1 Интерпретатор Moscow ML

Запуск интрепретатора осуществляется командой `mosml`.

```
C:\>mosml
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
-
```

Символ `-` в начале последней выведенной строки является приглашением интерпретатора для ввода команды. Каждая команда интерпретатора представляет выражение (команду) Standard ML (SML) после которой следует символ `;`. После ввода команды ML-система её анализирует, компилирует, выполняет и выводит результат выполнения на терминал. Выданный результат предваряется символом `>`.

Образец диалога:

```
- 3+2;
> val it = 5 : int
```

Здесь пользователь вводит фразу `3+2`, что означает «вычислить значение выражения `3+2`». ML вычисляет выражение и выдает сообщение о результате: «значение заданного выражения `5`, его тип — целочисленный — `int`, и это значение связано с временным идентификатором `it`».

В процессе диалога могут возникать различные виды ошибок. В основном они распадаются на три категории: синтаксические ошибки, ошибки в согласовании типов и ошибки времени исполнения. Пример того, что происходит, когда введена синтаксически неправильная фраза:

```
- let x=3 in x end;
! Toplevel input:
! let x=3 in x end;
!     ^
! Lexical error: ill-formed token.
```

Ошибки времени исполнения (как, например, деление на 0) являются одним из видов *исключительных событий*. Вот пример того, что можно получить при возникновении ошибки времени исполнения:

```
- 3 div 0;
! Uncaught exception:
! Div
```

Ошибки в согласовании типов возникают при некорректном использовании значений, например, при попытке прибавить `3` к `true`:

```
- 3+true;
! Toplevel input:
! 3+true;
!   ^^^^
! Type clash: expression of type
!   bool
! cannot have type
!   int
```

Одной из ошибок, не распознаваемых ML-системой, является бесконечный цикл. Если вы подозреваете, что ваша программа зациклилась, вы можете прекратить ее выполнение, нажав Ctrl-C. ML выдаст сообщение `Interrupted.` и вернется на верхний уровень выполнения программы.

Исполнение файла с программой на языке ML осуществляется с помощью команды `use`

```
- use "C:\\temp\\streams.sml";
```

Загрузка скомпилированного модуля происходит по команде `load`.

```
- load "Math";
```

Выход из интерпретатора происходит после команды $quit()$

```
- quit();
```

2 Базовые типы данных

2.1 Тип `unit`

Тип `unit` состоит из единственного значения, записываемого как `()`. Этот тип используется в тех случаях, когда выражение не имеет какого-либо осмысленного значения, а также вместо аргумента функции в тех случаях, когда функция не должна иметь аргументов.

```
- ();
> val it = () : unit
```

2.2 Тип `bool`

Тип `bool` состоит из значений `true` и `false`. Для работы со значениями этого типа имеются одноместная операция `not` и две двухместных операции `andalso` и `orelse` (соответствующие обычным логическим операциям «не», «и» и «или»).

Первый аргумент условного выражения `if` e `then` e_1 `else` e_2 должен иметь тип `bool`. Это выражение является в некоторой степени аналогом выражения $e?e_1{:}e_2$ в языке C. Типы выражений e_1 и e_2 должны совпадать.

```
- not true;
> val it = false : bool
- false andalso true;
> val it = false : bool
- false orelse true;
> val it = true : bool
- if false then 1 else 2;
> val it = 2 : int
- true:bool;
> val it = true : bool
- if true then false else true;
> val it = false : bool
```

2.3 Тип `int`

Тип `int` является множеством (положительных и отрицательных) целых чисел. Целые числа записываются обычным способом, за исключением того, что для отрицательных чисел знак записывается тильдой `~` вместо традиционного минуса. Целочисленные выражения могут содержать обычные знаки арифметических операций `+`, `-`, `*`, `div` и `mod` (`div` и `mod` обозначают соответственно целочисленное деление и получение остатка от деления нацело).

```
- 75;
> val it = 75 : int
- ~24;
> val it = ~24 : int
- (3 + 2) div 2;
> val it = 2 : int
- (3 + 2) mod 2;
> val it = 1 : int
- 2 - 3;
> val it = ~1 : int
```

