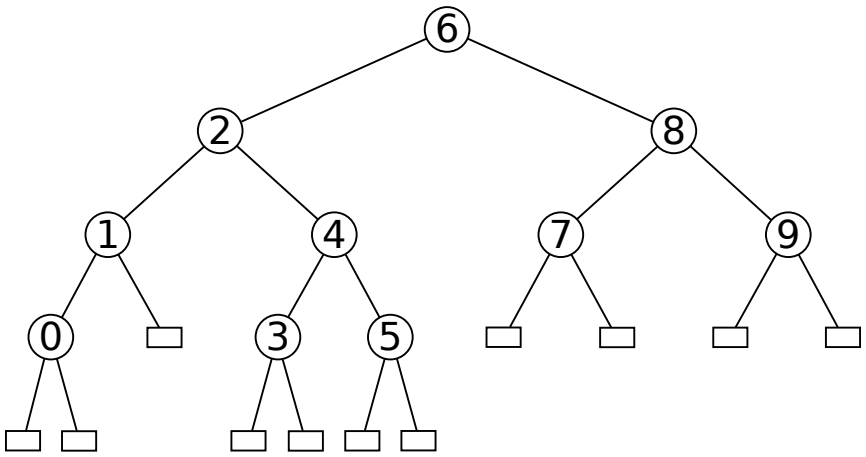

Рис. 2.1: Бинарное дерево поиска

```
  | Node(lb, r, rb) -> (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a arbre_bin -> 'a list = <fun>
```

Для того чтобы получить из списка бинарное дерево, определим функцию:

```
# let rec ajout x = function
    Empty -> Node(Empty, x, Empty)
  | Node(fg, r, fd) -> if x < r then Node(ajout x fg, r, fd)
                       else Node(fg, r, ajout x fd) ;;
val ajout : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

Функция трансформирующая список в бинарное дерево может быть получена использованием функции `ajout`

```
# let rec arbre_of_list = function
      [] -> Empty
  | t::q -> ajout t (arbre_of_list q) ;;
val arbre_of_list : 'a list -> 'a arbre_bin = <fun>
```

Тогда функция сортировки это всего–навсего композиция функций `arbre_of_list` и `list_of_arbre`.

```
# let tri x = list_of_arbre (arbre_of_list x) ;;
val tri : 'a list -> 'a list = <fun>
# tri [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
```

```
− : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

Общие планарные деревья

Воспользуемся функцией определённой в модуле List (см. **??**):

- `List.map`: которая применяет одну функцию ко всем элементам списка и возвращает список результатов.
- `List.fold_left`: эквивалент функции `fold_left` определённой на стр. 2.4.1
- `List.exists` применяет функцию булевого значения ко всем элементам и если одно из применений возвращает `true`, то результат `true`, иначе `false`.

Общее планарное дерево — это дерево в котором нет ограничения на число дочерних узлов: каждому узлу ассоциируем список дочерних узлов и длина этого списка не ограничена.

```
# type 'a arbre = Empty
                | Node of 'a * 'a arbre list ;;
type 'a arbre = Empty | Node of 'a * 'a arbre list
```

Пустое дерево представлено значением `Empty`, лист — это узел без дочерних узлов формы `Node(x,[])` или `Node(x, [ Empty ; Empty; .. ])`. Таким образом достаточно просто написать функции, манипулирующие подобными деревьями такие как принадлежность элемента дереву или вычисление высоты дерева.

Для проверки принадлежности элемента `e` воспользуемся следующим алгоритмом: если дерево пустое, тогда элемент `e` не принадлежит этому дереву, в противном случае `e` принадлежит дереву только если он равен значению корня дерева или принадлежит дочерним узлам.

```
val belongs : 'a -> 'a arbre -> bool = <fun>
# let rec appartient e = function
    Empty -> false
  | Node(v, fs) -> (e=v) or (List.exists (appartient e) fs) ;;
val appartient : 'a -> 'a arbre -> bool = <fun>
```

Для вычисления высоты дерева, напишем следующую функцию: высота пустого дерева равна 0, в противном случае его высота равна высоте самого большого под–дерева плюс 1.

```
# let rec hauteur =
  let max_list l = List.fold_left max 0 l in
   function
     Empty -> 0
   | Node (_, fs) -> 1 + (max_list (List.map hauteur fs)) ;;
val hauteur : 'a arbre -> int = <fun>
```

2.4.10 Не функциональные рекурсивные значения

Рекурсивное объявление не функциональных значений позволяет конструировать циклические структуры данных.

Следующее объявление создаёт циклический список одного элемента.

```
# let rec l = 1::l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;
   ...]
```

Применение рекурсивной функции к подобному списку приведет к переполнению памяти.

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

Структурное равенство таких списков может быть проверенно лишь в случае, когда физическое равенство верно.

```
l=l ;;
- : bool = true
```

В случае, если мы определим такой же новый список, не стоит использовать проверку на структурное равенство, под угрозой зацикливания программы. Таким образом следующая инструкция не рекомендуется.

```
let rec l2 = 1::l2 in l=l2 ;;
```

Однако, физическое равенство остаётся возможным.

```
# let rec l2 = 1::l2 in l==l2 ;;
- : bool = false
```

Предикат `==` сравнивает непосредственно значение или разделения структурного объекта (одним словом равенство по адресу). Мы воспользуемся этим тестом для того чтобы при просмотре списка, не проверять уже просмотренные под-списки. Сначала, определим функцию `memq` которая проверяет присутствие элемента в списке при помощи физического равенства. В это же время функция `mem` проверяет равенство структурное. Обе функции принадлежат модулю `List`.

```
# let rec memq a l = match l with
    [] -> false
  | b::l -> (a==b) or (memq a l) ;;
val memq : 'a -> 'a list -> bool = <fun>
```

Функция вычисляющая размер списка определена при помощи списка уже проверенных списков и останавливается при встрече списка второй раз.

```
# let special_size l =
  let rec size_aux previous l = match l with
      [] -> 0
    | _::l1 -> if memq l previous then 0
                          else 1 + (size_aux (l::previous) l1)
  in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1::2::l2 and l2 = 1::2::l1 in special_size l1 ;;
- : int = 4
```

2.5 Типизация, область определения и исключения

Вычисленный тип функции принадлежит подмножеству множества в котором она определена. Если входной аргумент функции типа `int`, это не значит что она сможет быть выполнена для любого переданного ей целого числа. Таких случаях мы используем механизм исключений Objective CAML. Запуск исключения провоцирует остановку вычислений, которое может быть выявлено и обработано. Для этого необходимо установить обработчик исключения, до того как исключение может быть возбуждено.

2.5.1 Частичные функции и исключения

Область определения функции соответствует множеству значений которыми функция манипулирует. Существует множество математических частичных функций, таких как например, деление или натуральный логарифм. Проблемы возникают в том случае, когда функция манипулирует более сложными структурами данных. Действительно, каков будет результат функции определяющей первый элемент списка, если список пуст. Аналогично, вычисление факториала для отрицательного числа приводит к бесконечному вычислению.

Определённое число исключительных ситуаций может произойти во время выполнения программы, как например деление на ноль. Попытка деления на ноль может привести в лучшем случае к остановке программы и в худшем к неправильному (некоэрантному) состоянию машины. Безопасность языка программирования заключается в гарантии того, что такие случаи не возникнут. Исключение есть один из способов такой гарантии.

Деление 1 на 0 провоцирует запуск специального исключения:

```
# 1/0;;
Exception: Division_by_zero.
```

Сообщение `Exception: Division_by_zero` указывает во первых на то, что исключение `Division_by_zero` возбуждено и во вторых, что оно не было обработано.

Часто, тип функции не соответствует области ее определения в том случае когда сопоставление с образцом не является полным, то есть когда он не отслеживает все возможные случаи. Чтобы избежать такой ситуации, Objective CAML выводит следующее сообщение.

```
# let tete l = match l with t::q -> t ;;
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val tete : 'a list -> 'a = <fun>
```

Однако, если программист все таки настаивает на своей версии, то Objective CAML воспользуется механизмом исключения в случае неправильного вызова частичной функции.

```
# tete [] ;;
Exception: Match_failure ("", 9, 9).
```

Мы уже встречали другое исключение определенное в Objective CAML: `Failure`, с входным аргументом типа `string`. Запустить это исключение можно функцией `failwith`. Воспользуемся этим и дополним определение функции `tete`.

2.5.2 Определения исключения

В Objective CAML, тип исключений — `exn`. Это особенный тип — расширяемая сумма: мы можем увеличить множество значений этого типа объявляя новые конструкторы. Такая особенность позволяет программисту определять свои собственные исключения.

Синтаксис объявления следующий:

Синтаксис:

```
exception Nom ;;
```

Синтаксис:

```
exception Nom of t ;;
```

Вот несколько примеров объявления исключений.

```
# exception A_MOI;;
exception A_MOI
# A_MOI;;
- : exn = A_MOI
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth 4
```

Имена исключений являются конструкторами — это значит они должны начинаться с заглавной буквы.

```
# exception minuscule ;;
Error: Syntax error
```

Предупреждение

Исключения мономорфны, при объявлении типа аргумента мы не можем указать параметризующий тип.

```
exception Value of 'a ;;
Error: Unbound type parameter 'a
```

Полиморфное исключение позволило бы определение функций, возвращающих результат любого типа. Подобный пример мы рассмотрим далее на **??**.

2.5.3 Возбуждение исключения

`raise` — функциональный примитив языка Objective CAML, её входной аргумент есть исключение, а возвращает он полиморфный тип.

```
# raise ;;
- : exn -> 'a = <fun>
# raise A_MOI;;
Exception: A_MOI.
# 1+(raise A_MOI);;
Exception: A_MOI.
# raise (Depth 4);;
Exception: Depth 4.
```

В Objective CAML нет возможности написать функцию `raise`, она должна быть определена системой.

2.5.4 Перехват исключения

Весь интерес возбуждения исключений содержится в возможности их обработать и изменить выполнение программы в зависимости от этого исключения. В этом случае, порядок вычисления выражения имеет значение для того чтобы определить какое исключения было запущенно. Таким образом мы выходим из рамок чисто функционального программирования, потому как порядок вычисления аргументов может изменить результат. Об этом мы поговорим в следующей главе (стр. **??**)

Следующая конструкция, вычисляющая значение какого–то выражения, отлавливает исключение, возбуждённое во время этого вычисления.

Синтаксис:

```
try expr with
  | p1 -> expr1
  :
  | pn -> exprn
```

Если вычисление `expr` не возбудит исключения, то тип результата будет типом подсчитываемым этим выражением. Иначе, в приведённом выше сопоставлении будет искаться ответвление соответствующее этому исключению и будет возвращено значение следующего за ним выражения.

Если ни одно ответвление не соответствует исключению, то оно будет переправлено до следующего обработчика `try-with` установленного в программе. Таким образом сопоставление исключения всегда исчерпывающий. Подразумевается что последний фильтр `|e->raise e`. Если ни одного обработчика не найдено в программе, система сама займётся обработкой этого исключения и остановит программу с выводом сообщения об ошибке.

Важно понимать разницу между вычислением исключения (то есть значение типа `exn`) и его обработкой, которое приостанавливает вычисления. Исключение, как и любой другой тип, может быть возвращено функцией.

```
# let rendre x = Failure x ;;
val rendre : string -> exn = <fun>
# rendre "test" ;;
- : exn = Failure "test"
# let declencher x = raise (Failure x) ;;
val declencher : string -> 'a = <fun>
# declencher "test" ;;
Exception: Failure "test".
```

Как вы наверно заметили, функция `declencher` (запустить) не возвращает значение типа `exn`, тогда как `rendre` (вернуть) возвращает.

Вычисления с исключениями

Кроме обработки не желаемых значений, исключения можно использовать для оптимизации. Следующий пример реализует произведение всех элементов списка целых чисел. Мы воспользуемся исключением для

остановки просмотра списка если обнаружим 0, который мы вернём как результат функции.

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
      [] -> 1
    | 0 :: _ -> raise Found_zero
    | n :: x -> n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
    try mult_rec l with Found_zero -> 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
- : int = 0
```

Оставшиеся вычисления, то есть произведения элементов после встреченного нуля, не выполняются. После встречи `raise`, вычисление вновь продолжится с `with`.

2.6 Полиморфизм и значения возвращаемые функциями

Полиморфизм Objective CAML позволяет определить функции, тип возвращаемого выражения которых конкретно не определён. Например:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

Однако, возвращаемый тип `id` зависит от типа аргумента. Таким образом, если мы применим функцию `id` к какому-нибудь аргументу, то механизм вывода (определения) типа сможет реализовать переменную типа `'a`. Для каждого частного использования тип функции `id` может быть определён.

Иначе, использование жёсткой статической типизации, которая обеспечивает правильность выполнения, не будет иметь смысла. Функция с полностью неопределённым типом как `'a->'b`, разрешит какое попало преобразование типа, что привело бы к ошибке выполнения, так как физическое представление значений не одно и то же.

Очевидное противоречие

Мы можем определить функции, которые возвращают значение, тип которого не соответствует типу аргументов. Рассмотрим несколько примеров и проанализируем почему это не противоречит жёсткой статической типизации.

Вот первый пример:

```
# let f x = [] ;;
val f : 'a -> 'b list = <fun>
```

Эта функция конструирует полиморфные значения из любого типа.

```
# f () ;;
- : 'a list = []
# f "n'importe quoi" ;;
- : 'a list = []
```

Однако, полученное значение не совсем является не определённым — подразумевается список. Таким образом она не может использоваться где попало.

Вот три примера функций типа `'a -> 'b`:

```
# let rec f1 x = f1 x ;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "n'importe quoi" ;;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [] ;;
val f3 : 'a -> 'b = <fun>
```

Эти функции не являются «опасными» в отношении правильности выполнения программы, так как их невозможно использовать для конструкции значений: первая зацикливается, две последние запускают исключение, которые останавливают выполнение программы.

Чтобы не было возможности определить функции типа `'a -> 'b`, новые конструкторы исключений не должны иметь аргументы типы которых содержат переменную.

Если бы могли объявить полиморфное исключение `Poly_exn` типа `'a -> exn`, то тогда следующая функция была бы возможна:

```
let f = function
    0 -> raise (Poly_exn false)
  | n -> n+1 ;;
```

Таким образом, тип функции `f` это `int->int` и тип `Poly_exn` это `'a -> exn`, теперь напишем следующее:

```
let g n = try f n with Poly_exn x -> x+1 ;;
```

Это правильно типизированная функция (поскольку аргумент `Poly_exn` может быть каким угодно) и вызов `g 0` попытается сложить целое с булевым значением!

2.7 Калькулятор

Для того чтобы понять как организуются программы на Objective CAML, необходимо самим реализовать такую программу. Напишем калькулятор манипулирующий целыми числами при помощи 4 стандартных операций.

Для начала, определим тип `key`, для представления кнопки калькулятора. Всего таких кнопок будет 15: по одной для каждой операции, для каждой цифры и для кнопки равно.

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int ;;
```

Заметим, что цифровые кнопки сгруппированы под одним конструктором `Digit` с аргументом целого типа. Таким образом, некоторые значения типа `key` не являются на самом деле кнопкой. Например, (`Digit 32`) есть значение типа `key`, но не представляет ни одну кнопку калькулятора.

Напишем функцию `valid` которая проверяет входной аргумент на принадлежность к типу `key`. Тип функции `key->bool`.

На первом этапе определим функцию проверяющую что число принадлежит интервалу 0-9. Объявим её с именем `is_digit`.

```
# let est_chiffre = function x -> (x>=0) && (x<=9) ;;
val est_chiffre : int -> bool = <fun>
```

Теперь функция `valid` фильтрующая свой аргумент тип которого `key`:

```
let valide tch = match tch with
      Chiffre n -> est_chiffre n
   | _ -> true ;;
val valide : touche -> bool = <fun>
```

Первый фильтр применяется в случае если входной аргумент построен конструктором `Digit`, в этом случае он проверяется функцией

`is_digit`. Второй фильтр применяется в любом другом случае. Напомним, что благодаря фильтру, тип фильтруемого значения обязательно будет `key`.

Перед тем как начать реализовывать алгоритм калькулятора, определим модель формально описывающую реакцию на нажатие кнопок. Мы подразумеваем что калькулятор располагает памятью в которой хранится результат последней операции, последняя нажатая кнопка, последний использованный оператор и число выведенное на экран. Все эти 4 значения назовем состоянием калькулятора, оно изменяется каждый раз при нажатии на одну из кнопок. Это изменение называется переходом, а теория управляющая подобным механизмом есть теория автоматов.

Состояние представлено следующим типом:

```
type etat = {
   dce : int; (* dernier calcul éeffectu *)
   dta : touche; (* èdernire touche éactionne *)
   doa : touche; (* dernier éoprateur éactionn *)
   vaf : int (* valeur éaffiche *)
 } ;;
```

В таблице 2.7 приведён пример состояний калькулятора.