Имеются и знаки для операций сравнения <, <=, >, >=, = и <>. Все эти операции требуют двух аргументов типа `int` и возвращают результат типа `bool` (`true` или `false` в соответствии с тем, выполнено условие или нет).

```
- 3 < 2;
> val it = false : bool
- 3 * 2 > 12 div 6
;
> val it = true : bool
- if 4 * 5 mod 3 = 1 then 17 else 51;
> val it = 51 : int
```

2.4 Тип `string`

Тип `string` состоит из всех конечных последовательностей литер. Строки записываются обычным способом, как последовательность литер между двойными кавычками. Если двойная кавычка должна войти в состав строки, она записывается как "\"". Если символ \ должен войти в состав строки, то он удваивается "\\".

```
- "Fish knuckles";
> val it = "Fish knuckles" : string
- "V";
> val it = "V" : string
```

Функция `size` возвращает длину строки в литерах. Функция `^` является инфиксной двухместной операцией конкатенации строк.

```
- "Rhinocerous " ^ "Party";
> val it = "Rhinocerous Party" : string
- size "Walrus whistle";
> val it = 14 : int
```

2.5 Тип `real`

Тип чисел с плавающей точкой называется `real`. Действительные числа записываются более-менее обычным для языков программирования способом: целое число, за которым следует десятичная точка, за которой следует последовательность из одной или более десятичных цифр, за которыми следует знак экспоненты, `E`, за которым следует другое число. Экспонента может быть опущена, если присутствует десятичная точка; десятичная точка может быть опущена, если присутствует экспонента. Все это нужно для того, чтобы отличать целые числа от действительных.

```
- 3.14159;
> val it = 3.14159 : real
- 3E2;
> val it = 300.0 : real
- 3.14159E~2;
> val it = 0.0314159 : real
```

Для действительных чисел имеются обычные арифметические операции `-`, `+`, `-`, `*`, `/` и операции сравнения `<`, `<=`, `>`, `>=`, `=`, `<>`. Функция `real` преобразует целый аргумент в соответствующее действительное число, а функция `round` округляет действительный аргумент в целое число. Модуль `Math` языка предоставляет функции `sin`, `sqrt`, `exp` и т. д., соответствующие обычным математическим функциям.

```
- 3.0 + 2.0;
> val it = 5.0 : real
- (3.0 + 2.0) = real(3 + 2);
> val it = true : bool
- round(~3.2);
> val it = ~3 : int
- Math.cos 0.0;
> val it = 1.0 : real
- Math.cos 0;
! Toplevel input:
! Math.cos 0;
!          ^
! Type clash: expression of type
!   int
! cannot have type
!   real
```

2.6 Упорядоченные n-ки

Тип $\alpha*\beta$, где α и β — произвольные типы, является типом упорядоченной пары, где первый член имеет тип α а второй — тип β. Упорядоченная пара записывается как (e_1,e_2), где e_1 и e_2 — выражения. Аналогично образуются упорядоченные n-ки. А именно, последовательность выражений, отделенных друг от друга запятыми, заключенная в круглые скобки. Если выражения e_1, e_2, ..., e_n имеют типы α_1, α_2, ..., α_n соответственно, то типом упорядоченной n-ки $(e_1, e_2, \ldots, e_n)$ будет $\alpha_1*\alpha_2*\cdots*\alpha_n$.

```
- (true, ());
> val it = (true, ()) : bool * unit
- (1, false, 17, "Blubbet");
> val it = (1, false, 17, "Blubbet") : int * bool * int * string
- (if 3 = 5 then "Yes" else "No", 14 mod 3);
> val it = ("No", 2) : string * int
```

Равенство между n-ками — покомпонентное: две n-ки равны, если их соответствующие компоненты равны. Если две n-ки имеют различные типы, то при попытке их сравнения возникает ошибка в согласовании типов.

```
- (14 mod 3, not false ) = (1 + 1, true);
> val it = true : bool
- ("abc",  (5 * 4) div 2) = ("a" ^ "bc", 11);
> val it = false : bool
```

2.7 Списки

Тип α `list` состоит из конечных последовательностей (списков) значений типа α. Так значение типа `int list` является списком целых чисел, а значение типа `bool list` является списком логических значений. Имеется два способа записи списков, основной и сокращенный.