Таблица 2.5: Переходы 3+21*2=

	Состояние	Значение
	$(0, =, =, 0)$	3
$\rightarrow$	$(0, 3, =, 3)$	+
$\rightarrow$	$(3, +, +, 3)$	2
$\rightarrow$	$(3, 2, +, 2)$	1
$\rightarrow$	$(3, 1, +, 21)$	*
$\rightarrow$	$(24, *, *, 24)$	2
$\rightarrow$	$(24, 2, *, 2)$	=
$\rightarrow$	$(48, =, =, 48)$	

Нам нужна функция `evaluate` с тремя аргументами: двумя операндами и оператором, она возвращает результат операции. Для этого функция фильтрует входной аргумент типа `key`:

```
# let evalue x y tch = match tch with
    Plus -> x + y
  | Moins -> x - y
  | Fois -> x * y
```

```
    | Par -> x / y
    | Egal -> y
    | Chiffre _ -> failwith "evalue : no op";;
val evalue : int -> int -> touche -> int = <fun>
```

Дадим определение переходам, перечисляя все возможные случаи. Предположим, что текущее состояние есть квадри–плет `(a, b, A, d)`:

- кнопка с цифрой x нажата, есть два возможных случая:
  - предыдущая нажатая кнопка тоже цифра, значит пользователь продолжает набирать число и необходимо добавить `x` к значению выводимому на экран то есть изменить его на `d * 10 + x`. Новое состояние будет: `(a,(Digit x), A, d * 10 + x)`
  - предыдущая нажатая кнопка не цифра, тогда мы только начинаем писать новое число. Новое состояние: `(a, (Digit x), A, x)`
- кнопка с оператором B была нажата, значит второй операнд был полностью введён и теперь мы хотим «что-то» сделать с двумя операндами. Для этого сохранена последняя операция (`A`). Новое состояние: `(A d,B,B,a A d)`

Для того чтобы написать функцию `transition`, достаточно дословно перевести в Objective CAML предыдущее описание при помощи сопоставления входного аргумента. Кнопку с двумя значениями обработаем локальной функцией `digit_transition` при помощи сопоставления.

```
# let transition et tou =
  let transition_chiffre n = function
      Chiffre _ -> { et with dta=tou; vaf=et.vaf*10+n }
    | _ -> { et with dta=tou; vaf=n }
  in
    match tou with
        Chiffre p -> transition_chiffre p et.dta
      | _ -> let res = evalue et.dce et.vaf et.doa
                       in { dce=res; dta=tou; doa=tou; vaf=res } ;;
val transition : etat -> touche -> etat = <fun>
```

Эта функция из `state` и `key` считает новое состояние `state`.

Протестируем теперь программу:

```
# let etat_initial = { dce=0; dta=Egal; doa=Egal; vaf=0 } ;;
val etat_initial : etat = {dce=0; dta=Egal; doa=Egal; vaf=0}
# let etat2 = transition etat_initial (Chiffre 3) ;;
val etat2 : etat = {dce=0; dta=Chiffre 3; doa=Egal; vaf=3}
# let etat3 = transition etat2 Plus ;;
val etat3 : etat = {dce=3; dta=Plus; doa=Plus; vaf=3}
# let etat4 = transition etat3 (Chiffre 2) ;;
val etat4 : etat = {dce=3; dta=Chiffre 2; doa=Plus; vaf=2}
# let etat5 = transition etat4 (Chiffre 1) ;;
val etat5 : etat = {dce=3; dta=Chiffre 1; doa=Plus; vaf=21}
# let etat6 = transition etat5 Fois ;;
val etat6 : etat = {dce=24; dta=Fois; doa=Fois; vaf=24}
# let etat7 = transition etat6 (Chiffre 2) ;;
val etat7 : etat = {dce=24; dta=Chiffre 2; doa=Fois; vaf=2}
# let etat8 = transition etat7 Egal ;;
val etat8 : etat = {dce=48; dta=Egal; doa=Egal; vaf=48}
```

Мы можем сделать тоже самое, передав список переходов.

```
# let transition_liste et lt = List.fold_left transition et lt ;;
val transition_liste : etat -> touche list -> etat = <fun>
# let exemple = [ Chiffre 3; Plus; Chiffre 2; Chiffre 1; Fois; Chiffre 2; Egal
    ]
 in transition_liste etat_initial exemple ;;
- : etat = {dce=48; dta=Egal; doa=Egal; vaf=48}
```

2.8 Exercises

2.9 Резюме

В этой главе мы изучили основы функционального программирования и параметризованный полиморфизм, что вместе являются основными характеристиками Objective CAML. Также мы описали синтаксис функциональной части ядра и типов. Это дало нам возможность написать первые программы. Так же мы подчеркнули глубокую разницу между типом функции и областью её применения. Механизм исключений позволяет решить эту проблему, и в то же время вводит новый стиль программирования, где мы явно указываем манеру вычисления.

2.10 To learn more

Глава 3

Императивное программирование

3.1 Введение

В отличие от функционального программирования, где значение вычисляется посредством применения функции к аргументам, не заботясь о том как это происходит, императивное программирование ближе к машинному представлению, так как оно вводит понятие состояния памяти, которое изменяется под воздействием программы. Каждое такое воздействие называется инструкцией и императивная программа есть набор упорядоченных инструкций. Состояние памяти может быть изменено при выполнении каждой инструкции. Операции ввода/вывода можно рассматривать как изменение оперативной памяти, видео памяти или файлов.

Подобный стиль программирования напрямую происходит от программирования на ассемблере. Мы встречаем этот стиль в языках программирования первого поколения (*Fortran, C, Pascal*, etc). Следующие элементы Objective CAML соответствуют приведённой модели:

- физически изменяемые [1] структуры данных, такие как массив или запись с изменяемыми (mutable) полями
- операции ввода/вывода
- структуры контроля выполнения программы как цикл и исключения.

[1] в оригинальном издании (французском) используется выражение физически изменяемые, тогда как в английском переводе лишь изменяемые

Некоторые алгоритмы реализуются проще таким стилем программирования. Примером может послужить произведение двух матриц. Несмотря на то что эту операцию можно реализовать чисто функциональным способом, при этом списки заменят массивы, это не будет ни эффективнее, ни естественней по отношению к императивному стилю.

Интерес интеграции императивной модели в функциональный язык состоит в написании при необходимости подобных алгоритмов в этом стиле программирования. Два главных недостатка императивного программирования по отношению к функциональному это:

- усложнение системы типов языка и отклонение (rejecting) некоторых программ, которые бы без этого рассматривались бы как «корректные»,
- необходимость учитывать состоянием памяти и порядок вычисления.

Однако, при соблюдении некоторых правил написания программ, выбор стиля программирования предоставляет большие возможности в написании алгоритмов, что является главной целью языков программирования. К тому же, программа написанная в стиле близкому к стилю алгоритма, имеет больше шансов быть корректной (или по крайней мере быстрее реализована).

По этим причинам в Objective CAML имеются типы данных, значения которых физически изменяемые, структуры контроля выполнения программ и библиотека ввода/вывода в императивном стиле.

3.2 План главы

В данной главе продолжается представление базовых элементов Objective CAML затронутых в предыдущей главе, но теперь мы заинтересуемся императивными конструкциями. Глава разбита на 5 разделов. Первый, наиболее важный, раскрывает различные физически изменяемые структуры данных и описывает их представление в памяти. Во втором разделе кратко излагаются базовые операции ввода/вывода. Третий знакомит с новыми итеративными структурами контроля. В четвёртом разделе рассказывается об особенностях выполнения императивных программ, в частности о порядке вычисления аргументов функции. В последнем разделе мы переделаем калькулятор предыдущей главы в калькулятор с памятью.

3.3 Физически изменяемые структуры данных

Значения следующих типов: массивы, строки, записи с модифицируемыми полями и указатели есть структурированные значения, компоненты которых могут быть физически изменены.

Мы уже видели, что переменная в Objective CAML привязана к значению и хранит эту связку до конца её существования. Мы можем изменить связку лишь переопределением, но и в этом случае речь не будет идти о той же самой переменной. Новая переменная с тем же именем скроет предыдущую, которая перестанет быть доступной напрямую и останется неизменной. В случае с изменяемыми переменными, мы можем ассоциировать новое значение переменной без её переопределения. Значение переменной доступно в режиме чтение/запись.

3.3.1 Векторы

Векторы или одномерные массивы группируют определённое число однотипных элементов. Существует несколько способов создания векторов, первый — перечисление элементов, разделённых запятой и заключённых между символами [| и |] .

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

Функция `Array.create` с двумя аргументами на входе: размер вектора и начальное значение, возвращает новый вектор.

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

Для того чтобы изменить или просто просмотреть значение элемента необходимо указать его номер.

Синтаксис

```
expr1 . ( expr2 )
```

Синтаксис

```
expr1 . ( expr2 ) <- expr3
```

`expr1` должен быть вектором (тип `array`) с элементами типа `expr3`. Конечно, выражение `expr2` должно быть типа `int`. Результат модификации есть выражение типа `unit`. Номер первого элемента вектора 0 и

последнего — размер вектора минус 1. Скобки вокруг выражения обязательны.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100.; 3.14; 3.14|]
```

Если индекс элемента не принадлежит интервалу индексов вектора, то возбуждается исключения в момент доступа.

```
# v.(-1) +. 4.0;;
Exception: Invalid_argument "index out of bounds".
```

Эта проверка осуществляется в момент выполнения программы, что может сказаться на её быстроте. Однако, это необходимо для избежания заполнения зоны памяти вне вектора, что может привести к серьёзным ошибкам.

Функции манипуляции векторами являются частью модуля `Array` стандартной библиотеки, описание которого дано в главе **??**. В следующих примерах мы воспользуемся тремя функциями из этого модуля:

- create создающая вектор заданного размера и начальным значением
- возвращающая размер вектора
- для конкатенации двух векторов

Совместное использование значений вектора

Все элементы вектора содержат значение, переданное во время создания вектора. В случае, если это значение структурное, оно будет совместно использоваться (sharing). Создадим, для примера, матрицу как вектор векторов при помощи функции `Array.create`:

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

Если поменять одно из полей вектора `v`, который мы использовали для создания `m`, мы изменим все строки матрицы (см. рисунки 3.1 и 3.2).

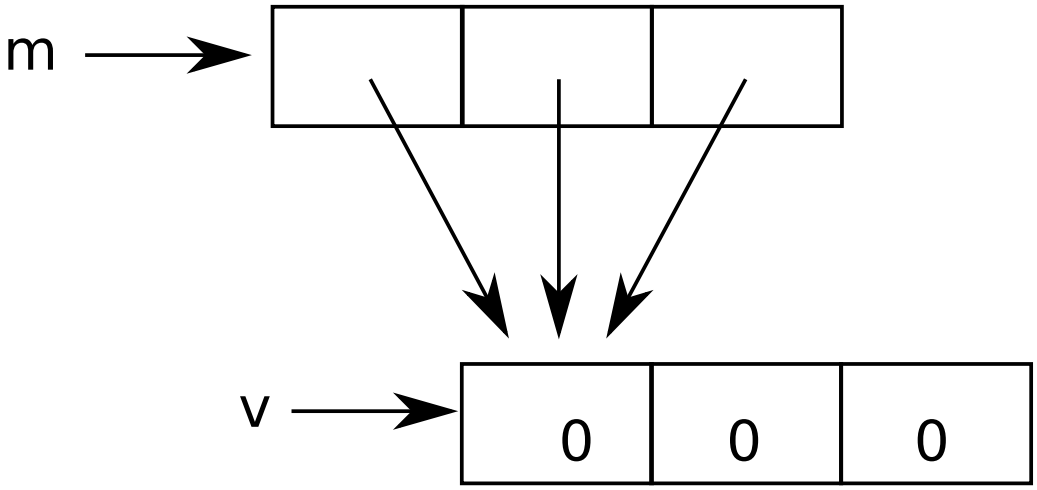

Рис. 3.1: Представление в памяти вектора с разделением элементов

```
# v.(0) <− 1;;
− : unit = ()
# m;;
− : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

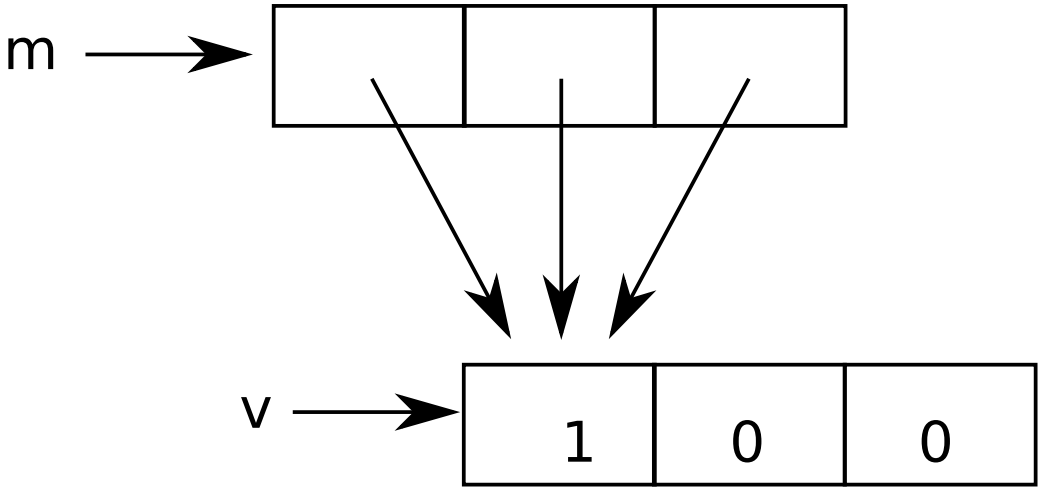

Рис. 3.2: Представление в памяти вектора с разделением элементов

Если значение инициализации вектора (второй аргумент функции `Array.create`) простое, атомарное, то оно копируется каждый раз и совместно используется если это значение структурное.

Мы называем атомарным значением то, размер которого не превышает стандартный размер значений Objective CAML, то есть `memory word` — целое, символ, булевый и конструкторы констант. Другие значения, называемые структурные, представлены указателем на зону памяти. Эта разница объяснена более детально в главе **??**.

Векторы чисел с плавающей запятой есть особый случай. Несмотря на то что эти числа (`float`) — структурные значения, при создании вектора начальное значение копируется. Делается это для оптимизации. В главе **??**, в которой обсуждается интерфейс с языком C мы обсудим эту проблему.

Не квадратная матрица

Матрица, вектор векторов, не обязательно должна быть квадратной. Действительно, мы можем заменить вектор на другой вектор с иным размером, что удобно для ограничения размера матрицы. Следующее значение `t` создаёт треугольную матрицу для коэффициентов треугольника Паскаля.

```
# let t = [|
            [|1|];
            [|1; 1|];
            [|1; 2; 1|];
            [|1; 3; 3; 1|];
            [|1; 4; 6; 4; 1|];
            [|1; 5; 10; 10; 5; 1|]
          |] ;;
val t : int array array =
  [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; 1|];
    [|1; 5; 10; 10; 5; 1|]|]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

В этом примере элемент вектора `t` по индексу `i` есть вектор целых чисел размером `i + 1`. Для манипуляции такой матрицей необходимо знать размер каждого элемента вектора.

Копия векторов

При копировании вектора или конкатенации двух векторов мы получаем новый вектор. Модификация исходных векторов не повлияет на значение копий, кроме случая разделения значений рассмотренного ранее.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

В приведённом примере видно что копия m содержит лишь указатели на `v`, если один из элементов `v` изменён, то `m2` тоже будет изменён. Конкатенация создаёт новый вектор длина которого равна сумме длин двух других.

```
# let mm = Array.append m m ;;
val mm : int array array =
  [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
    [|1; 352; 0|]; [|1; 352; 0|]|]
# Array.length mm ;;
- : int = 6
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array = [|[|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
# mm ;;
- : int array array =
[|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
  [|1; 352; 0|]|]
```

Однако, изменение `v`, разделяемого `m` и `mm`, приведёт к изменению обоих матриц:

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
[|[|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|];
  [|1; 18; 0|]|]
```

3.3.2 Строки

Строки можно рассматривать как частный случай массива символов. Однако, по причинам использования памяти этот тип специализирован и синтаксис доступа к элементам изменён.

Синтаксис

```
expr1 . [expr2]
```

Элементы строки могут быть физически изменены.

Синтаксис

```
expr1 . [expr2] <- expr3
```

```
# let s = "hello";;
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

3.3.3 Изменяемые поля записей

Поля записи могут быть объявлены изменяемыми. Для этого достаточно указать при декларации типа поля ключевое слово `mutable`

Синтаксис

```
type name = { ...; mutable name : t ; ...}
```

Пример определения точки плоскости.

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc : float; mutable yc : float; }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc = 1.; yc = 0.}
```

Для изменения значение поля объявленного `mutable` используйте следующий синтаксис.

Синтаксис

```
expr1.nom <- expr2
```

Выражение `expr1` должно быть типа запись содержащее поле `nom`. Операция модификации возвращает значение типа `unit`.

```
# p.xc <− 3.0 ;;
− : unit = ()
# p ;;
− : point = {xc = 3.; yc = 0.}
```

Напишем функцию перемещения точки, которая изменяет её координаты. Здесь мы используем локальную декларацию с фильтражом, для упорядочения побочных эффектов.

```
# let moveto p dx dy =
    let () = p.xc <− p.xc +. dx
    in p.yc <− p.yc +. dy ;;
val moveto : point −> float −> float −> unit = <fun>
# moveto p 1.1 2.2 ;;
− : unit = ()
# p ;;
− : point = {xc = 4.1; yc = 2.2}
```

Мы можем определить изменяемые или не изменяемые поля, только поля, помеченные ключевым словом `mutable`, будут изменяемые.

```
# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1 : int; mutable c2 : int; }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1 = 0; c2 = 0}
# r.c1 <− 1 ;;
Error: The record field label c1 is not mutable
# r.c2 <− 1 ;;
− : unit = ()
# r ;;
− : t = {c1 = 0; c2 = 1}
```

Далее, мы приводим пример использование записей с изменяемыми полями и массивов для того чтобы реализовать структуру стека (см **??**).

3.3.4 Указатели

Objective CAML предлагает полиморфный тип `ref`, рассматриваемый как указатель на любое значение. В Objective CAML указатель точнее будет назвать указатель на значение. Указываемое значение может быть изменено. Тип `ref` определён как запись с изменяемым полем.

```
type 'a ref = {mutable contents:'a}
```

Создать ссылку на значение можно функцией `ref`. Указываемое значение может быть получено использованием префикса !. Для модификации значения используем инфиксную функцию (:=).

```
# let x = ref 3 ;;
val x : int ref = {contents = 3}
# x ;;
- : int ref = {contents = 3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

3.3.5 Полиморфизм и изменяемые значения

Тип `ref` параметризованный (2.4.6), что позволяет создавать указатели на любой тип. Однако, существует несколько ограничений. Представим, что нет никаких ограничений и мы можем объявить

```
let x = ref [] ;;
```

В этом случае тип переменной `x` будет `'a list ref` и можем изменить значение способом не соответствующим статической типизации Objective CAML.