Первый основывается на следующем определении списка: список значений типа α либо пуст, либо представляет из себя значение типа α, за которым следует список значений типа α. В соответствии с этим определением пустой список записывается как `nil`, а непустой список записывается как $e :: l$, где e — выражение типа α, а l — выражение типа α list. Название операции :: произносится как «конс». Тогда список из n элементов можно представить как

$$e_1 \texttt{ :: } e_2 \texttt{ :: } \cdots \texttt{ :: } e_n \texttt{ :: nil}.$$

В сокращенной записи список представляется в виде последовательности разделенных запятыми элементов, заключенной в квадратные скобки.

```
- 3 :: 4 :: nil;
> val it = [3, 4] : int list
- (3 :: nil) :: (4 :: 5 :: nil) :: nil;
> val it = [[3], [4, 5]] : int list list
- ["This", "is", "it"];
> val it = ["This", "is", "it"] : string list
- nil;
> val 'a it = [] : 'a list
- [];
> val 'a it = [] : 'a list
- [3 + 5, 7, 8 div 3];
> val it = [8, 7, 2] : int list
```

Тип списка `nil` — особый: он включает *переменную типа* (выводится как `'a` и произносится как «альфа»). Причиной этого является то, что в пустом списке нет никакой информации о том, списком элементов какого типа он является. Константа `nil` может рассматриваться как `int list`, как `bool list` или как `int list list` в зависимости от контекста, в котором она находится. Это выражается тем, что константе `nil` приписывается тип `'a list`, где `'a` является переменной, пробегающей множество типов. Частные случаи типа, включающего переменные типа (такой тип мы будем также называть *полиморфным типом*, или, короче, *политипом*), получаются путем замены всех вхождений данной пе-

ременной типа на какой-либо тип (при этом *все* вхождения данной переменной заменяются на *один и тот же* тип); подставляемый вместо переменной тип сам может быть полиморфным. Например, типы `int list` и `(int * 'b) list` являются частными случаями политипа `'a list`. Типы, которые не содержат переменных типа, называются *мономорфными типами* (или, короче, *монотипами*).

Равенство для списков является поэлементным: два списка равны, если они состоят из одного и того же числа элементов и соответствующие элементы равны между собой. Как и в случае упорядоченных n-ок, невозможно сравнение двух списков с элементами разных типов, и попытка выполнить такую операцию приведет к ошибке в согласовании типов.

```
- [1, 2, 3] = 1 :: 2 :: 3 :: nil;
> val it = true : bool
- [[1],  [2, 4] ] = [[2 div 2], [1 + 1, 9 div 3]];
> val it = false : bool
```

2.8 Записи

Записи в ML аналогичны записям в языке Pascal, структурам в C и т. д. Запись состоит из конечного множества помеченных полей, каждое из которых является значением некоторого типа; как и в случае упорядоченных n-ок, различные поля могут иметь различные типы. Запись задается последовательностью равенств в форме n_i=e_i (где n_i есть метка и e_i — выражение), разделенных запятыми, заключенной в фигурные скобки. Каждое равенство n_i=e_i устанавливает значение поля, помеченного меткой n_i равным значению выражения e_i.