```
x := 1 :: !x ;; x := true :: !x ;;
```

При этом получаем одну и ту же переменную типа `int list` в один момент и `bool list` в другой.

Во избежании подобной ситуации, система типов Objective CAML использует новую категорию типа переменной — слабо типизированные переменные. Синтаксически, они отличаются предшествующим символом подчёркивания.

```
# let x = ref [] ;;
val x : '_a list ref = {contents = []}
```

Переменная типа `'_a` не является параметром типа, то есть её тип не известен до момента её реализации. Лишь в момент первого использования `x`, после объявления, тип будет окончательно зафиксирован.

```
# x := 0::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents = [0]}
```

Таким образом тип переменной `x int list ref`.

Тип, содержащий неизвестную, является мономорфным, не смотря на то что его тип ещё не был указан и невозможно реализовать то что неизвестно полиморфным типом.

```
# let x = ref [] ;;
val x : '_a list ref = {contents = []}
# x := (function y -> ())::!x ;;
- : unit = ()
# x ;;
- : ('_a -> unit) list ref = {contents = [<fun>]}
```

В этом примере, не смотря на то что мы реализуем неизвестную типом a priori полиморфным (`'a->unit`), тип остался мономорфным с новой неизвестной типа.

Это ограничение полиморфизма распространяется не только на указатели, но и на любое значение содержащее изменяемую часть: массивы, записи с полями объявленные `mutable`, и так далее. Все параметры типа, даже не имеющие никакого отношения к модифицируемым атрибутам, есть переменные слабого (`weak`) типа.

```
# type ('a,'b) t = { ch1 :'a list ; mutable ch2 : 'b list } ;;
type ('a, 'b) t = { ch1 : 'a list; mutable ch2 : 'b list; }
# let x = { ch1 = [] ; ch2 = [] } ;;
val x : ('a, '_b) t = {ch1 = []; ch2 = []}
```

Предупреждение

Эта модификация типа повлияет на чисто функциональные программы. Если к полиморфной переменной применена полиморфная функция, то в результате мы получим переменную слабого типа, так как нельзя не учитывать что функция может создать физически изменяемое значение. Иными словами, результат будет всегда мономорфный.

```
# (function x -> x) [] ;;
- : 'a list = []
```

Тот же результат мы получим для частичных функций.

```
# let f a b = a ;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1 ;;
val g : '_a -> int = <fun>
```

Для получения полиморфного типа необходимо вычесть второй аргумент `f` и применить её:

```
# let h x = f 1 x ;;
val h : 'a -> int = <fun>
```

Действительно, выражения определяющее `x` есть функциональное выражение `function x -> f 1 x`. Его вычисление даст замыкание, которое не рискует произвести побочный эффект, так как тело функции не вычислено.

В общем случае мы делаем различие между т.н. «не расширяемыми» выражениями, которые не вызывают побочных эффектов при вычислении и между «расширяемыми» выражениями. Система типов Objective CAML классифицирует выражения в соответствии с синтаксисом:

- выражения не расширяемые состоят в основном из переменных, конструкторов не изменяемых значений и абстракций
- расширяемые выражения включают в себя преимущественно создание и применение изменяемых значений. Сюда также необходимо включить такие структуры управления как условные операторы и сопоставление с образцом.

3.4 Ввод/вывод

Функции в/в вычисляют значение (чаше всего типа `unit`), однако в ходе вычисления, они изменяют состояние устройств в/в: изменение буфера клавиатуры, перерисовка экрана, запись в файл и так далее. Для каналов ввода и вывода определены два типа: `in_channel` и `out_channel`. При обнаружении конца файла возбуждается исключение `End_of_file`. Следующие константы соответствуют стандартным каналам ввода, вывода и ошибок *a la* Unix: `stdin`, `stdout` и `stderr`.

3.4.1 Каналы

Функции ввода/вывода стандартной библиотеки Objective CAML, манипулируют каналами типа `in_channel` и `out_channel` соответственно. Для создания подобных каналов используется функция:

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

`open_in` открывает файл [2], если он не существует, возбуждается исключение `Sys_error`.

`open_out` создаёт указанный файл если такой не существует, если же он существует то его содержимое уничтожается.

```
# let ic = open_in "koala";;
val ic : in_channel = <abstr>
# let oc = open_out "koala";;
val oc : out_channel = <abstr>
```

Функции закрытия каналов:

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

3.4.2 Чтение/запись

Стандартные функции чтения/записи:

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
```

- `input_line ic`: читает из входного канала `ic` символы до обнаружения возврата каретки или конца файла и возвращает в форме строки (не включая возврат каретки).

[2] с правом на чтение при необходимости

- `input ic s p l`: считывает `l` символов из канала `ic` и помещает их в строку `c` со смещением `p`. Функция возвращает число прочитанных символов.
- `output oc s p l`: записывает в выходной канал `oc` часть строки `s` начиная с символа p длинной `l`.

Следующие функции чтения (записи) из (в) стандартного входа:

```
# read_line ;;
- : unit -> string = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

Другие значения простых типов данных могут быть прочтены/записаны, значения таких типов могут быть переведены в тип список символов.

Локальное объявление и порядок выполнения

Попробуем вывести на экран выражение формы `let x=e1 in e2`. Зная, что в общем случае локальная переменная `x` может быть использована в `e2`, выражение `e1` вычислено первым и только затем `e2`. Если эти оба выражения есть императивные функции с побочным эффектом результат которых (), мы выполнили их в правильном порядке. В частности, так как нам известен тип возвращаемого результата `e1`: константа () типа `unit`, мы получим последовательный вывод на экран используя сопоставление с образцом ().

```
# let () = print_string "and one," in
 let () = print_string " and two," in
 let () = print_string " and three" in
    print_string " zero";;
and one, and two, and three zero- : unit = ()
```

3.4.3 Пример Больше/Меньше

В следующем примере мы приводим игру Больше/Меньше, состоящую в поиске числа, после каждого ответа программа выводит на экран сообщение в соответствии с тем что предложенное число больше или меньше загаданного.

```
# let rec hilo n =
    let () = print_string "type a number: " in
    let i = read_int ()
    in
      if i = n then
        let () = print_string "BRAVO" in
        let () = print_newline ()
        in print_newline ()
      else
        let () =
          if i < n then
            let () = print_string "Higher"
            in print_newline ()
          else
            let () = print_string "Lower"
            in print_newline ()
        in hilo n ;;
val hilo : int -> unit = <fun>
```

Результат запуска

```
# hilo 64;;
type a number: 88
Lower
type a number: 44
Higher
type a number: 64
BRAVO

- : unit = ()
```

3.5 Структуры управления

Операции ввода/вывода и изменяемые значения имеют побочный эффект. Их использование упрощено императивным стилем программирования с новыми структурами контроля. В этом параграфе мы поговорим о последовательных и итеративных структурах.

Нам уже приходилось встречать (см. 2.3.2) условную структуру контроля `if then` смысл которой такой же как и в императивных языках программирования.

Пример

```
# let n = ref 1 ;;
val n : int ref = {contents = 1}
# if !n > 0 then n := !n - 1 ;;
- : unit = ()
```

3.5.1 Последовательность

Самая типичная императивная структура — последовательность, которая позволяет вычислять выражения разделённые точкой с запятой в порядке слева направо.

Синтаксис

```
expr1; ... ;exprn
```

Последовательность — это выражение, результат которого есть результат последнего выражения (здесь `exprn`). Все выражения вычислены и побочный эффект каждой учтён.

```
# print_string "2 = "; 1+1 ;;
2 = - : int = 2
```

С побочным эффектом, мы получаем обычные конструкции императивного программирования.

```
# let x = ref 1 ;;
val x : int ref = {contents = 1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

В связи с тем что значения предшествующие точке запятой не сохранены, Objective CAML выводит предупреждение в случае если их тип отличен от типа `unit`.

```
# print_int 1; 2 ; 3 ;;
Warning 10: this expression should have type unit.
1- : int = 3
```

Чтобы избежать этого сообщения, можно воспользоваться функцией `ignore`.

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

Другое сообщение будет выведено в случае если забыт аргумент функции

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents = 10}
# let u = 1 in g a ; g a u ;;
Warning 5: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
Warning 5: this function application is partial,
maybe some arguments are missing.
- : unit = ()
```

Чаще всего мы используем скобки для определения зоны видимости. Синтаксически расстановка скобок может принимать 2 формы:

Синтаксис

```
(expr)
```

Синтаксис

```
begin expr end
```

Теперь мы можем написать Больше/Меньше простым стилем (стр. **??**).

```
# let rec hilo n =
    print_string "type a number: ";
    let i = read_int () in
    if i = n then print_string "BRAVO\n\n"
     else
       begin
         if i < n then print_string "Higher\n" else print_string "Lower\
             n" ;
         hilo n
       end ;;
val hilo : int -> unit = <fun>
```

3.5.2 Циклы

Структуры итеративного контроля тоже не принадлежат «миру» функционального программирования. Условие повторения цикла или выхода из него имеет смысл в случае когда есть физическое изменение памяти. Существует две структуры итеративного контроля: цикл `for` для ограниченного количества итераций и `while` для неограниченного. Структуры цикла возвращают константу () типа `unit`.

Цикл `for` может быть возрастающим (`to`) или убывающим (`downto`) с шагом в одну единицу.

```
for name = expr1 to expr2 do expr3 done
for name = expr1 downto expr2 do expr3 done
```

Тип выражений `expr1` и `expr2` — int. Если тип `expr3` не `unit`, компилятор выдаст предупреждение.

```
# for i=1 to 10 do print_int i; print_string "  " done; print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i; print_string "  " done;
    print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

Неограниченный цикл `while` имеет следующий синтаксис

```
while expr1 do expr2 done
```

Тип выражения `expr1` должен быть `bool`. И, как в случае `for`, если тип `expr2` не `unit`, компилятор выдаст предупреждение.

```
# let r = ref 1
 in while !r < 11 do
      print_int !r ;
      print_string "  " ;
      r := !r+1
    done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

Необходимо помнить, что циклы — это такие же выражения как и любые другие, они вычисляют значение () типа `unit`.

```
# let f () = print_string "-- end\n" ;;
val f : unit -> unit = <fun>
```

```
# f (for i=1 to 10 do print_int i; print_string "  " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()
```

Обратите внимание на то, что строка « end
n» выводится после цифр 1...10: подтверждение того что аргументы (в данном случае цикл) вычисляются перед передачей функции.

В императивном стиле тело цикла (выражение `expr`) не возвращает результата, а производит побочный эффект. В Objective CAML, если тип тела цикла не `unit`, то компилятор выдаёт сообщение :

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
# for i=0 to 5 do List.tl s done ;;
Warning 10: this expression should have type unit.
- : unit = ()
```

3.5.3 Пример: реализация стека

В нашем примере структура данных `'a stack` будет реализована в виде записи с 2 полями: структура данных `stack` будет запись с двумя полями: массив элементов и индекс первой свободной позиции в массиве.

```
type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array } ;;
```

В поле `size` будет хранится максимальный размер стека.

Операции над стеком будут `init_stack` для инициализации, `push` для добавления и `pop` для удаления элемента.

```
# let init_stack n = {ind=0; size=n; elts =[||]} ;;
val init_stack : int -> 'a stack = <fun>
```

Эта функция не может создавать не пустой массив, так для создания массива необходимо передать элемент. Поэтому поле `elts` получает пустой массив.

Два исключения добавлены на случай попытки удалить элемент из пустого стека и добавить элемент в полный. Они используются в функциях `pop` и `push`.

```
# exception Stack_empty ;;
exception Stack_empty
# exception Stack_full ;;
exception Stack_full
```

```
# let pop p =
    if p.ind = 0 then raise Stack_empty
    else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
val pop : 'a stack -> 'a = <fun>
# let push e p =
    if p.elts = [||] then
      (p.elts <- Array.create p.size e;
       p.ind <- 1)
    else if p.ind >= p.size then raise Stack_full
    else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

Небольшой пример использования:

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind = 0; size = 4; elts = [||]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Exception: Stack_full.
# p ;;
- : int stack = {ind = 4; size = 4; elts = [|1; 2; 3; 4|]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3
```

Чтобы избежать исключения `Stack_full` при переполнении стека, мы можем каждый раз увеличивать его размер. Для этого нужно чтобы поле `size` было модифицируемым.

```
# type 'a stack =
    {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
type 'a stack = {
  mutable ind : int;
  mutable size : int;
  mutable elts : 'a array;
}
# let init_stack n = {ind=0; size=max n 1; elts = [||]} ;;
val init_stack : int -> 'a stack = <fun>
# let n_push e p =
    if p.elts = [||]
    then
```

```
    begin
      p.elts <− Array.create p.size e;
      p.ind <− 1
    end
  else if p.ind >= p.size then
    begin
      let nt = 2 ∗ p.size in
      let nv = Array.create nt e in
      for j=0 to p.size−1 do nv.(j) <− p.elts.(j) done ;
      p.elts <− nv;
      p.size <− nt;
      p.ind <− p.ind + 1
    end
  else
    begin
      p.elts.(p.ind) <− e ;
      p.ind <− p.ind + 1
    end ;;
val n_push : 'a −> 'a stack −> unit = <fun>
```

Однако необходимо быть осторожным со структурами которые могут беспредельно увеличиваться. Вот небольшой пример стека, увеличивающегося по необходимости.

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind = 0; size = 4; elts = [|||]}
# for i = 1 to 5 do n_push i p done ;;
− : unit = ()
# p ;;
− : int stack = {ind = 5; size = 8; elts = [|1; 2; 3; 4; 5; 5; 5; 5|]}
# p.stack ;;
Error: Unbound record field label stack
```

В функции `pop` желательно добавить возможность уменьшения размера стека, это позволит сэкономить память.

3.5.4 Пример: расчёт матриц

В этом примере мы определим тип «матрица» — двумерный массив чисел с плавающей запятой и несколько функций. Мономорфный тип `mat` это запись, состоящая из размеров и элементов матрицы. Функции

`create_mat`, `access_mat` и `mod_mat` служат для создания, доступа и изменения элементов матрицы.

```
# type mat = { n:int; m:int; t: float array array };;
type mat = { n : int; m : int; t : float array array; }
# let create_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0
    } ;;
val create_mat : int -> int -> mat = <fun>
# let access_mat m i j = m.t.(i).(j) ;;
val access_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
# let a = create_mat 3 3 ;;
val a : mat =
  {n = 3; m = 3; t = [|[|0.; 0.; 0.|]; [|0.; 0.; 0.|]; [|0.; 0.; 0.|]|]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0 ;;
- : unit = ()
# a ;;
- : mat =
{n = 3; m = 3; t = [|[|0.; 0.; 0.|]; [|0.; 2.; 1.|]; [|0.; 1.; 0.|]|]}
```

Сумма двух матриц `a` и `b` есть матрица `c`, такая что $c_{ij} = a_{ij} + b_{ij}$

```
# let add_mat p q =
   if p.n = q.n && p.m = q.m then
     let r = create_mat p.n p.m in
     for i = 0 to p.n-1 do
       for j = 0 to p.m-1 do
         mod_mat r i j (p.t.(i).(j) +. q.t.(i).(j))
       done
     done ;
     r
   else failwith "add_mat : dimensions incompatible";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat a a ;;
- : mat =
{n = 3; m = 3; t = [|[|0.; 0.; 0.|]; [|0.; 4.; 2.|]; [|0.; 2.; 0.|]|]}
```

Произведение двух матриц `a` и `b` есть матрица `c`, такая что $c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{ki}$

```
# let mul_mat p q =
   if p.m = q.n then
```

```
    let r = create_mat p.n q.m in
    for i = 0 to p.n−1 do
      for j = 0 to q.m−1 do
        let c = ref 0.0 in
        for k = 0 to p.m−1 do
          c := !c +. (p.t.(i).(k) ∗. q.t.(k).(j))
        done;
        mod_mat r i j !c
      done
    done;
    r
  else failwith "mul_mat : dimensions incompatible" ;;
val mul_mat : mat −> mat −> mat = <fun>
# mul_mat a a;;
− : mat =
{n = 3; m = 3; t = [|[|0.; 0.; 0.|]; [|0.; 5.; 2.|]; [|0.; 2.; 1.|]|]}
```

3.6 Порядок вычисления аргументов

В функциональном языке программирования порядок вычисления аргументов не имеет значения. Из–за того что нет ни изменения памяти, ни приостановки вычисления, расчёт одного аргумента не влияет на вычисление другого. Objective CAML поддерживает физически изменяемые значения и исключения, поэтому пренебречь порядком вычисления аргументов нельзя. Следующий пример специфичен для Objective CAML 3.12.1 OC Linux на платформе Intel:

```
# let new_print_string s = print_string s; String.length s ;;
val new_print_string : string −> int = <fun>
# (+) (new_print_string "Hello ") (new_print_string "World!") ;;
World!Hello − : int = 12
```

По выводу на экран мы видим что вторая строка печатается после первой.

Таков же результат для исключений:

```
# try (failwith "function") (failwith "argument") with Failure s −> s;;
Warning 20: this argument will not be used by the function.
− : string = "argument"
```

Если необходимо указать порядок вычисления аргументов, необходимо использовать локальные декларации, форсируя таким образом порядок перед вызовом функции. Предыдущий пример может быть переписан следующим способом:

```
# let e1 = (new_print_string "Hello ")
 in let e2 = (new_print_string "World!")
 in (+) e1 e2 ;;
Hello World!- : int = 12
```

В Objective CAML порядок вычисления не указан, на сегодняшний день все реализации caml делают это слева направо. Однако рассчитывать на это может быть рискованно, в случае если в будущем язык будет реализован иначе.

Это вечный сюжет дебатов при концепции языка. Нужно ли специально не указывать некоторые особенности языка и предложить программистам не пользоваться ими, иначе они рискуют получить разные результаты для разных компиляторов. Или же необходимо их указать и, следовательно, разрешить программистам ими пользоваться, что усложнит компилятор и сделает невозможным некоторые оптимизации?

3.7 Калькулятор с памятью

Вернёмся к нашему примеру с калькулятором описанным в предыдущей главе и добавим ему пользовательский интерфейс. Теперь мы будем вводить операции напрямую и получать результат сразу без вызова функции переходов (из одного состояния в другое) при каждом нажатии на кнопку.

Добавим 4 новые кнопки: `C` которая очищает экран, `M` для сохранения результата в памяти, `m` для его вызова из памяти и `OFF` для выключения калькулятора. Что соответствует следующему типу:

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int
            | Store | Recall | Clear | Off ;;
type key =
    Plus
  | Minus
  | Times
  | Div
  | Equals
  | Digit of int
  | Store
```

```
  | Recall
  | Clear
  | Off
```

Теперь определим функцию, переводящую введённые символы в значение типа `key`. Исключение `Invalid_key` отлавливает все символы, которые не соответствуют кнопкам калькулятора. Функция `code` модуля `Char` переводит символ в код *ASCII*.

```
# exception Invalid_key ;;
exception Invalid_key
# let translation c = match c with
      '+' -> Plus
    | '-' -> Minus
    | '*' -> Times
    | '/' -> Div
    | '=' -> Equals
    | 'C' | 'c' -> Clear
    | 'M' -> Store
    | 'm' -> Recall
    | 'o' | 'O' -> Off
    | '0'..'9' as c -> Digit ((Char.code c) - (Char.code '0'))
    | _ -> raise Invalid_key ;;
val translation : char -> key = <fun>
```

В императивном стиле, функция перехода (`transition`) не приведёт к новому состоянию, а физически изменит текущее состояние калькулятора. Таким образом необходимо изменить тип `state` так, чтобы его поля были модифицируемые. Наконец, определим исключение `Key_off` для обработки нажатия на кнопку `OFF`.

```
# type state = {
  mutable lcd : int; (* last computation done *)
  mutable lka : bool; (* last key activated *)
  mutable loa : key; (* last operator activated *)
  mutable vpr : int; (* value printed *)
  mutable mem : int (* memory of calculator *)
 };;
type state = {
  mutable lcd : int;
  mutable lka : bool;
  mutable loa : key;
  mutable vpr : int;
```

```
  mutable mem : int;
}
# exception Key_off ;;
exception Key_off
# let transition s key = match key with
      Clear -> s.vpr <- 0
    | Digit n -> s.vpr <- ( if s.lka then s.vpr*10+n else n );
                 s.lka <- true
    | Store -> s.lka <- false ;
                 s.mem <- s.vpr
    | Recall -> s.lka <- false ;
                 s.vpr <- s.mem
    | Off -> raise Key_off
    | _ -> let lcd = match s.loa with
                        Plus -> s.lcd + s.vpr
                      | Minus -> s.lcd - s.vpr
                      | Times -> s.lcd * s.vpr
                      | Div -> s.lcd / s.vpr
                      | Equals -> s.vpr
                      | _ -> failwith "transition: impossible match
                          "
                   in
           s.lcd <- lcd ;
           s.lka <- false ;
           s.loa <- key ;
           s.vpr <- s.lcd;;
val transition : state -> key -> unit = <fun>
```

Определим функцию запуска калькулятора `go`, которая возвращает (), так как нас интересует только её эффект на окружение (ввод/вывод, изменение состояния). Её аргумент есть константа (), так как наш калькулятор автономен (он сам определяет своё начальное состояние) и интерактивен (данные необходимые для расчёта вводятся с клавиатуры по мере необходимости). Переходы реализуются в бесконечном цикле (`while true do`) из которого мы можем выйти при помощи исключения `Key_off`.

```
# let go () =
  let state = { lcd=0; lka=false; loa=Equals; vpr=0; mem=0 }
  in try
       while true do
         try
```

```
            let input = translation (input_char stdin)
            in transition state input ;
               print_newline () ;
               print_string "result: " ;
               print_int state.vpr ;
               print_newline ()
          with
            Invalid_key -> () (* no effect *)
        done
      with
        Key_off -> () ;;
val go : unit -> unit = <fun>
```

Заметим, что начальное состояние должно быть либо передано в параметре, либо объявлено локально внутри функции `go`, для того чтобы оно каждый раз инициализировалось при запуске этой функции. Если бы мы использовали значение `initial_state` в функциональном стиле, калькулятор начинал бы работать со старым состоянием, которое он имел перед выключением. Таким образом было бы не просто использовать два калькулятора в одной программе.

3.8 Exercises

3.9 Резюме

В этой главе вы видели применение стилей императивного программирования (физически изменяемые значения, ввод–вывод, структуры итеративного контроля) в функциональном языке. Только `mutable` значения, такие как строки, векторы и записи с изменяемыми полями могут быть физически изменены. Другие значения не могут меняться после их создания. Таким образом мы имеем значения `read-only` для функциональной части и значения `read-write` для императивной.

3.10 Калькулятор с памятью

Глава 4

Функциональный и императивный стиль

4.1 Введение

Языки функционального и императивного программирования различаются контролем выполнения программы и управлением памятью.

- функциональная программа вычисляет выражение, в следствии чего мы получаем некоторый результат. Порядок в котором выполнены операции расчёта и физическое представление данных не влияют на результат, он одинаков во всех случаях. При таком порядке вычислений, сбор памяти в Objective CAML неявно осуществляется вызовом автоматического сборщика мусора или Garbage Collector (GC) (см. главу **??**).

- императивная программа это список инструкций изменяющих состояние памяти. Каждый этап выполнения строго определён структурами контроля, указывающими на следующую инструкцию. Такие программы чаще всего манипулируют указателями на значение, чем самими значениями. Отсюда необходимость явного выделения и освобождения памяти, что может порой приводить к ошибкам доступа к памяти, однако ничто не запрещает использование GC.

Императивные языки предоставляют больший контроль над выполнением программы и памятью. Находясь ближе к реальной машине, такой код может быть эффективнее, но при этом код теряет в устойчивости выполнения. Функциональное программирование предоставляет

более высокий уровень абстракции и таким образом лучший уровень устойчивости выполнения: типизация (динамическая или статическая) помогает избежать некорректных значений, автоматическая сборка мусора, хотя и замедляет скорость выполнения, обеспечивает правильное манипулирование областями памяти.

Исторически обе эти парадигмы программирования существовали в разных сферах: символьные программы для первого случая и числовые для второго. Однако, с тех времён кое–что изменилось, в частности появилась техника компиляции функциональных языков и выросла эффективность GC. С другой стороны, устойчивость выполнения стала важным критерием, иногда даже важнейшим критерием качества программного обеспечения. В подтверждение этому «коммерческий аргумент» языка Java: эффективность не должна преобладать над правильностью, оставаясь при этом благоразумно хорошей. Эта идея приобретает с каждым днем новых сторонников в мире производителей программного обеспечения.

Objective CAML придерживается этой позиции: он объединяет обе парадигмы, расширяя таким образом область своего применения и облегчая написание алгоритмов в том или ином стиле. Он сохраняет, однако, хорошие свойства правильности выполнения благодаря статической типизации, GC и механизму исключений. Исключения — это первая структура контроля выполнения позволяющая приостановить/продолжить расчёт при возникновении определённых условий. Эта особенность находится на границе двух стилей программирования, хоть она и не изменяет результат, но может изменить порядок вычислений. Введение физически изменяемых значений может повлиять на чисто функциональную часть языка. Порядок вычисления аргументов функции становится определимым если это вычисление производит побочный эффект. По этим причинам подобные языки называются “не чистые функциональные языки”. Мы теряем часть абстракции, так как программист должен учитывать модель памяти и выполнение программы. Это не всегда плохо, в частности для читаемости кода. Однако, императивные особенности изменяют систему типов языка: некоторые функциональные программы, с теоретически правильными типами, не являются правильными на практике из-за введения ссылок (reference). Хотя такие программы могут быть легко переписаны.

4.2 План главы

В этой главе мы сравним функциональную и императивную модель Objective CAML по критерию контроля выполнения программы и представления значений в памяти. Смесь обоих стилей позволяет конструировать новые структуры данных. Это будет рассмотрено в первом разделе. Во втором разделе мы обсудим выбор между композицией функций (composition of functions) или последовательности (sequencing) с одной стороны и разделение (sharing) или копирование значений с другой. Третий раздел выявляет интерес к смешиванию двух стилей для создания функциональных изменяемых данных (mutable functional data), что позволит создавать не полностью вычисленные (evaluated) данные. В четвёртом разделе рассмотрены streams, потенциально бесконечные потоки данных и их интеграция, посредством сопоставления с образцом.

4.3 Сравнение между функциональным и императивным стилями

Воспользуемся строками (`string`) и списками (`'a list`) для иллюстрации разницы между двумя стилями.

4.3.1 С функциональной стороны

`map` это одна из классических функций в среде функциональных языков. В чистом функциональном стиле она пишется так:

```
# let rec map f l = match l with
      [] -> []
  | h::q -> (f h) :: (map f q) ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Конечный список состоит из результатов применения функции `f` к элементам списка переданного в аргументе. Он рекурсивно создаётся указывая начальный элемент (заголовок) (`f t`) и последующую часть (хвост) (`map f q`). В частности, программа не указывает какой из них будет вычислен первым.

При этом, для написания такой функции программисту не нужно знать физическое представление данных. Проблемы выделения памяти и разделения данных решаются Objective CAML без внешнего вмешательства программиста. Следующий пример иллюстрирует это:

```
# let example = [ "one" ; "two" ; "three" ] ;;
val example : string list = ["one"; "two"; "three"]
# let result = map (function x -> x) example ;;
val result : string list = ["one"; "two"; "three"]
```

Оба списка `example` и `result` содержат одинаковые значения:

```
# example = result ;;
- : bool = true
```

Оба значения имеют одинаковую структуру, хоть они и представлены в памяти по разному. Убедимся в этом при помощи проверки на физическое равенство:

```
# example == result ;;
- : bool = false
# (List.tl example) == (List.tl result) ;;
- : bool = false
```

4.3.2 Императивная сторона

Вернёмся к предыдущему примеру и изменим строку списка `result`.

```
# (List.hd result).[1] <- 's' ;;
- : unit = ()
# result ;;
- : string list = ["ose"; "two"; "three"]
# example ;;
- : string list = ["ose"; "two"; "three"]
```

Определено, изменив список `result`, мы изменили список `example`. То есть знание физической структуры необходимо как только мы пользуемся императивными особенностями.

Рассмотрим теперь как порядок вычисления аргументов функции может стать ловушкой в императивном программировании. Определим структуру изменяемого списка, а так же функции создания, добавления и доступа:

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c : 'a list; }
# let icreate () = { c = [] }
 let iempty l = (l.c = [])
 let icons x y = y.c <- x::y.c ; y
```

```
 let ihd x = List.hd x.c
 let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
    if iempty l then icreate()
    else icons (f (ihd l)) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
```

Несмотря на то что мы переняли общую структуру функции `map` предыдущего параграфа, с `imap` мы получим другой результат:

```
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x -> x) example ;;
Exception: Failure "hd".
```

В чем тут дело? Вычисление (`itl l`) произошло раньше чем (`ihd l`) и в последней итерации `imap`, список `l` пустой в момент обращения к его заголовку. Список `example` теперь пуст, хоть мы и не получили никакого результата:

```
# example ;;
- : string ilist = {c = []}
```

Проблема функции `imap` в недостаточном контроле смеси стилей программирования: мы предоставили системе выбор порядка вычисления. Переформулируем функцию `imap` явно указав порядок при помощи конструкции `let .. in ..`

```
# let rec imap f l =
    if iempty l then icreate()
    else let h = ihd l in icons (f h) (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x -> x) example ;;
- : string ilist = {c = ["one"; "two"; "three"]}
```

Однако, начальный список опять утерян:

```
# example ;;
- : string ilist = {c = []}
```

Использование оператора последовательности (`sequencing operator`) и цикла есть другой способ явного указания порядка.

```
# let imap f l =
    let l_res = icreate ()
    in while not (iempty l) do
         ignore (icons (f (ihd l)) l_res) ;
         ignore (itl l)
       done ;
       { l_res with c = List.rev l_res.c } ;;
Warning 23: this record is defined by a 'with' expression,
but no fields are borrowed from the original.
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c = ["one"; "two"; "three"]}
# imap (function x -> x) example ;;
- : string ilist = {c = ["one"; "two"; "three"]}
```

Присутствие `ignore` — это факт того что нас интересует побочный эффект функций над её аргументами, а не их (функций) результат. Так же было необходимо выстроить в правильном порядке элементы результата (функцией `List.rev`).

4.3.3 Рекурсия или итерация

Часто ошибочно ассоциируют рекурсию с функциональным стилем и императивный с итерацией. Чисто функциональная программа не может быть итеративной, так как значение условия цикла не меняется никогда. Тогда как императивная программа может быть рекурсивной: функция `imap` тому пример.

Значения аргументов функции хранятся во время её выполнения. Если она (функция) вызовет другую функцию, то эта последняя сохранит и свои аргументы. Эти значения хранятся в стеке выполнения. При возврате из функции эти значения извлечены из стека. Зная что пространство памяти ограничено, мы можем дойти до её предела, используя функцию с очень большой глубиной рекурсии. В подобных случаях Objective CAML возбуждает исключение `Stack_overflow`.

```
# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;;
```

```
val succ : int -> int = <fun>
# succ 100000 ;;
- : int = 100001
```

В итеративной версии место занимаемое функцией `succ_iter` в стеке не зависит от величины аргумента.

```
# let succ_iter n =
    let i = ref 0 in
      for j=0 to n do incr i done ;
      !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;
- : int = 100001
```

У следующей версии функции *a priori* аналогичная глубина рекурсии, однако она успешно выполняется с тем же аргументом.

```
# let succ_tr n =
    let rec succ_aux n accu =
      if n = 0 then accu else succ_aux (n-1) (accu+1)
    in
      succ_aux 1 n ;;
val succ_tr : int -> int = <fun>
# succ_tr 100000 ;;
- : int = 100001
```

Данная функция имеет специальную форму рекурсивного вызова, так называемую хвостовую рекурсию при которой результат вызова функции будет результатом функции без дополнительных вычислений. Таким образом отпадает необходимость хранить аргументы функции во время вычисления рекурсивного вызова. Если Objective CAML распознал конечную рекурсивную функцию, то перед её вызовом аргументы извлекаются из стека.

Распознание конечной рекурсии существует во многих языках, однако это свойство просто необходимо функциональным языкам, где она естественно много используется.

4.4 Какой стиль выбрать?

Естественно, речь не идёт о «священных» либо эстетических для каждого понятиях, *a priori* не существует стиля красивее или лучше чем

другой. Однако, в зависимости от решаемой проблемы, один стиль может быть более подходящим или более адаптированным чем другой.

Первое правило — простота. Желаемый алгоритм (будь то в книге или лишь в голове программиста) уже определён в каком-то стиле, вполне естественно его и использовать при реализации.

Второй критерий — эффективность программы. Можно сказать что императивная программа (хорошо написанная) более эффективна чем её функциональный аналог, но в очень многих случаях разница между ними не достаточно существенна для оправдания сложности кода императивного стиля, там где функциональный был бы естественен. Функция `map` есть хороший пример натурально выраженной в функциональном стиле проблемы и мы ни в чем не выиграем написав её в императивном стиле.

4.4.1 Последовательность или композиция функций

Мы уже видели, что как только в программе появляются побочные эффекты, необходимо явно указывать порядок выполнения элементов программы. Это может быть сделано двумя способами:

функциональным опираясь на то что Objective CAML строгий язык, то есть аргумент вычислен до того как он будет передан функции. В выражение `(f (g x))` сначала вычисляется `(g x)` и затем передача этого результат функции `f`. В более сложных выражениях, промежуточный результат может быть именован конструкцией `let in`, но принцип остаётся тем же: `let aux=(g x) in (f aux)`.

императивный используя последовательность или другую структуру контроля (цикл). В этом случае, результатом будет побочный эффект над памятью, а не значение возвращённое функцией : `aux:=(g x) ; (f!aux)`.

Давайте рассмотрим данную проблему выбора стиля на следующем примере. Рекурсивный алгоритм быстрой сортировки вектора описывается следующим образом:

1. выбрать опорную точку: выбрать индекс элемента в векторе

2. переставить элементы вокруг этой точки: переставить элементы вектора так чтобы значения меньше значения в опорной точке были слева от неё, а большие значения справа

3. отсортировать (тем же алгоритмом) два полученных вектора: элементы после опорной точки и до неё

Сортировка вектора — означает изменение его состояния, поэтому мы должны использовать императивный стиль, как минимум для манипуляции данными.

Начнём с определения функции переставляющей элементы вектора.

```
# let permute_element vec n p =
    let aux = vec.(n) in vec.(n) <- vec.(p) ; vec.(p) <- aux ;;
val permute_element : 'a array -> int -> int -> unit = <fun>
```

Выбор правильной точки опоры важен для эффективности алгоритма, но мы ограничимся самым простым способом: вернём индекс первого элемента вектора.

```
# let choose_pivot vec start finish = start ;;
val choose_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Напишем теперь желаемый алгоритм переставляющий элементы вектора вокруг выбранной точки.

- установить опорную точку в начало вектора
- `i` индекс второго элемента вектора
- `j` индекс последнего элемента вектора
- если элемент с индексом `j` больше чем значение в опорной точке, поменяем местами их значения и увеличим `i` на единицу, иначе уменьшим `j` на единицу
- до тех пор пока `i` меньше чем `j` повторить предыдущую операцию
- к этому этапу каждый элемент с индексом меньшим чем `i` (или `j`) меньше значения в опорной точке, а остальные элементы больше: если элемент с индекс `i` меньше чем опорное значение мы меняем их местами, иначе меняем с предыдущим элементом.

При реализации алгоритма мы естественно воспользуемся императивными управляющими структурами.

```
# let permute_pivot vec start finish ind_pivot =
    permute_element vec start ind_pivot ;
    let i = ref (start+1) and j = ref finish and pivot = vec.(start) in
```

```
    while !i < !j do
      if vec.(!j) >= pivot then decr j
      else
        begin
          permute_element vec !i !j ;
          incr i
        end
    done ;
    if vec.(!i) > pivot then decr i ;
    permute_element vec start !i ;
    !i
;;
val permute_pivot : 'a array -> int -> int -> int -> int = <fun>
```

Кроме эффекта над вектором, функция возвращает индекс опорной точки.

Нам остаётся лишь собрать воедино различные этапы и написать рекурсивный вызов для подвекторов.