Типом записи является последовательность пар вида $n_i : \alpha_i$ (где n_i есть метка и α_i — тип), разделенных запятыми и заключенных в фигурные скобки. Порядок равенств, задающих запись, не имеет значения: компоненты записи определяются своими метками, а не положением в записи. Равенство для записей — покомпонентное: две записи равны, если они имеют одно и то же множество меток полей, и поля, помеченные одинаковыми метками, имеют равные значения.

```
- {name = "Foo", used = true};
> val it = {name = "Foo", used = true} : {name : string, used : bool}
- {name = "Foo", used = true} = {used = not false, name = "Foo"};
> val it = true : bool
- {name = "Bar", used = true} = {name = "Foo", used = true};
> val it = false : bool
```

Упорядоченные n-ки являются частным случаем записей: упорядоченная n-ка типа $\alpha_1 * \alpha_2 * \cdots * \alpha_n$ является сокращенным обозначением для записи типа `{1 :` α_1`, 2 :` α_2`, ..., `n `:` α_n`}`.

Например, выражения `(3, 4)` и `{1 = 3, 2 = 4}` обозначают в точности одно и то же.

3 Идентификаторы, привязки и объявления

Переменные (идентификаторы значений) вводятся путем *привязки* идентификатора к значению. Для этого используется специальная конструкция

$$\texttt{val}\ v\ \texttt{=}\ e$$

где v — идентификатор, e — выражение.

```
- val x = 4 * 5;
> val x = 20 : int
- val s = "Abc" ^ "def";
> val s = "Abcdef" : string
- val pair = (x, s);
> val pair = (20, "Abcdef") : int * string
```

С помощью ключевого слова `and`, используемого в качестве разделителя, можно привязать к значениям сразу несколько идентификаторов:

```
- val x = 17;
> val x = 17 : int
- val x = true and y = x;
> val x = true : bool
  val y = 17 : int
```

Несколько привязок, соединенных словом `and`, вычисляются параллельно — сначала вычисляются их правые части, а затем идентификаторы, указанные в левых частях, привязываются к полученным значениям.

Конструкция `local` организует среду со своей собственной привязкой значений к идентификаторам для облегчения построения других объявлений. Общий формат конструкции следующий:

$$\texttt{local}\ E_l\ \texttt{in}\ E_g\ \texttt{end}$$

где E_l — последовательность локальных объявлений, E_g — последовательность объявлений. Объявленные в E_l идентификаторы являются локальными при вычислении значений выражений в объявлениях E_g. По окончании выполнения конструкции `local` доступными остаются лишь значения идентификаторов, объявленных в E_g.

```
- val x = 17;
> val x = 17 : int
- val y = 15;
> val y = 15 : int
- local
    val x = 5;
    val z = 3;
  in
    val y = x * x - z;
    val v = y + x;
  end;
```

```
> val y = 22 : int
  val v = 27 : int
- x;
> val it = 17 : int
- y;
> val it = 22 : int
- z;
! Toplevel input:
! z;
! ^
! Unbound value identifier: z
- v;
> val it = 27 : int
```

Выражение `let` позволяет локализовать объявление, используемое при вычисления выражения. Формат выражения `let` следующий:

$$\texttt{let } E_l \texttt{ in } e \texttt{ end}$$

Так же как и в конструкции `local`, E_l представляет собой последовательность объявлений; e — некоторое выражение, для которого объявления в E_l являются локальными. Значением выражения `let` является вычисленное значение выражения e.

```
- val y =
    let
      val x = 10
    in
      x * x + 2 * x + 1
    end;
> val y = 121 : int
```

4 Образцы

Выделение частей составных значений выполняется с помощью *сопоставления с образцом*. Значения составных типов сами являются составными; они строятся из составляющих их значений с помощью конструкторов. Естественно использовать аналогичную конструкцию для разложения их на составляющие значения.