```
# let rec quick vec start finish =
    if start < finish
    then
      let pivot = choose_pivot vec start finish in
      let place_pivot = permute_pivot vec start finish pivot in
      quick (quick vec start (place_pivot-1)) (place_pivot+1) finish
    else vec ;;
val quick : 'a array -> int -> int -> 'a array = <fun>
```

Здесь мы воспользовались двумя стилями. Выбранная опорная точка служит аргументом при перестановке вокруг неё и её порядок в векторе после этой процедуры служит аргументом рекурсивного вызова. Зато, полученный после перестановки вектор мы получаем в результате побочного эффекта, а не как возвращённое значение функции `permute_pivot`. Тогда как функция `quick` возвращает вектор и сортировка подвекторов реализуется композицией рекурсивных вызовов.

Теперь главная функция:

```
# let quicksort vec = quick vec 0 ((Array.length vec)-1) ;;
val quicksort : 'a array -> 'a array = <fun>
```

Это полиморфная функция, так как отношение порядка < полиморфное.

```
# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"the"; "little"; "cat"; "is"; "dead"|] ;;
val t2 : string array = [|"the"; "little"; "cat"; "is"; "dead"|]
# quicksort t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
# t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
```

4.4.2 Общее использование или копии значений

До тех пор пока наши данные не изменяемые, нет необходимости знать используются они совместно или нет.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]
```

Является ли `b` копией `a` или же это один и тот же список не имеет значения, так как по любому это «неосязаемые» значения. Однако, если на место целых чисел, мы поместим изменяемые значения, необходимо будет знать скажется ли изменение одного значения на другое.

Реализация полиморфизма Objective CAML вызывает (causes) копирование непосредственных значений (immediate values) и разделение (sharing) структурных значений. Хоть передача аргументов осуществляется копированием, в случае структурных значений передается лишь указатель. Как в случае с функцией `id`.

```
# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
```

```
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]
```

Здесь мы действительно имеем случай выбора стиля программирования, от которого зависит эффективность представления данных. С одной стороны, использование изменяемых значений позволяет немедленное манипулирование данными (без дополнительного выделения памяти). Однако это вынуждает, в некоторых случаях, делать копии там, где использование неизменяемого значения позволило бы разделение. Проиллюстрируем это двумя способами реализации списков.

```
# type 'a list_immutable = LInil | LIcons of 'a * 'a list_immutable ;;
type 'a list_immutable = LInil | LIcons of 'a * 'a list_immutable
# type 'a list_mutable = LMnil | LMcons of 'a * 'a list_mutable ref ;;
type 'a list_mutable = LMnil | LMcons of 'a * 'a list_mutable ref
```

Фиксированные списки строго эквивалентны спискам Objective CAML, тогда как изменяемые больше в стиле C, где ячейка состоит из значения и ссылки на следующую ячейку.

С фиксированными списками существует единственный способ реализовать конкатенацию и он вынуждает копирование структуры первого списка, тогда как второй список может быть разделен с конечным списком.

```
# let rec concat l1 l2 = match l1 with
    LInil -> l2
  | LIcons (a,l11) -> LIcons(a, (concat l11 l2)) ;;
val concat : 'a list_immutable -> 'a list_immutable -> 'a
  list_immutable =
 <fun>
# let li1 = LIcons(1, LIcons(2, LInil))
 and li2 = LIcons(3, LIcons(4, LInil)) ;;
val li1 : int list_immutable = LIcons (1, LIcons (2, LInil))
val li2 : int list_immutable = LIcons (3, LIcons (4, LInil))
# let li3 = concat li1 li2 ;;
val li3 : int list_immutable =
  LIcons (1, LIcons (2, LIcons (3, LIcons (4, LInil))))
# li1==li3 ;;
- : bool = false
# let tlLI l = match l with
   LInil -> failwith "Liste vide"
 | LIcons(_,x) -> x ;;
```

```
val tlLI : 'a list_immutable -> 'a list_immutable = <fun>
# tlLI(tlLI(li3)) == li2 ;;
- : bool = true
```

В этом примере мы видим что первые ячейки `li1` и `li3` различны, тогда как вторая часть `li3` есть именно `li2`.

С изменяемыми списками можно изменить аргументы (функция `concat_share`) или создать новое значение (функция `concat_copy`)

```
# let rec concat_copy l1 l2 = match l1 with
    LMnil -> l2
  | LMcons (x,l11) -> LMcons(x, ref (concat_copy !l11 l2)) ;;
val concat_copy : 'a list_mutable -> 'a list_mutable -> 'a list_mutable
    =
  <fun>
```

Это решение, `concat_copy`, аналогично предыдущей функции `concat`. Вот второе вариант, с общим использованием.

```
# let concat_share l1 l2 =
  match l1 with
    LMnil -> l2
  | _ -> let rec set_last = function
                LMnil -> failwith "concat_share : impossible case!!
                    "
              | LMcons(_,l) -> if !l=LMnil then l:=l2 else set_last !l
            in
              set_last l1 ;
              l1 ;;
val concat_share : 'a list_mutable -> 'a list_mutable -> 'a
    list_mutable =
  <fun>
```

Конкатенация с общим использованием не нуждается в выделении памяти (мы не используем конструктор `LMcons`), мы лишь ограничиваемся тем что последняя ячейка первого списка теперь указывает на второй список. При этом, конкатенация способна изменить аргументы переданные функции.

```
# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
 and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int list_mutable =
  LMcons (1, {contents = LMcons (2, {contents = LMnil})})
val lm2 : int list_mutable =
```

```
  LMcons (3, {contents = LMcons (4, {contents = LMnil})})
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int list_mutable =
  LMcons (1,
   {contents =
     LMcons (2,
      {contents = LMcons (3, {contents = LMcons (4, {contents = LMnil
          })})})})})
```

Мы получили ожидаемый результат для `lm3`, однако значение `lm1` изменено.

```
# lm1 ;;
- : int list_mutable =
LMcons (1,
 {contents =
   LMcons (2,
    {contents = LMcons (3, {contents = LMcons (4, {contents = LMnil})
        })})})
```

Таким образом это может повлиять на оставшуюся часть программы.

4.4.3 Критерии выбора

В чисто функциональной программе побочных эффектов не существует, это свойство лишает нас операций ввода/вывода, исключений и изменяемых структур данных. Наше определение функционального стиля является менее ограниченным, то есть функция не изменяющая своё глобальное окружение может быть использована в функциональном стиле. Подобная функция может иметь локальные изменяемые значения (и значит следовать императивному стилю), но не должна изменять ни глобальные переменные ни свои аргументы. С внешней стороны, такие функции можно рассматривать как «чёрный ящик», чьё поведение сравнимо с поведением функции в чисто функциональном стиле, с разницей что выполнение первой может быть приостановлен возбуждением исключения. В том же духе, изменяемое значение, которое больше не изменяется после инициализации, может быть использовано в функциональном стиле.

С другой стороны, программа написанная в императивном стиле, унаследует следующие достоинства Objective CAML: правильность статической типизации, автоматическое управление памятью, механизм исключений, параметризованный полиморфизм и вывод типов.

Выбор между стилем функциональным и императивным зависит от программного обеспечения которое вы хотите реализовать. Выбор может быть сделан в соответствии со следующими характеристиками:

выбор структур данных: от использования или нет изменяемых структур данных будет зависеть выбор стиля программирования. Действительно, функциональный стиль по своей природе не совместим с изменением значений. Однако, создание и просмотр значения не зависят от её свойств. Таким образом мы возвращаемся к дискуссии «изменение на месте vs копия» в 4.4.2, к которой мы вернёмся для обсуждения критерия эффективности.

структура данных существует: если в программе необходимо менять изменяемые структуры данных (modify mutable data structures), то императивный стиль является единственно возможным. Однако, если необходимо лишь читать значения, то применение функционального стиля гарантирует целостность данных. Использование рекурсивных структур данных подразумевает рекурсивные функции, которые могут быть определены используя особенности того или иного стиля программирования. Однако в общем случае бывает проще истолковывать создание значения следуя рекуррентному определению. Этот подход более близок к функциональному стилю, чем повторяющаяся обработка этого значения рекурсией.

критерий эффективности: немедленное изменение без сомнения лучше чем создание значения. В случае когда эффективность кода является главенствующим критерием, чаша весов перевешивает в сторону императивного стиля. Однако отметим что совместное использование значений может оказаться нелёгкой задачей и в конце концов более дорогостоящей, чем копирование значений с самого начала. Чистая функциональность имеет определённую цену: частичное применение (partial application) и использование функций переданных в виде аргумента другой функции несёт более серьёзные накладные расходы в процессе выполнения программы, чем явное применение видимой функции. Стоит избегать использования этой функциональной особенности в тех местах, где критерий производительности является решающим.

критерий разработки: более высокий уровень абстракции функционального стиля программирования позволяет более быстрое написание компактного кода, содержащего меньше ошибок чем императивный вариант, который по своей природе является более «мно-

гословным». Таким образом, функциональный стиль является выгодным при разработке значительных программ. Независимость функции к своему контексту окружения позволяет разделить код на более маленькие части и тем самым облегчить его проверку и читаемость. Более высокая модульность функционального стиля плюс возможность передавать функции (а значит и обработку) в аргумент другим функциям повышает вероятность повторного использования программ.

Эти несколько замечаний подтверждают тот факт, что часто смешанное использование обоих стилей является рациональным выбором. Функциональный стиль быстрее в разработке и обеспечивает более простую организацию программы, однако части кода где необходимо повысить скорость выполнения лучше написать в императивном стиле.

4.5 Смешивая стили

Как мы уже заметили, язык программирования имеющий функциональные и императивные особенности даёт свободу выбора наиболее подходящего стиля для реализации того или иного алгоритма. Конечно, мы можем использовать оба аспекта Objective CAML совместно в одной и той же функции. Именно этим мы сейчас и займёмся.

4.5.1 Замыкания и побочные эффекты

Обычно функция с побочным эффектом рассматривается как процедура и она возвращает значение () типа `unit`. Однако, иногда бывает полезно произвести побочный эффект внутри функции и вернуть определённое значение. Мы уже использовали подобный коктейль стилей в функции `permute_pivot` в быстрой сортировке.

В следующем примере реализуем генератор символов, который создаёт новый символ при каждом вызове функции. Мы используем счётчик, значение которого увеличивается с каждым вызовом.

```
# let c = ref 0;;
val c : int ref = {contents = 0}
# let reset_symb = function () -> c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s -> c:=!c+1 ; s^(string_of_int !c) ;;
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
```

```
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

А теперь спрячем ссылку на с от всей программы следующим образом:

```
# let (reset_s , new_s) =
    let c = ref 0
    in let f1 () = c := 0
       and f2 s = c := !c+1 ; s^(string_of_int !c)
    in (f1,f2) ;;
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

Таким образом мы объявили пару функций, разделяющие локальную (для декларации) переменную `c`. Использование обеих функций производит тот же результат что и раньше.

Этот пример иллюстрирует способ представления замыкания. Замыкание можно рассматривать как пару, состоящую из кода (то есть часть `function`) и локальное окружение, содержащее значения свободных переменных замыкания с другой стороны. На диаграмме 4.1 мы можем увидеть представление в памяти замыканий `reset_s` и `new_s`.

Оба эти замыкания разделяют одно и тоже окружение (значение `c`). Когда одно из них меняет ссылку `c`, то оно меняет содержимое памяти разделяемое с другим окружением.

4.5.2 Физическое изменение и исключения

Исключения позволяет отслеживать ситуацию, при которой дальнейшее продолжение программы невозможно. В подобном случае сборщик исключений позволит продолжить вычисление, зная что произошла ошибка. В момент возбуждения исключения может возникнуть проблема побочного эффекта с состоянием модифицируемых данных. Это состояние не может быть гарантированно если произошли физические изменения в ответвлении неудавшегося вычисления.

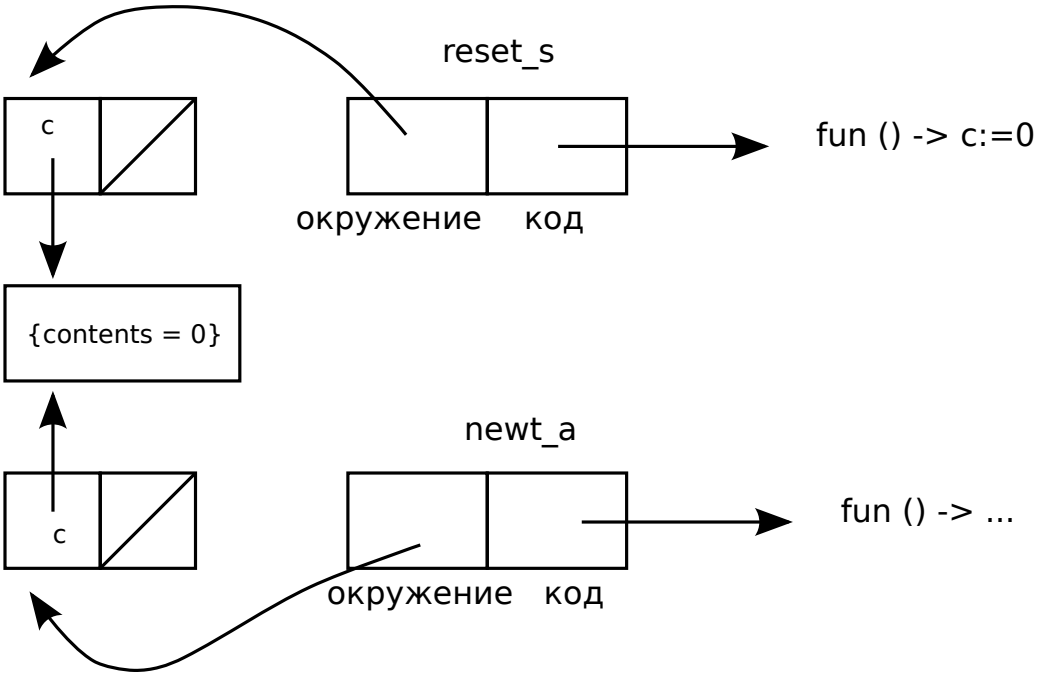

Рис. 4.1: Представление в памяти замыканий

Определим функцию увеличения (++), с аналогичным результатом что и в C:

```
# let (++) x = x:=!x+1; x;;
val ( ++ ) : int ref -> int ref = <fun>
```

Следующий пример, иллюстрирует небольшой расчёт в котором деление на ноль совпадает с побочным эффектом:

```
# let x = ref 2;;
val x : int ref = {contents = 2}
# !((++) x) * (1/0) ;;
Exception: Division_by_zero.
# x;;
- : int ref = {contents = 2}
# (1/0) * !((++) x) ;;
Exception: Division_by_zero.
# x;;
- : int ref = {contents = 3}
```

Переменная `x` не изменяется во время вычисления выражения (*1*), тогда как она изменяется во время (*2*). Если заранее не сохранить

начальные значения, конструкция `try .. with ..` не должна (в части `with ..`) зависеть от изменяемых переменных, которые участвуют в вычислении возбудившем исключение.

4.5.3 Изменяемые функциональные структуры данных

В функциональном программирование программа, как функциональное выражение, является в то же время данными — чтобы уяснить этот момент напишем список ассоциированных значений в виде функционального выражения. Этот список ассоциаций `('a * 'b) list` можно рассматривать как частичную функцию из `'a` (множество ключей) в `'b` (множество соответствующих значений). Другими словами, функция `'a -> 'b`.

Пустой список это неопределённая функция, которую мы будем моделировать возбуждением исключения:

```
# let nil_assoc = function x -> raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

Теперь напишем функцию `add_assoc` добавляющую элемент в список или другими словами расширим функцию новыми значениями.

```
# let add_assoc (k,v) l = function x -> if x = k then v else l x ;;
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc) ;;
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Exception: Not_found.
```

Перепишем функцию `mem_assoc`:

```
# let mem_assoc k l = try (l k) ; true with Not_found -> false ;;
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

Однако, написание функции удаляющей элементы из списка, не совсем простое дело. У нас больше нет доступа к значениям входящим в

замыкание. Для этого мы спрячем старое значение возбудив исключения `Not_found`.

```
# let rem_assoc k l = function x -> if x=k then raise Not_found else
    l x ;;
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Exception: Not_found.
```

Естественно, можно воспользоваться ссылками и побочным эффектом для использования таких значений, однако существует несколько предостережений.

```
# let add_assoc_again (k,v) l = l := (function x -> if x=k then v else
    !l x) ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
```

Мы получили функцию `l` которая указывает сама на саму себя и значит будет зацикливаться. Этот досадный побочных эффект есть результат того что разыменование `!l` находится внутри замыкания `function x ->`. Значение `!l` вычислено при выполнении, а не компиляции. В этот момент, `l` указывает на значение изменённое `add_assoc`. Необходимо исправить наше определение используя замыкание полученное определением `add_assoc`:

```
# let add_assoc_again (k, v) l = l := add_assoc (k, v) !l ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents = <fun>}
# add_assoc_again ('1',1) l ;;
- : unit = ()
# add_assoc_again ('2',2) l ;;
- : unit = ()
# !l '1' ;;
- : int = 1
# !l 'x' ;;
Exception: Not_found.
```

4.5.4 Ленивые изменяемые структуры

Смесь императивных особенностей с функциональным языком образует хорошие средства реализации языков программирования. В данном параграфе мы проиллюстрируем эту особенность в реализации структур данных с отсроченным вычислением. Такая структура данных не вычисляется полностью, вычисление продвигается в зависимости от использования структуры.

Отложенное вычисление, часто используемое в чистых функциональных языках, можно моделировать использованием функциональных значений (возможно, изменяемых). Выгода от использования данных с отложенным выполнением двойная; во первых вычисляется лишь то что необходимо для расчёта, во вторых использование потенциально бесконечных данных.

Определим тип `vm` элементы которого либо уже вычисленное значение (конструктор `Imm`) либо значение которое будет вычислено (конструктор `Deferred`).

```
# type 'a v =
    Imm of 'a
  | Deferred of (unit -> 'a);;
type 'a v = Imm of 'a | Deferred of (unit -> 'a)
# type 'a vm = {mutable c : 'a v };;
type 'a vm = { mutable c : 'a v; }
```

Откладывание вычислений может быть получено инкапсуляцией в замыкание. Функция вычисляющая такое значение должна или вернуть значение если оно уже вычислено или, в противном случае, вычислить и сохранить результат.

```
# let eval e = match e.c with
   Imm a -> a
 | Deferred f -> let u = f () in e.c <- Imm u ; u ;;
val eval : 'a vm -> 'a = <fun>
```

Операции задержки и активации вычисления также называют замораживанием и размораживанием значения.

Напишем условный контроль в виде функции:

```
# let if_deferred c e1 e2 =
   if eval c then eval e1 else eval e2;;
val if_deferred : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
```

А теперь воспользуемся этим в рекурсивной функции подсчёта факториала.

```
# let rec facr n =
    if_deferred
            {c=Deferred(fun () -> n = 0)}
            {c=Deferred(fun () -> 1)}
            {c=Deferred(fun () -> n*(facr(n-1)))};;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120
```

Заметим, что классический `if` не может быть записан в виде функции. Действительно, определим функцию `if_function` следующим образом:

```
# let if_function c e1 e2 = if c then e1 else e2;;
val if_function : bool -> 'a -> 'a -> 'a = <fun>
```

Дело в том что все три аргумента вычисляются, это приводит к зацикливанию, так как рекурсивный вызов `fact(n - 1)` всегда вычисляется, даже в случае когда `n = 0`.

```
# let rec fact n = if_function (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).
```

Модуль Lazy

Трудности во внедрении замороженных значений происходят от конструкции выражений с отложенным вычислением в контексте немедленного вычисления Objective CAML. Мы это увидели при попытке переопределить условное выражение. Нельзя написать функцию замораживающую значение при конструкции объекта типа `vm`.

```
# let freeze e = { c = Deferred (fun () -> e) };;
val freeze : 'a -> 'a vm = <fun>
```

Эта функция следует стратегии вычисления Objective CAML, то есть выражение `e` вычисляется перед тем как создать замыкание `fun () ->e`. Проиллюстрируем это в следующем примере:

```
# freeze (print_string "trace"; print_newline(); 4*5);;
```

```
trace
- : int vm = {c = Deferred <fun>}
```

По этой причине была введена следующая синтаксическая форма:
Синтаксис

```
lazy expr
```

Предупреждение

Это особенность является расширением языка и может измениться в следующих версиях.

Когда к выражению применяется ключевое слово `lazy` то создаётся значение особого типа, который определён в модуле `Lazy`:

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int lazy_t = <lazy>
```

Выражение (`print_string "Hello"`) не вычислено, так как не было никакого вывода на экран. При помощи функции `Lazy.force` мы можем вынудить вычисление выражения.

```
# Lazy.force x ;;
Hello- : int = 12
```

Тут мы замечаем, что значение x изменилось:

```
# x ;;
- : int lazy_t = <lazy>
```

Теперь это значение замороженного выражения, в данном случае 12.

Новый вызов функции `force` просто возвращает вычисленное значение:

```
# Lazy.force x ;;
- : int = 12
```

Строка `"Hello"` больше не выводится на экран.

«Бесконечные» структуры данных

Другой интерес использования отложенного вычисления состоит в возможности построения потенциально бесконечных структур данных, как на пример множество натуральных чисел. Вместо того чтобы конструировать каждое число, мы определим лишь первый элемент и способ получения следующего.

Определим настраиваемую (`generic`) структуру `'a enum` с помощью которой мы будем определять элементы множества.

```
# type 'a enum = { mutable i : 'a; f :'a -> 'a } ;;
type 'a enum = { mutable i : 'a; f : 'a -> 'a; }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

Для того, чтобы получить множество натуральных достаточно конкретизировать (`instanciating`) поля этой структуры.

```
# let nat = { i=0; f=fun x -> x + 1 };;
val nat : int enum = {i = 0; f = <fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Другой пример — ряд чисел Фибоначчи, который определён как:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_n + u_{n+1}; \end{cases}$$

Функция, вычисляющая текущее значение, должна использовать значения $u_n - 1$ и $u_n - 2$. Для этого воспользуемся состоянием `c` в следующем замыкании.

```
# let fib = let fx = let c = ref 0 in fun v -> let r = !c + v in c:=v ; r
            in { i=1 ; f=fx } ;;
val fib : int enum = {i = 1; f = <fun>}
# for i=0 to 10 do print_int (next fib); print_string "  " done ;;
1 1 2 3 5 8 13 21 34 55 89 - : unit = ()
```

4.6 Поток данных

Потоки это потенциально бесконечная последовательность данных определённого рода. Вычисление части потока выполняется по необходимости для текущего вычисления — пассивные структуры данных.

`stream` абстрактный тип данных, реализация которого неизвестна. Мы манипулируем объектами этого типа при помощи функций конструкции и деструкции. Для удобства пользователя, Objective CAML предоставляет пользователю простые синтаксические конструкции для создания и доступа к элементам потока.

Предупреждение

Это особенность является расширением языка и может измениться в следующих версиях.

4.6.1 Создание

Упрощённый синтаксис конструкции потоков похож на синтаксис конструкции списков или векторов. Пустой поток создаётся следующим способом:

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

Другой способ конструкции потока состоит в перечислении элементов этого потока, где перед каждым из них ставится апостроф.

```
# [< '0; '2; '4 >] ;;
- : int Stream.t = <abstr>
```

Выражения, перед которыми не стоит апостроф, рассматриваются как под–потоки.

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
- : int Stream.t = <abstr>
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
- : int Stream.t = <abstr>
# let concat_stream a b = [< a ; b >] ;;
val concat_stream : 'a Stream.t -> 'a Stream.t -> 'a Stream.t = <fun>
# concat_stream [< '"if"; '"c";'"then";'"1" >] [< '"else";'"2" >] ;;
- : string Stream.t = <abstr>
```

Остальные функции сгруппированы в модуле `Stream`. На пример, функции `of_channel` и `of_string` возвращают поток состоящий из символов полученных из входного потока или строки символов.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
```

```
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

Отложенное вычисление потоков позволяет использовать бесконечные структуры данных, как это было в случае с типом `'a enum` на стр. 122. Определим поток натуральных чисел при помощи первого элемента и функции вычисляющей поток из следующих элементов.

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
val nat_stream : int -> int Stream.t = <fun>
# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

4.6.2 Уничтожение и сопоставление потоков

Элементарная операция `next` одновременно вычисляет, возвращает и извлекает первый элемент потока.

```
# for i=0 to 10 do
    print_int (Stream.next nat) ;
    print_string " "
    done ;;
0 1 2 3 4 5 6 7 8 9 10 - : unit = ()
# Stream.next nat ;;
- : int = 11
```

Когда данные в потоке закончились возбуждается исключение.

```
# Stream.next [< >] ;;
Uncaught exception: Stream.Failure
```

Для использования потоков Objective CAML предоставляет специальное сопоставление для потоков — уничтожающее сопоставление (destructive matching). Сопоставляемое значение вычисляется и удаляется из потока. Понятие исчерпываемости сопоставления (exhaustive match) не существует для потоков, так как мы используем пассивные и потенциально бесконечные структуры данных. Синтаксис сопоставления следующий:

Синтаксис

```
match expr with parser [< 'p1 ...>] -> expr1 | ...
```

Функция `next` может быть переписана в следующей форме:

```
# let next s = match s with parser [< 'x >] -> x ;;
```

```
val next : 'a Stream.t -> 'a = <fun>
# next nat;;
- : int = 12
```

Заметьте, что перечисление чисел началось с того места где мы остановились.

Существует другая форма синтаксиса для фильтрации функционального параметра типа `Stream.t`.

Синтаксис

```
parser p -> ...
```

Перепишем функцию `next` используя новый синтаксис.

```
# let next = parser [<'x>] -> x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat ;;
- : int = 13
```

Мы можем сопоставлять пустой поток, но необходимо помнить о следующем: образец потока `[<>]` сопоставляется с каким угодно потоком. То есть, поток `s` всегда равен потоку `[< [<>]; s >]`. Поэтому нужно поменять обычный порядок сопоставления.

```
# let rec it_stream f s =
  match s with parser
    [< 'x ; ss >] -> f x ; it_stream f ss
  | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Используя тот факт что сопоставление уничтожающее, перепишем предыдущую функцию.

```
# let rec it_stream f s =
  match s with parser
    [< 'x >] -> f x ; it_stream f s
  | [<>] -> () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Несмотря на то что потоки пассивные, они добровольно и с «восторгом» отдают свой первый элемент, который после этого будет утерян для потока. Эта особенность отображается на сопоставлении. Следующая функция есть попытка (обречённая на неудачу) вывода на экран двух чисел из потока, в конце потока может остаться один элемент.

```
# let print_int2 n1
 n2 =
  print_string "(" ; print_int n1 ; print_string "," ;
  print_int n2 ; print_string ")" ;;
val print_int2 : int -> int -> unit = <fun>
# let rec print_stream s =
  match s with parser
    [< 'x; 'y >] -> print_int2 x y; print_stream s
  | [< 'z >] -> print_int1 z; print_stream s
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)Uncaught exception: Stream.Error("")
```

Два первых элемента потока были выведены на экран без проблем, однако во время вычисления рекурсивного вызова (`print_stream [< 3 >]`) образец обнаружил `x`, который был «употреблён», но для y в потоке ничего нет. Это и привело к ошибке. Дело в том что второй образец абсолютно бесполезный, если поток не пустой то первый образец всегда совпадёт.

Для того, чтобы получить ожидаемый результат необходимо упорядочить сопоставление.

```
# let rec print_stream s =
  match s with parser
    [< 'x >]
     -> (match s with parser
           [< 'y >] -> print_int2 x y; print_stream s
         | [<>] -> print_int1 x; print_stream s)
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)3
- : unit = ()
```

Если сопоставление не срабатывает на первом элементе образца, то фильтр работает как обычно.

```
# let rec print_stream s =
  match s with parser
    [< '1; 'y >] -> print_int2 1 y; print_stream s
  | [< 'z >] -> print_int1 z; print_stream s
  | [<>] -> print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>] ;;
(1,2)3
- : unit = ()
```

Пределы сопоставления

Из–за своего свойства уничтожения сопоставление потоков отличается от сопоставления типов сумма. Давайте рассмотрим на сколько глубоко они отличаются.

Вот простейшая функция складывающая два элемента потока.

```
# let rec sum s =
  match s with parser
    [< 'n; ss >] -> n+(sum ss)
  | [<>] -> 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

Однако, мы можем поглотить поток «изнутри» придав имя полученному результату.

```
# let rec sum s =
  match s with parser
    [< 'n; r = sum >] -> n+r
  | [<>] -> 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

В главе 11, посвящённой лексическому и синтаксическому анализу, мы рассмотрим другие примеры использования потоков. В частности, мы увидим как «поглощение» потока изнутри можно с выгодой использовать.

4.7 Exercises

4.8 Резюме

В этой главе мы сравнили функциональный и императивный стили программирования. Основные различия состоят в контроле выполнения (неявный в функциональном и явный в императивном стиле) и представлении в памяти данных (явное разделение или копирования в императивном стиле, не имеющее такой важности в функциональном). Реализация алгоритмов в функциональном и императивном стилях подразумевает эти различия. Выбор между обоими стилями на самом деле приводит к их одновременному использованию. Это позволяет явно выразить представление замыкания, оптимизировать критические части программы и создать изменяемые функциональные данные. Физическое изменение значений в окружении замыкания помогает нам лучше понять что такое функциональное значение. Одновременное использование обоих стилей предоставляет мощные средства для реализации. Мы воспользовались этим при создании потенциально бесконечных данных.

4.9 To Learn More

Глава 5

Графический интерфейс

Глава 6

Приложения

Часть II

Средства разработки

Глава 7

Компиляция и переносимость

Глава 8

Библиотеки

Глава 9

Автоматический сборщик мусора

Глава 10

Средства анализа программ

Глава 11

Средства лексического и синтаксического анализа

11.1 Введение

Определение и реализация средств лексического и синтаксического анализа являлись важным доменом исследования в информатике. Эта работа привела к созданию генераторов лексического и синтаксического анализа `lex` и `yacc`. Команды `camllex camlyacc`, которые мы представим в этой главе, являются их достойными наследниками. Два указанных инструмента стали de–facto стандартными, однако существуют другие средства, как например потоки или регулярные выражения из библиотеки `Str`, которые могут быть достаточны для простых случаев, там где не нужен мощный анализ.

Необходимость подобных инструментов особенно чувствовалась в таких доменах как компиляция языков программирования. Однако и другие программы могут с успехом использовать данные средства: базы данных, позволяющие определять запросы или электронная таблица, где содержимое ячейки можно определить как результат какой–нибудь формулы. Проще говоря, любая программа, в которой взаимодействие с пользователем осуществляется при помощи языка, использует лексический и синтаксический анализ.

Возьмём простой случай. Текстовый формат часто используется для хранения данных, будь то конфигурационный системный файл или данные табличного файла. Здесь, для использования данных, необходим лексический и синтаксический анализ.

Обобщая, скажем что лексический и синтаксический анализ преобразует линейный поток символов в данные с более богатой структурой:

последовательность слов, структура записи, абстрактное синтаксическое дерево программы и т.д.

У каждого языка есть словарный состав (лексика) и грамматика, которая описывает каким образом эти составные объединяются (синтаксис). Для того, чтобы машина или программа могли корректно обрабатывать язык, этот язык должен иметь точные лексические и синтаксические правила. У машины нет «тонкого чувства» для того чтобы правильно оценить двусмысленность натуральных языков. По этой причине к машине необходимо обращаться в соответствии с чёткими правилами, в которых нет исключений. В соответствии с этим, понятия лексики и семантики получили формальные определения, которые будут кратко представлены в данной главе.

11.2 План главы

Данная глава знакомит нас со средствами лексического и синтаксического анализа, которые входят в дистрибутив Objective CAML. Обычно синтаксический анализ следует за лексическим. В первой части мы узнаем о простом инструменте лексического анализа из модуля `Genlex`. После этого ознакомимся с формализмом рациональных выражений и тем самым рассмотрим более детально определение множества лексических единиц. А так же проиллюстрируем их реализацию в модуле `Str` и инструменте `ocamllex`. Во второй части мы определим грамматику и рассмотрим правила создания фраз языка. После этого рассмотрим два анализа фраз: восходящий и нисходящий. Они будут проиллюстрированы использованием `Stream` и `ocamlyacc`. В приведённых примерах используется контекстно–независимая грамматика. Здесь мы узнаем как реализовать контекстный анализ при помощи Stream. В третьей части мы вернёмся к интерпретатору BASIC (см. стр **??**) и при помощи `ocamllex` и `ocamlyacc` добавим лексические и синтаксические функции анализа языка.

11.3 Лексика

Синтаксический анализ это предварительный и необходимый этап обработки последовательностей символов: он разделяет этот поток в последовательность слов, так называемых лексические единицы или лексемы.

11.3.1 Модуль `Genlex`

В данном модуле имеется элементарное средство для анализа символьного потока. Для этого используются несколько категорий предопределённых лексических единиц. Эти категории различаются по типу:

```
# type token = Kwd of string
    | Ident of string
    | Int of int
    | Float of float
    | String of string
    | Char of char ;;
```

Таким образом мы можем распознать в потоке символов целое число (конструктор `Int`) и получить его значение (аргумент конструктора `int`). Распознаваемые символы и строки подчиняются следующим общепринятым соглашением: строка окружена символами ("), а символ окружён (’). Десятичное число представлено либо используя запись с точкой (например 0.01), либо с мантиссой и экспонентой (на пример 1E-2). Кроме этого остались конструкторы `Kwd` и `Ident`.

Конструктор `Ident` предназначен для определения идентификаторов. Идентификатором может быть имя переменной или функции языка программирования. Они состоят из любой последовательности букв и цифр, могут включать символ подчёркивания (_) или апостроф (’). Данная последовательность не должна начинаться с цифры. Любая последовательность следующих операндов тоже будет считаться идентификатором: +,*,> или -. И наконец, конструктор `Kwd` определяет категорию специальных идентификаторов или символов.

Категория ключевых слова — единственная из этого множества, которую можно сконфигурировать. Для того, чтобы создать лексический анализатор, воспользуемся следующей конструкцией, которой необходимо передать список ключевых слов на место первого аргумента.

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

Тем самым получаем функцию, которая принимает на вход поток символов и возвращает поток лексических единиц (с типом `token`).

Таким образом, мы без труда реализуем лексический анализатор для интерпретатора BASIC. Объявим множество ключевых слов:

```
# let keywords =
    [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN"; "-";
    "!"; "+"; "-"; "*"; "/"; "%";
```

```
    "="; "<"; ">"; "<="; ">="; "<>";
    "&"; "|" ] ;;
```

При помощи данного множества, определим функцию лексического анализа:

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

Приведённая функция `line_lexer`, из входящего потока символов создаёт поток соответствующих лексем.

11.3.2 Использование потоков

Мы также можем реализовать лексический анализ «в ручную» используя потоки.