Предположим, что `x` имеет тип `int * bool`. Тогда `x` является парой, левая компонента которой является целым, а правая — логическим значением. Можем получить значения левой и правой компонент, используя следующую обобщенную форму привязки к значению:

```
- val x = (17, true);
> val x = (17, true) : int * bool
- val (left, right) = x;
> val left = 17 : int
  val right = true : bool
- left;
> val it = 17 : int
```

Левая часть второй привязки является *образцом*. В общем случае образец строится из переменных и констант с помощью конструкторов. Таким образом, образец есть выражение, возможно, включающее переменные. Отличие от ранее рассматривавшихся выражений состоит в том, что в образце переменные не изображают значения определенные предшествующими привязками, а являются переменными, которые должны быть привязаны к значениям в процессе сопоставления с образцом. В приведенном выше примере `left` и `right` суть новые идентификаторы, которые должны быть привязаны к значениям. Сопоставление с образцом состоит в параллельном разложении на составные части значения `x` и образца, и сопоставлении компонент значения с соответствующими им частями образца. Переменная может быть сопоставлена с любым значением, и тогда идентификатор привязывается к этому значению. Если же образец содержит константу, то она должна совпадать с соответствующей частью значения, с которым выполняется сопоставление. В приведенном выше примере, поскольку `x` является упорядоченной парой, сопоставление завершается успешно; при этом левая компонента `x` привязывается к `left`, а правая — к `right`.

Любая попытка сопоставления образцов, имеющих разную внутреннюю структуру, рассматривается как ошибка в согласовании типов, и обнаруживается статически, т. е. в период компиляции. Сопоставление с образцом может потерпеть неудачу и динамически, т.е. во время исполнения программы:

```
- val x = (false, 17);
> val x = (false, 17) : bool * int
- val (false, w) = x;
> val w = 17 : int
- val (true, w) = x;
! Uncaught exception:
! Bind
```

Сопоставление с образцом может быть выполнено для значений имеющих любой из введенных ранее типов. Например:

```
- val lst = ["Lo", "and", "behold"];
> val lst = ["Lo", "and", "behold"] : string list
- val [x1, x2, x3] = lst;
> val x1 = "Lo" : string
  val x2 = "and" : string
  val x3 = "behold" : string
- val hd :: tl = lst;
> val hd = "Lo" : string
  val tl = ["and", "behold"] : string list
```

Чтобы избавиться от необходимости указывать имена, которые в дальнейшем все равно не будут использоваться, ML позволяет писать вместо них универсальный образец — «джокер» (обозначаемый знаком _ подчеркивание), который может быть сопоставлен с любым значением без привязки к нему идентификатора.

```
- val _ :: tl = lst;
> val tl = ["and", "behold"] : string list
- val (_, x) = (5, 6.7);
> val x = 6.7 : real
```

Сопоставление с образцом может быть применено и для записей, и делается это с помощью помеченных полей. Следующий пример иллюстрирует сопоставление с образцом для записей:

```
- val r = {name = "Foo", used = true};
> val r = {name = "Foo", used = true} :
                      {name : string, used : bool}
- val {used = u, name = n} = r;
> val u = true : bool
  val n = "Foo" : string
```

Частичное сопоставление записи с образцом может быть выполнено с помощью *универсального образца для записей*, как в следующем примере:

```
- val {used = u, ...} = r;
> val u = true : bool
```

Поскольку выделение одного поля из записи является широко распространенной операцией, для нее предусмотрено специальное обозначение: поле `name` записи `r` может быть обозначено как `#name r`. Поскольку упорядоченные n-ки являются частным случаем записи (именами полей у них являются целые числа от 1 до n), `i`-тая компонента упорядоченной n-ки может быть выделена с помощью `#i`.

Образцы могут вкладываться друг в друга, как в приводимом ниже примере:

```
- val x = (("foo", true), 17);
> val x = (("foo", true), 17) : (string * bool) * int
- val ((ll, lr), r) = x;
> val ll = "foo" : string
  val lr = true : bool
  val r = 17 : int
```

Иногда бывает удобно ввести «промежуточные» переменные в образце. Например, нам может понадобиться привязать к паре `(ll, rr)` идентификатор `l`. Это выполняется с помощью *многоуровневых образцов*. Многоуровневый образец получается путем приписывания образца к переменной внутри другого образца, как в следующем примере:

```
- val (l as (ll, lr), r) = x;
> val l = ("foo", true) : string * bool
  val ll = "foo" : string
  val lr = true : bool
  val r = 17 : int
```

Любая переменная может входить в образец только один раз. В частности, нельзя задать образец вроде `(x, x)` — который должен был бы быть сопоставим только с симметричными парами.