В следующем примере определён лексический анализатор арифметических выражений. Функции lexer передаётся поток символов из которого она создаёт поток лексических единиц с типом `lexeme Stream.t` [1]. Символы пробел, табуляция и переход на новую строку удаляются. Для упрощения, мы не будем обрабатывать переменные и отрицательны целые числа.

```
# let rec spaces s =
    match s with parser
      [<'' ' ; rest >] -> spaces rest
      | [<''\t' ; rest >] -> spaces rest
      | [<''\n' ; rest >] -> spaces rest
      | [<>] -> ();;
val spaces : char Stream.t -> unit = <fun>
# let rec lexer s =
   spaces s;
   match s with parser
     [< ''(' >] -> [< 'Lsymbol "("; lexer s >]
     | [< '')' >] -> [< 'Lsymbol ")"; lexer s >]
     | [< ''+' >] -> [< 'Lsymbol "+" ; lexer s >]
     | [< ''-' >] -> [< 'Lsymbol "-" ; lexer s >]
     | [< ''*' >] -> [< 'Lsymbol "*" ; lexer s >]
     | [< ''/' >] -> [< 'Lsymbol "/" ; lexer s >]
```

[1] тип `lexeme` определён на стр. **??**

```
    | [< ''0'..'9' as c;
      i,v = lexint (Char.code c - Char.code('0')) >]
      ->[<'Lint i ; lexer v>]
  and lexint r s =
    match s with parser
      [< ''0'..'9' as c >]
      -> let u = (Char.code c) - (Char.code '0') in lexint (10*r + u) s
      | [<>] -> r,s ;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t = <fun>
```

Функция `lexint` предназначена для анализа той части потока символов, которая соответствует числовой постоянной. Она вызывается, когда функция `lexer` встречает цифры. В этом случае функция `lexint` поглощает все последовательные цифры и выдаёт соответствующее значение полученного числа.

11.3.3 Регулярные выражения

Оставим ненадолго практику и рассмотрим проблему лексических единиц с теоретической точки зрения.

Лексическая единица является словом. Слово образуется при конкатенации элементов алфавита. В нашем случае алфавитом является множество символов ASCII.

Теоретически, слово может вообще не содержать символов (пустое слово [2]) или состоять из одного символа.

Теоретические исследования конкатенации элементов алфавита для образования лексических элементов (лексем) привели к созданию простого формализма, известного как регулярные выражения.

Определение

Регулярные выражения позволяют определить множества слов. Пример такого множества: идентификаторы. Принцип определения основан на некоторых теоретико–множественный операциях. Пусть M и N — два множества слов, тогда мы можем определить:

- объединение M и N, записываемое $M|N$.
- дополнение M, записываемое $\wedge M$: множество всех слов, кроме тех, которые входят в M.

[2] традиционно, такое слово обозначается греческой буквой эпсилон: ε

- конкатенация M и N: множество всех слов созданных конкатенацией слова из M и слова из N. Записывается просто MN.

- мы можем повторить операцию конкатенации слов множества M и тем самым получить множество слов образованных из конечной последовательности слов множеств M. Такое множество записывается $M+$. Он содержит все слова множества M, все слова полученные конкатенацией двух слов множества M, трёх слов, и т.д. Если мы желаем чтобы данное множество содержало пустое слово, необходимо писать $M*$.

- для удобства, существует дополнительная конструкция $M?$, которая включает все слова множества M, а так же пустое слово.

Один единственный символ ассоциируется с одно элементным множеством. В таком случае выражение *a| b| c* описывает множество состоящее из трёх слов *a*, *b* и *c*. Существует более компактная запись: *[abc]*. Так как наш алфавит является упорядоченным (по порядку кодов ASCII), можно определить интервал. Например, множество цифр запишется как $[0-9]$. Для группировки выражений можно использовать скобки.

Для того, чтобы использовать в записи сами символы–операторы, как обычные символы, необходимо ставить перед ними обратную косую черту: \. Например множество $(\backslash *)*$ обозначает множество последовательностей звёздочек.

Пример

Пусть существует множество из цифр (0,1,2,3,4,5,6,7,8,9), символы плюс $(+)$ и минус $(-)$, точки (.) и буквы E. Теперь мы можем определить множество чисел *num*. Назовем *integers* множество определённое выражением $[0-9]+$. Множество неотрицательных чисел *unum* определяется так:

$$integers?(.integers)?(E(\backslash + | -)?integers)?$$

Множество отрицательных и положительных чисел записывается:

$$unum| - unum \quad \text{или} \quad -?unum$$

Распознавание

Теперь, после того как множество выражений определено, остаётся проблема распознавания принадлежности строки символов или одной из её

подстрок этому множеству. Для решения данной задачи необходимо реализовать программу обработки выражений, которая соответствует формальным определениям множества. Для регулярных выражений такая обработка может быть автоматизирована. Подобная автоматизация реализована в модуле `Genlex` из библиотеки `Str` и инструментом `ocamllex`, которые будут представлены в следующих двух параграфах.

Библиотека `Str`

В данном модуле имеется абстрактный тип `regexp` и функция `regexp`. Указанный тип представляет регулярные выражения, а функция трансформирует регулярное выражение, представленное в виде строки символов, в абстрактное представление.

Модуль `Str` содержит несколько функций, которые используют регулярные выражения и манипулируют символьными строками. Синтаксис регулярных выражений библиотеки `Str` приведён в таблице 11.1.

Таблица 11.1: Регулярные выражения

.	любой символ, кроме \
$*$	ноль или несколько экземпляров предыдущего выражения
$+$	хотя бы один экземпляр предыдущего выражения
?	ноль или один экземпляр предыдущего выражения
[..]	множество символов
.	любой символ, кроме \
	интервал записывается при помощи - (пример $[0-9]$)
	дополнение записывается при помощи $\wedge$ (пример $[\wedge A-Z]$)
$\wedge$	начало строки (не путать с дополнением $\wedge$)
$	конец строки
|	вариант
(..)	группировка в одно выражение (можно ссылаться на это выражение)
i	числовая константа i ссылается на i–ый элемент группированного выражения
\	забой, используется для сопоставления зарезервированных символов в регулярных выражениях

Пример

В следующем примере напишем функцию, которая переводит дату из английского формата во французский. Предполагается, что входной файл

состоит из строк, разбитых на поля данных и элементы даты разделяются точкой. Определим функцию, которая из полученной строки (строка файла), выделяет дату, разбивает её на части, переводит во французских формат и тем самым заменяет старую дату на новую.

```
# let french_date_of d =
  match d with
   [mm; dd; yy] -> dd^"/"^mm^"/"^yy
   | _ -> failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
  try
  let i=Str.search_forward english_date_format l 0 in
  let d1 = Str.matched_string l in
  let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
    Str.global_replace english_date_format d2 l
  with Not_found -> l ;;
  val trans_date : string -> string = <fun>

# trans_date
".............06.13.99..........." ;;
- : string = ".............13/06/99..........."
```

11.3.4 Инструмент `Ocamllex`

`ocamllex` — это лексический генератор созданный для Objective CAML по модели `lex`, написанном на языке C. При помощи файла, описывающего элементы лексики в виде множества регулярных выражений, которые необходимо распознать, он создаёт файл–исходник на Objective CAML. К описанию каждого лексического элемента можно привязать какое–нибудь действие, называемое семантическое действие. В полученном коде используется абстрактный тип `lexbuf` из модуля `Lexing`. В данном модуле также имеется несколько функций управления лексическими буферами, которые могут быть использованы программистом для того чтобы определить необходимые действия.

Обычно, файлы, описывающие лексику, имеют расширение `.mll`. Для того, чтобы из файла `lex_file.mll` получить файл на Objective CAML,

необходимо выполнить следующую команду:

```
ocamllex lex_file.mll
```

После этого, мы получим файл `lex_file.ml`, содержащий код лексического анализатора. Теперь, данный файл можно использовать в программе на Objective CAML. Каждому множеству правил анализа соответствует функция, которая принимает лексический буфер (типа `Lexing.lexbuf`) и затем возвращает значение, определённое семантическим действием. Значит, все действия для определённого правила должны создавать значение одного и того же типа.

Формат у файла для `ocamllex` следующий:

```
{
  header
}
let ident = regexp
          ...
rule ruleset1 = parse
                  regexp { action }
                | ...
                | regexp { action }
and ruleset2 = parse
          ...
and ...
{
  trailer-and-end
}
```

Части «заголовок» и «продолжение-и-конец» не являются обязательными. Здесь вставляется код Objective CAML, определяющий типы данных, функции и т.д. необходимые для обработки данных. В последней части используются функции, которые используют правила анализа множества лексических данных из средней части. Серия объявлений, которая предшествует определению правил, позволяет дать имя некоторым регулярным выражениям. Эти имена будут использоваться в определении правил.

Пример

Вернёмся к нашему интерпретатору BASIC и усовершенствуем тип возвращаемых лексических единиц. Таким образом, мы можем воспользо-

ваться функцией `lexer`, (см. стр. **??**) которая возвращает такой же тип результата (`lexeme`), но на входе она получает буфер типа `Lexing.lexbuf`.

```
  {
    let string_chars s =
    String.sub s 1 ((String.length s)-2) ;;
  }

let op_ar = ['-' '+' '*' '\%' '/']
let op_bool = ['!' '\&' '|']
let rel = ['=' '<' '>']

rule lexer = parse
    [' '] { lexer lexbuf }
  | op_ar { Lsymbol (Lexing.lexeme lexbuf) }
  | op_bool { Lsymbol (Lexing.lexeme lexbuf) }
  | "<=" { Lsymbol (Lexing.lexeme lexbuf) }
  | ">=" { Lsymbol (Lexing.lexeme lexbuf) }
  | "<>" { Lsymbol (Lexing.lexeme lexbuf) }
  | rel { Lsymbol (Lexing.lexeme lexbuf) }
  | "REM" { Lsymbol (Lexing.lexeme lexbuf) }
  | "LET" { Lsymbol (Lexing.lexeme lexbuf) }
  | "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
  | "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
  | "IF" { Lsymbol (Lexing.lexeme lexbuf) }
  | "THEN" { Lsymbol (Lexing.lexeme lexbuf) }
  | '-'? ['0'-'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
  | ['A'-'z']+ { Lident (Lexing.lexeme lexbuf) }
  | '"' [^ '"']* '"' { Lstring (string_chars (Lexing.lexeme lexbuf
    )) }
```

После обработки данного файла командой `ocamllex` получим функцию `lexer` типа `Lexing.lexbuf -> lexeme`. Далее, мы рассмотрим каким образом подобные функции используются в синтаксическом анализе (см. стр. **??**).

11.4 Синтаксис

Благодаря лексическому анализу, мы в состоянии разбить поток символов на более структурированные элементы: лексические элементы. Теперь необходимо знать как правильно объединять эти элементы в син-

таксически корректные фразы какого-нибудь языка. Правила синтаксической группировки определены посредством грамматических правил. Формализм, произошедший из лингвистики, был с успехом перенят математиками, занимающимися теориями языков, и специалистами по информатике. На странице **??** мы уже видели пример грамматики для языка BASIC. Здесь мы снова вернёмся к этому примеру, чтобы более углублённо ознакомится с базовыми концепциями грамматики.

11.4.1 Грамматика

Говоря формальным языком, грамматика основывается на четырёх элементах:

1. Множество символов, называемых терминалами. Эти символы являются лексическими элементами языка. В Бэйсике к ним относятся символы операторов, арифметических отношений и логические ($+$, &, $<$, $\leq$, $\cdots$), ключевые слова языка **(GOTO, PRINT, IF, THEN,** $\cdots$**)**, целые числа (элемент `integer`) и переменные (элемент `variable`).

2. Множество нетерминальных символов, которые представляют синтаксические компоненты языка. Например, программа на языке Бэйсик состоит из строк. Таким образом мы имеем компоненту Line, которая в свою очередь состоит из выражений (Expression), и т.д.

3. Множество так называемых порождающих правил. Они описывают каким образом комбинируются терминальные и нетерминальные символы, чтобы создать синтаксическую компоненту. Строка в Бэйсике начинается с номера, за которой следует инструкция. Смысл правила следующий:

   $$\text{Line} ::= \textit{integer}\ \text{Instruction}$$

   Одна и та же компонента может быть порождена несколькими способами. В этом случае варианты разделяются символом | как в:

   $$\begin{array}{rcl} \text{Instruction} & ::= & \text{LET variable} = \text{Instruction} \\ & | & \text{GOTO integer} \\ & | & \text{PRINT Expression} \\ \text{etc} & & \end{array}$$

Среди всех нетерминальных символов различают один, называемый начальным. Правило, которое порождает эту аксиому является порождающим правилом всего языка.

11.4.2 Порождение и распознавание

С помощью порождающих правил можно определить принадлежит ли последовательность лексем языку.

Рассмотрим простой язык, описывающий арифметические выражения:

$$\begin{array}{lcll} \text{Exp} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{Exp} + \text{Exp} & \text{(R2)} \\ & | & \text{Exp} * \text{Exp} & \text{(R3)} \\ & | & (\ \text{Exp}\ ) & \text{(R4)} \end{array}$$

здесь *(R1) (R2) (R3) (R4)* — имена правил. По окончании лексического анализа выражение $1 * (2 + 3)$ становится последовательностью следующих лексем:

*integer * (integer + integer)*

Для того, чтобы проанализировать эту фразу и убедиться в том что она принадлежит языку арифметических выражений, воспользуемся правилами справа налево: если часть выражения соответствует правому члену какого-нибудь правила, мы заменяем это выражение соответствующим левым членом. Этот процесс продолжается до тех пор, пока выражение не будет редуцировано до аксиомы. Ниже представлен результат такого анализа [3]:

$$\begin{array}{lll} \text{integer} * (\text{integer} + \text{integer}) & \text{(R1)} & \text{Exp} * (\text{integer} + \text{integer}) \\ & \text{(R1)} & \text{Exp} * (\text{Exp} + \text{integer}) \\ & \text{(R1)} & \text{Exp} * (\text{Exp} + \text{Exp}) \\ & \text{(R2)} & \text{Exp} * (\text{Exp}) \\ & \text{(R4)} & \text{Exp} * \text{Exp} \\ & \text{(R3)} & \text{Exp} \end{array}$$

Начиная с последней линии, которая содержит лишь Exp и следуя стрелкам, можно определить каким образом было полученное выражение исходя из аксиомы Exp. Соответственно данная фраза является правильно сформированной фразой языка арифметических выражений, определённого грамматикой.

[3] анализируемая часть выражения подчёркнута, а так же указано используемое правило

Преобразование грамматики в программу, способную распознать принадлежность последовательности лексем языку, который определён грамматикой, является более сложной проблемой, чем проблема использования регулярных выражений. Действительно, в соответствии с математическим результатом любое множество (слов), определённое формализмом регулярных выражений, может быть определено другим формализмом: детерминированные конечные автоматы. Такие автоматы легко реализуются программами, принимающими поток символов. Подобный результат для грамматик в общем не существует. Однако, имеются менее строгие (?) (weaker) результаты устанавливающие эквивалентность между определёнными классами грамматик и более богатыми автоматами: автомат со стеком. Здесь мы не станем вдаваться ни в детали этих результатов, ни в точное определение таких автоматов. Однако, мы можем определить какие классы грамматики могут использоваться в средствах генерации синтаксических анализаторов или для реализации напрямую анализатора.

11.4.3 Нисходящий анализ

Разбор выражения $1 * (2 + 3)$ в предыдущем параграфе не является единственным: мы с таким же успехом могли бы начать редуцирование *integer*, то есть воспользоваться правилом *(R2)* редуцирования $2 + 3$. Эти два способа распознавания являются двумя типами анализа: восходящий анализ (справа налево) и нисходящий слева направо. Последний анализ легко реализуется при помощи потоков лексем модуля `Stream`. Средство `ocamlyacc` использует восходящий анализ, при котором применяется стек, как это уже было проиллюстрировано синтаксическим анализатором программ на Бэйсике. Выбор анализа не просто дело вкуса, в зависимости от используемой для спецификации языка формы грамматики, можно или нет применять нисходящий анализ.

Простой случай

Каноническим примером нисходящего анализа является префиксная запись арифметических выражений, определяемая как:

$$\begin{array}{rcl} \text{Exp} & ::= & \text{integer} \\ & | & + \text{ Exp Exp} \\ & | & * \text{ Exp Exp} \end{array}$$

В данном случае достаточно знать первую лексему, для того чтобы определить какое правило может быть использовано. При помощи по-

добной предсказуемости нет необходимости явно управлять стеком, достаточно положиться на рекурсивный вызов анализатора. И тогда при помощи `Genlex` и `Stream` очень просто написать программу реализующую нисходящий анализ. Функция `infix_of` из полученного префиксного выражения возвращает его инфиксный эквивалент:

```
# let lexer s =
    let ll = Genlex.make_lexer ["+";"*"]
    in ll (Stream.of_string s);;
val lexer : string -> Genlex.token Stream.t = <fun>

# let rec stream_parse s =
    match s with parser
      [<'Genlex.Ident x>] -> x
    | [<'Genlex.Int n>] -> string_of_int n
    | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] -> "("^
        e1^"+"^e2^")"
    | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] -> "("^
        e1^"*"^e2^")"
    | [<>] -> failwith "Parse error" ;;
val stream_parse : Genlex.token Stream.t -> string = <fun>

# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>

# infix_of "* +3 11 22";;
- : string = "((3+11)*22)"
```

Однако не стоит забывать о некоторой примитивности лексического анализа. Советуем периодически добавлять пробелы между различными лексическими элементами.

```
# infix_of "*+3 11 22";;
- : string = "*+"
```

Случай посложней

Синтаксический анализ при помощи потоков предсказуем, он облагает грамматику двумя условиями:

- В правилах грамматики не должно быть левой рекурсии. Правило называется рекурсивным слева, если его правый член начинается с

нетерминального символа, который является левой частью правила. Например: `Exp ::= Exp + Exp`

- Не должно существовать правил начинающихся одним и тем же выражением.

Грамматика арифметических выражений, приведённая на стр. 154, не подходит для нисходящего анализа: они не удовлетворяют ни одному из условий. Для того, чтобы применить нисходящий анализ необходимо переформулировать грамматику таким образом, чтобы удалить левую рекурсию и неопределённость правил. Вот полученный результат:

$$
\begin{array}{rcl}
\text{Exp} & ::= & \text{Atom NextExpr} \\
\text{NextExpr} & ::= & + \text{ Atom} \\
& | & - \text{ Atom} \\
& | & * \text{ Atom} \\
& | & / \text{ Atom} \\
& | & \varepsilon \\
\text{Atom} & ::= & \text{integer} \\
& | & (\text{ Expr })
\end{array}
$$

Заметьте использование пустого слова ε в определении `NextExpr`. Оно необходимо, если мы хотим чтобы просто целое число являлось выражением.