Конструкция `case` производит сопоставление с образцом и, в зависимости от результата сопоставления, выдает соответствующий результат.

```
- val m = 5 and d = 9;
> val m = 5 : int
  val d = 9 : int
- case (m, d)
    of (1, 1)  => "Новый год"
     | (5, 9)  => "День победы"
     | (3, 8)  => "Женский день"
     | (11, 4) => "День народного единства"
     | _       => "Не праздник";
> val it = "День победы" : string
> val m = 7 : int
  val d = 12 : int
- case (m, d)
    of (1, 1)  => "Новый год"
     | (5, 9)  => "День победы"
     | (3, 8)  => "Женский день"
     | (11, 4) => "День народного единства"
     | _       => "Не праздник";
> val it = "Не праздник" : string
```

5 Определение функций

Рассмотрим *привязки к функциональным значениям*, посредством которых в ML определяются новые функции.

Функции используются путем *применения* их к аргументам (мы будем также использовать термин *аппликация)*. Синтаксически это записывается как два выражения одно за другим (значением первого выражения должна являться функция, а значением второго — ее аргумент) — как, например, `size "abc"` для вызова функции `size` с аргументом `"abc"`. Обычно, при необходимости использовать функции (содержательно) нескольких аргументов, аргументов функции «упаковываются» в один — упорядоченную n-ку. Так, например, если функция `append` должна получать два аргумента-списка и возвращать результат-список, применение ее будет иметь вид `append (l1, l2)`: формально функция применяется к одному аргументу, который является упорядоченной парой `(l1, l2)`.

Для некоторых функций от двух аргументов используется специальный синтаксис — так называемая *инфиксная запись*, в которой знак функции записывается между двумя ее аргументами. Например, запись e_1 `+` e_2 в действительности означает «применить функцию `+` к упорядоченной паре `(`e_1`,` e_2`)`.

Аппликация в ML может иметь более сложную форму, чем в других языках программирования. Причиной этого является следующее: в большинстве языков программирования функция может обозначаться только идентификатором, и поэтому вызов функции всегда имеет вид $f(e_1, \ldots, e_n)$, где f — идентификатор. В ML

нет такого ограничения: функция является обычным значением, и может быть получена в результате вычисления выражения. Поэтому в общем случае аппликация в ML имеет вид $e\ e'$. Вычисление такого выражения выполняется следующим образом. Сначала вычисляется значение выражения e, в результате чего получается некоторая функция f затем вычисляется выражение e', в результате чего получается некоторое значение v; после этого функция f применяется к значению v. В простейшем случае, когда выражение e является идентификатором (как, например, `size`), вычисление e является крайне простой операцией: нужно просто взять значение, к которому привязан идентификатор (оно должно быть функцией). Но в общем случае процесс вычисления e может быть сколь угодно сложным. Правила вычисления аппликации предполагают передачу аргумента *по значению*, поскольку аргумент вычисляется до применения функции.

Функции являются значениями, а каждое значение в ML имеет тип. *Функциональные типы* являются составными типами, членами которых являются функции. Функциональные типы записываются как α `->` β, где α и β — два типа.

Выражение такого типа имеет в качестве значения функцию, которая может быть применена к аргументу типа α (и только к аргументу такого типа), и, если ее вычисление завершается успешно, возвращает результат типа β. Тип α называется типом области определения функции, а тип β — типом области значений. Аппликация $e\ e'$ допустима тогда и только тогда, когда тип e есть α `->` β, тип e' ’— α; тип всего выражения есть β.

```
- size;
> val it = fn : string -> int
- not;
> val it = fn : bool -> bool
- val x = 17.0;
> val x = 17.0 : real
- (if x > 0.0 then Math.cos else Math.sin) x;
> val it = ~0.275163338052 : real
```

Поскольку функции являются значениями, мы можем привязать идентификатор к функции, используя обычный механизм привязки идентификатора к значению. Например:

```
- val len = size;
> val len = fn : string -> int
- len "abc";
> val it = 3 : int
```

Определение функции производится с помощью конструкции `fun`. Несколько примеров определений функций:

```
- fun fact x = if x = 0 then 1 else x * fact (x - 1);
> val fact = fn : int -> int
- fact 5;
> val it = 120 : int
- fun plus (x, y) = x + y;
> val plus = fn : int * int -> int
- plus (4, 5);
> val it = 9 : int
- fun rplus (x, y) : real = x + y;
> val rplus = fn : real * real -> real
- rplus (3.5 , 4.0);
> val it = 7.5 : real
- fun rmult (x : real, y) = x * y;
> val rmult = fn : real * real -> real
- rmult (3.5, 0.75);
> val it = 2.625 : real
```

Конструкция `fun` производит *привязку идентификатора к функциональному значению*. За идентификатором — именем функции следуют ее параметры, которые задаются образцом. Когда выполняется применение определенной пользователем функции, значение аргумента сопоставляется с параметром-образцом — точно так же, как и при привязке к значению; результатом этого сопоставления является некоторая среда, в которой и выполняется вычисление тела функции.

При определении функции может быть определено несколько образцов аргументов, каждому из которых сопоставляется свое тело функции. При определении функции эти сопоставления отделяются друг от друга вертикальной чертой `|`. Например:

```
- fun is_nil (nil)  = true
  |   is_nil (_::_) = false;
> val 'a is_nil = fn : 'a list -> bool
- val lst1 = nil;
> val 'a lst1 = [] : 'a list
- val lst2 = [2, 3];
> val lst2 = [2, 3] : int list
- is_nil lst1;
> val it = true : bool
- is_nil lst2;
> val it = false : bool
```

В общем случае, если тип аргумента определяемой функции имеет более одного конструктора, то определение функции должно содержать по одному предложению на каждый конструктор. Это гарантирует то, что функция может принять любой аргумент данного типа. Такой способ определения функций называется *определением с помощью разбора случаев*, поскольку определение содержит по одному предложению для каждого случая формы аргумента.

Например при помощи определения с помощью разбора случаев можно переопределить функцию факториал:

```
- fun fact 0 = 1
  |   fact n = n * fact (n - 1);
> val fact = fn : int -> int
- fact 5;
> val it = 120 : int
```

Конструкцию `fun` можно применять для определения функций, возвращающих другие функции. Для этого вместо одного аргумента указывается последовательность аргументов. Например:

```
- fun times x y = x * y;
> val times = fn : int -> int -> int
- times 2 3;
> val it = 6 : int
- val twice = times 2;
> val twice = fn : int -> int
- twice 4;
> val it = 8 : int
```

Если вместо одного аргумента при определении функции указывается последовательность n аргументов, то такую функцию называют каррированной функцией от n аргументов. В приведенном примере `times` — каррированная функция двух аргументов.

Функциональное значение можно получить без установки его привязки к идентификатору. Для этого используется выражение `fn`. Общий вид выражения `fn` следующий:

$$\texttt{fn}\ a\ \texttt{=>}\ e$$

где a — образец аргумента, e — выражение, сопоставляемое с образцом. В выражении `fn` возможно присутствие последовательности пар a `=>` e, разделенных вертикальной чертой `|`.

```
- fun listify x = [x];
> val 'a listify = fn : 'a -> 'a list
- val listify2 = fn x => [x];
> val 'a listify2 = fn : 'a -> 'a list
- listify 7;
> val it = [7] : int list
- listify2 7;
> val it = [7] : int list
- map (fn x => [x]) [1, 2, 3];
> val it = [[1], [2], [3]] : int list list
```

```
- val lst = [(1, 2), (3, 4), (1, 4), (2, 8)];
> val lst = [(1, 2), (3, 4), (1, 4), (2, 8)] : (int * int) list
- map (fn (x, 4) => x * x
        | (x, y) => y - x) lst;
> val it = [1, 9, 1, 6] : int list
```