Следующий анализатор есть просто перевод вышеуказанной грамматики в код. Он реализует абстрактное синтаксическое дерево арифметических выражений.

```
# let rec rest = parser
    [< 'Lsymbol "+"; e2 = atom >] -> Some (PLUS,e2)
  | [< 'Lsymbol "-"; e2 = atom >] -> Some (MINUS,e2)
  | [< 'Lsymbol "*"; e2 = atom >] -> Some (MULT,e2)
  | [< 'Lsymbol "/"; e2 = atom >] -> Some (DIV,e2)
  | [< >] -> None
 and atom = parser
     [< 'Lint i >] -> ExpInt i
   | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] -> e
 and expr s =
   match s with parser
     [< e1 = atom >] ->
       match rest s with
           None -> e1
```

```
        | Some (op,e2) -> ExpBin(e1,op,e2) ;;
val rest : lexeme Stream.t -> (bin_op * expression) option = <fun>
val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>
```

Сложность использования нисходящего анализа заключается в том, что грамматика должна быть очень ограниченной формы. Если язык выражен естественно с использованием левой рекурсии (как в инфиксных выражениях), то не всегда легко определить эквивалентную грамматику, то есть определяющую такой же язык, которая бы удовлетворяла требованиям нисходящего анализа. По этой причине, средства `yacc` и `ocamlyacc` реализуют восходящий анализ, который разрешает определение более естественных грамматик. Однако, мы увидим, что даже в этом случае существуют ограничения.

11.4.4 Восходящий анализ

Мы уже вкратце представили на странице **??** принципы восходящего анализа: сдвиг и вывод [4]. После каждого подобного действия, состояние стека изменяется. Из этой последовательности можно вывести правила грамматики, в случае если грамматика это позволяет, как в примере с нисходящим анализом. Опять же, сложности возникают из-за неопределённости правил, когда невозможно выбрать между продвинутся или сократить. Проиллюстрируем действие восходящего анализа и его недостатки на все тех же арифметических выражениях в постфиксном и инфиксном написании.

Положительная сторона

Упрощённая постфиксная грамматика арифметических выражений выглядит так:

$$\begin{array}{lcll} \text{Exp} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{Exp Exp +} & \text{(R2)} \\ & | & \text{Exp Exp -} & \text{(R3)} \end{array}$$

Данная грамматика является двойственной по отношению к префиксной: для того чтобы точно знать какое правило следует применить, необходимо дождаться окончания анализа. В действительности анализ по-

[4] Обычно на русский язык термины **shift** и **reduce** переводят как сдвиг и вывод, но в данной книге для термина **reduce** будет также использоваться перевод «сокращение» — прим. пер.

добных выражений схож с вычислением при помощи стека. Только вместо проталкивания результата вычисления, проталкиваются грамматические символы. Если в начале стек пустой, то после того как ввод закончен, необходимо получить стек содержащий лишь нетерминальную аксиому. Приведём изменение стека: если мы продвигаемся, то проталкивается текущий нетерминальный символ; если сокращаем, то первые символы стека соответствуют правому члену (в обратном порядке) правила и тогда мы заменяем эти элементы соответствующими нетерминальными элементами.

В таблице 11.2 приведён восходящий анализ выражения 1 2 + 3 * 4 +. Считываемая лексическая единица подчёркивается для более удобного чтения. Конец потока помечается символом $.

Таблица 11.2: Восходящий анализ

Действие	Вход	Стек
	1 2 + 3 * 4 + $	[]
Сдвиг		
	2 + 3 * 4 + $	[1]
Сократить *(R1)*		
	2 + 3 * 4 + $	[Exp]
Сдвиг		
	+ 3 * 4 + $	[2 Exp]
Сдвиг, Сократить *(R1)*		
	+ 3 * 4 + $	[Exp Exp]
Сдвиг, Сократить *(R2)*		
	3 * 4 + $	[Exp]
Сдвиг, Сократить *(R1)*		
	* 4 + $	[Exp Exp]
Сдвиг, Сократить *(R3)*		
	4 + $	[Exp]
Сдвиг, Сократить *(R1)*		
	+ $	[Exp Exp]
Сдвиг, Сократить *(R2)*		
	$	[Exp]

Отрицательная сторона

Вся трудность перехода от грамматики к программе распознающей язык заключается в определении действия, которое необходимо применить. Проиллюстрируем эту проблему на трёх примерах, приводящих к трём неопределённостям.

Первый пример есть грамматика выражений использующих операцию сложения:

$$\begin{array}{lcll} \text{E0} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E0} + \text{E0} & \text{(R2)} \end{array}$$

Неопределённость данной грамматики проявляется при использовании правила *R2*. Предположим следующую ситуацию:

Действие	Вход	Стек
:		
	$\underline{+}$	E0 + E0 ...
:		

В подобном случае невозможно определить необходимо сдвинуть и протолкнуть в стек + или сократить в соответствии с правилом *(R2)* оба E0 и присутствующий в стеке +. Подобная ситуация называется конфликтом сдвиг-вывод (shift/reduce). Она является следствием того, что выражение *integer* + *integer* + *integer* может быть выведено справа двумя способами.

Первый вариант:

$$\begin{array}{ll} \text{E0} & (\textit{R2})\ \text{E0} + \underline{\text{E0}} \\ & (\textit{R1})\ \underline{\text{E0}} + \textit{integer} \\ & (\textit{R2})\ \text{E0} + \underline{\text{E0}} + \textit{integer} \end{array}$$

Второй вариант:

$$\begin{array}{ll} \text{E0} & (\textit{R2})\ \text{E0} + \underline{\text{E0}} \\ & (\textit{R1})\ \text{E0} + \text{E0} + \underline{\text{E0}} + \textit{integer} \\ & (\textit{R2})\ \text{E0} + \underline{\text{E0}} + \textit{integer} \end{array}$$

Выражения, полученные двумя выводами, могут показаться одинаковыми с точки зрения вычисления выражения,

(*integer* + *integer*) + *integer* и *integer* + (*integer* + *integer*)

но разными для конструкции синтаксического дерева (см. рис. **??** на стр. **??**)

Второй пример грамматики, порождающей конфликт между сдвиг-вывод, содержит такую же неопределённость: явное заключение в скобки. Но в отличии от предыдущего случая, выбор сдвиг-вывод изменяет смысл выражения. Пусть есть грамматика:

$$\begin{array}{lcl} \text{E1} & ::= & \textit{integer}\ (\textit{R1}) \\ & | & \text{E1} + \text{E1}\ (\textit{R2}) \\ & | & \text{E1} * \text{E1}\ (\textit{R3}) \end{array}$$

Здесь мы снова получаем предыдущий конфликт как в случае с + так и для *, но к этому добавляется другой, между + и *. Опять же, одно и то же выражение может быть получено двумя способами, так как у него существует два вывода справа:

integer + *integer* * *integer*

Первый вариант:

$$\begin{array}{ll} \text{E1} & (\textit{R3})\ \text{E0} * \underline{\text{E1}} \\ & (\textit{R1})\ \underline{\text{E1}} * \textit{integer} \\ & (\textit{R2})\ \text{E1} + \underline{\text{E1}} * \textit{integer} \end{array}$$

Второй вариант:

$$\begin{array}{ll} \text{E1} & (\textit{R2})\ \text{E1} + \underline{\text{E1}} \\ & (\textit{R2})\ \text{E1} + \text{E1} * \underline{\text{E1}} \\ & (\textit{R1})\ \text{E1} + \underline{\text{E1}} * \textit{integer} \end{array}$$

В данном выражении обе пары скобок имеют разный смысл:

(*integer* + *integer*) * *integer* $\neq$ *integer* + (*integer* * *integer*)

Подобную проблему, мы уже встречали в выражениях Basic (см. стр. **??**). Она была разрешена при помощи приоритетов, которые присваиваются операторам: сначала редуцируется правило (R3), затем (R2), что соответствует заключению в скобки произведения.

Данную проблему выбора между + и * можно решить изменив грамматику. Для того, введём два новых терминальных символа: член T (*term*) и множитель F (*factor*). Отсюда получаем:

$$\begin{array}{lcll} \text{E} & ::= & \text{E} + \text{T} & (\textit{R1}) \\ & | & \text{T} & (\textit{R2}) \\ \text{T} & ::= & \text{T} + \text{F} & (\textit{R3}) \\ & | & \text{F} & (\textit{R4}) \\ \text{F} & ::= & \text{T} + \textit{integer} & (\textit{R5}) \end{array}$$

После этого, единственный способ получить *integer* + *integer* * *integer*: посредством правила (*R1*).

Третий и последний случай касается условных конструкций языка программирования. На пример в Pascal существует две конструкции: `if .. then` и `if .. then .. else`. Пусть существует следующая грамматика:

$$\begin{array}{lcll} \text{Instr} & ::= & \textit{if}\ \text{EXP}\ \textit{then}\ \text{Instr} & (R1) \\ & - & \textit{if}\ \text{EXP}\ \textit{then}\ \text{Instr}\ \textit{else}\ \text{Instr} & (R2) \\ & - & \text{etc}\ \ldots & \end{array}$$

И в следующей ситуации:

Действие	Вход	Стек
:		
	else	[Instr *then* Epx *if* ...]

Невозможно определить соответствуют ли элементы стека правила (*R1*) и в этом случае необходимо сократить или соответствуют первому `Instr` правила (*R2*) и тогда необходимо сдвинуть.

Кроме конфликтов сдвиг-вывод, восходящий анализ вызывает конфликт вывод-вывод.

Мы представим здесь инструмент `ocamlyacc`, который использует подобную технику может встретить указанные конфликты.

11.5 Пересмотренный Basic

Теперь, используя совместно `ocamllex` и `ocamlyacc`, заменим функцию `parse` для Бэйсика, приведённую на странице **??**, на функции полученные при помощи файлов спецификации лексики и синтаксиса языка.

Для этого, мы не сможем воспользоваться типами лексических единиц, в таком виде как они были определены. Необходимо определить более точные типы, чтобы различать операторы, команды и ключевые слова.

Так же, нам понадобится изолировать в отдельном файле `basic_types.mli` декларации типов, относящиеся к абстрактному синтаксису. В нем будут содержатся декларации типа `sentences`, а так же других типы необходимые этому.

11.5.1 Файл `basic_parser.mly`

Заголовок

Данный файл содержит вызовы деклараций типов абстрактного синтаксиса и две функции перевода строк символов в их эквивалент абстрактного синтаксиса.

```
%{
open Basic_types ;;

let phrase_of_cmd c =
 match c with
   "RUN" -> Run
 | "LIST" -> List
 | "END" -> End
 | _ -> failwith "line : unexpected command"
;;

let bin_op_of_rel r =
 match r with
   "=" -> EQUAL
 | "<" -> INF
 | "<=" -> INFEQ
 | ">" -> SUP
 | ">=" -> SUPEQ
 | "<>" -> DIFF
 | _ -> failwith "line : unexpected relation symbol"
;;

%}
```

Декларации

Здесь содержится три части: декларация лексем, декларация правил ассоциативности и приоритетов, декларация стартового символа `line`, которая соответствует анализу линии программы или команды.

Ниже представлены лексические единицы:

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
```

```
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

Имена деклараций говорят сами за себя и они описаны в файле `basic_lexer.mll` (см. стр. **??**).

Правила приоритета операторов схожи со значениями, которые определяются функциями `priority_uop` и `priority_binop`, которые были определены грамматикой Бейсика (см. стр. **??**).

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

Символ `Lop` необходим для обработки унарных минусов. Он не является терминальным символом, а «псевдо–терминальным». Благодаря этому, получаем перегрузку операторов, когда в двух случаях использования одного и того же оператора, приоритет меняется в зависимости от контекста. Мы вернёмся к этому случаю, когда будем рассматривать правила грамматики.

Здесь нетерминалом является `line`. Полученная функция возвращает дерево абстрактного синтаксиса, которое соответствует проанализированной линии.

```
%start line
%type <Basic_types.phrase> line
```

Правила грамматики

Грамматика делится на 3 нетерминальных элемента: `line` для линии, `inst` для инструкции и `exp` для выражений. Действия, которые привязаны к каждому правилу лишь создают соответствующую часть абстрактного синтаксиса.

```
 %%
line :
    Lint inst Leol { Line {num=$1; inst=$2} }
  | Lcmd Leol { phrase_of_cmd $1 }
  ;

inst :
    Lrem { Rem $1 }
  | Lgoto Lint { Goto $2 }
  | Lprint exp { Print $2 }
  | Linput Lident { Input $2 }
  | Lif exp Lthen Lint { If ($2, $4) }
  | Llet Lident Lequal exp { Let ($2, $4) }
  ;

exp :
    Lint { ExpInt $1 }
  | Lident { ExpVar $1 }
  | Lstring { ExpStr $1 }
  | Lneg exp { ExpUnr (NOT, $2) }
  | exp Lplus exp { ExpBin ($1, PLUS, $3) }
  | exp Lminus exp { ExpBin ($1, MINUS, $3) }
  | exp Lmult exp { ExpBin ($1, MULT, $3) }
  | exp Ldiv exp { ExpBin ($1, DIV, $3) }
  | exp Lmod exp { ExpBin ($1, MOD, $3) }
  | exp Lequal exp { ExpBin ($1, EQUAL, $3) }
  | exp Lrel exp { ExpBin ($1, (bin_op_of_rel $2), $3) }
  | exp Land exp { ExpBin ($1, AND, $3) }
  | exp Lor exp { ExpBin ($1, OR, $3) }
  | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
  | Lpar exp Rpar { $2 }
  ;
  %%
```

Данные правила не нуждаются в особых комментариях, кроме следующего:

```
exp :
  ...
  | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
```

Это правило касается использования унарного минуса -. Ключевое слово `%prec` означает, что указанная конструкция получает приоритет от `Lop` (в данном случае наивысший).

11.5.2 Файл `basic_lexer.mll`

Лексический анализ содержит лишь одно множество: `lexer`, которое точно соответствует старой функции `lexer` (см. стр. **??**).

Семантическое действие, которое связано с распознаванием лексических единиц, возвращает результат соответствующего конструктора. Необходимо загрузить файл синтаксических правил, так как в нем декларируется тип лексических единиц. Добавим так же функцию, которая удаляет кавычки вокруг строк.

```
{
 open Basic_parser ;;

 let string_chars s = String.sub s 1 ((String.length s)−2) ;;
}

rule lexer = parse
    [' ' '\t'] { lexer lexbuf }

 | '\n' { Leol }

 | '!' { Lneg }
 | '&' { Land }
 | '|' { Lor }
 | '=' { Lequal }
 | '%' { Lmod }
 | '+' { Lplus }
 | '−' { Lminus }
 | '*' { Lmult }
 | '/' { Ldiv }

 | ['<' '>'] { Lrel (Lexing.lexeme lexbuf) }
 | "<=" { Lrel (Lexing.lexeme lexbuf) }
 | ">=" { Lrel (Lexing.lexeme lexbuf) }

 | "REM" [^ '\n']* { Lrem (Lexing.lexeme lexbuf) }
 | "LET" { Llet }
```

```
| "PRINT" { Lprint }
| "INPUT" { Linput }
| "IF" { Lif }
| "THEN" { Lthen }
| "GOTO" { Lgoto }

| "RUN" { Lcmd (Lexing.lexeme lexbuf) }
| "LIST" { Lcmd (Lexing.lexeme lexbuf) }
| "END" { Lcmd (Lexing.lexeme lexbuf) }

| ['0'−'9']+ { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'−'z']+ { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"' { Lstring (string_chars (Lexing.lexeme lexbuf)
    ) }
```

Заметьте, что нам пришлось изолировать символ =, который используется одновременно в выражениях и приравниваниях.

Для двух рациональных выражений необходимо привести определённые объяснения. Линия комментариев соответствует выражению `("REM"[^'\n']*)`, где за ключевым словом `REM` следует какое угодно количество символов и затем перевод строки. Правило, которое соответствует символьным строкам, `('\ [^ '\]* '\)`, подразумевает последовательность символов, отличных от "и заключённых в кавычки ".

11.5.3 Компиляция, компоновка

Компиляция должна быть реализована в определённом порядке. Это связано с взаимозависимостью деклараций лексем. Поэтому в нашем случае, необходимо выполнить команды в следующем порядке:

```
ocamlc −c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc −c basic_parser.mli
ocamlc −c basic_lexer.ml
ocamlc −c basic_parser.ml
```

После чего получим файлы `basic_lexer.cmo` и `basic_parser.cmo`, которые можно использовать в нашей программе.

Теперь, у нас есть весь необходимый арсенал, для того чтобы переделать программу.

Удалим все типы и функции параграфов «лексический анализ» (стр. ??) и «синтаксический анализ» (стр. ??) для программы Бэйсик. Также в функции `one_command` (стр. ??) заменим выражение:

```
match parse (input_line stdin) with
```

на

```
match line lexer (Lexing.from_string ((input_line stdin)^"\n")) with
```

Заметьте, что необходимо поместить в конце линии символ конца '\n', который был удалён функцией `input_line`. Это необходимо, потому что данный символ используется для указания конца командной линии (`Leol`).

11.6 Exercises

11.7 Резюме

В данной главе были описаны различные средства лексического и синтаксического анализа Objective CAML. По порядку описания:

- модуль `Str` для фильтрования рациональных выражений
- модуль `Genlex` для быстрого создания простых лексических анализаторов
- `ocamllex` представитель семейства `lex`
- `ocamlyacc` представитель семейства `yacc`
- потоки, для построения нисходящих анализаторов, в том числе и контекстных

При помощи инструментов `ocamllex` и `ocamlyacc` мы переделали синтаксический анализ Бэйсика, который проще поддерживать, чем анализатор представленный на стр. ??.

11.8 To Learn More

Глава 12

Взаимодействие с языком C

Глава 13

Приложения

Часть III

Устройство программы

Глава 14

Модульное программирование

Глава 15

Объектно-ориентированное программирование

Глава 16

Сравнение моделей устройств программ

Глава 17

Приложения

Часть IV

Параллелизм и распределение

Глава 18

Процессы и связь между процессами

Глава 19

Параллельное программирование

Глава 20

Распределённое программирование

Глава 21

Приложения

Часть V

Разработка программ с помощью Objective CAML

Часть VI

Приложения