

Primitivni tipi

```
()          unit
true, false  bool
0, 42, ~123  int
3.14         real
#"a"         char
"Hello World" string
poljuben tip 'a
primerjalni tip ''a
```

Operatorji

```
+      num * num -> num
-      num * num -> num
*      real * real -> real
div    int * int -> int
/      real * real -> real
mod    int * int -> int
~      num -> num
not    bool -> bool
andalso bool * bool -> bool
orelse  bool * bool -> bool
<      ''a * ''a -> bool
>      ''a * ''a -> bool
<=     ''a * ''a -> bool
>=     ''a * ''a -> bool
<>     ''a * ''a -> bool
=      ''a * ''a -> bool
:=     'a ref * 'a -> unit
!      'a ref -> 'a
^      string * string -> string
@      'a list * 'a list -> 'a list
```

num predstavlja tip int ali real.

Terke

```
(v1, ..., vn)      'a1 * ... * 'an
(#"p", ())          char * unit
(123, "abc", false) int * string * bool
{1=123, 2="abc", 3=false} int * string * bool
```

Zapisi

```
{k1=v1, ..., kn=vn}  {k1: 'a1, ..., kn: 'an}
{ime="Jan", visina: 150} {ime: string, visina: int}
```

Do vrednosti v zapisu dostopamo z #kljuc zapis.

Seznami

```
[]          'a list
hd::tl      'a list
[v1, v2, ..., vn] 'a list
[1,2,3]     int list
1::2::3::[] int list
```

Funkcije za delo s seznami:

```
null      fn : 'a list -> bool
length    fn : 'a list -> int
op @      fn : 'a list * 'a list -> 'a list
hd        fn : 'a list -> 'a
tl        fn : 'a list -> 'a list
List.last fn : 'a list -> 'a
List.nth  fn : 'a list * int -> 'a
List.take fn : 'a list * int -> 'a list
List.drop fn : 'a list * int -> 'a list
rev       fn : 'a list -> 'a list
map       fn : ('a -> 'b) -> 'a list -> 'b list
List.filter fn : ('a -> bool) -> 'a list -> 'a list
foldl     fn : ('a*'b -> 'b) -> 'b -> 'a list -> 'b
foldr     fn : ('a*'b -> 'b) -> 'b -> 'a list -> 'b
```

Razlika med foldl in foldr:

```
fun f (x, acc) = ... ;
foldl f 0 [1, 2, 3] = f(3,f(2,f(1,0)))
foldr f 0 [1, 2, 3] = f(1,f(2,f(3,0)))
```

Funkcije

```
fun ime args = ...      fn: 'a -> 'b
fun add (a, b) = a + b  fn: int * int -> int
fun add a b = a + b     fn: int -> int -> int
fn (a, b) => a+b         fn: int * int -> int
fun id x = x            fn: 'a -> 'a
fn x => x               fn: 'a -> 'a
```

Z ključno besedo fun lahko definiramo funkcije po vzorcu:

```
fun f vzorec1 = izraz1
  | f vzorec2 = izraz2
  | ...
  | f vzorecn = izrazn
```

Opcije

```
NONE      'a option
SOME v     'a option
SOME 13    int option
```

Unije tipov

```
datatype ime_tipa = CONS1 of 'a1
                  | ...
                  | CONSn of 'an
datatype int_or_bool = INT of int
                  | BOOL of bool
datatype 'a option = NONE | SOME of 'a
datatype 'a list = [] | :: of 'a * 'a list
datatype ('a, 'b) generic = A of 'a
                          | B of 'b
                          | AB of 'a * 'b
                          | NONE
```

```
datatype oseba =
  OSEBA of {ime: string, priimek: string}
```

Sinonimi za tipe

```
type ime_sinonima = tip
type oseba = {ime: string, priimek: string}
```

SML-NJ

```
type 'a seznam = 'a list
```

Ujemanje vzorcev

```
case izraz of
  vzorec1 => izraz1
  | vzorec2 => izraz2
  | ...
  | vzorecn => izrazn
```

Vzorec je lahko poljuben konstruktor ali primitivna vrednost. Vzorce lahko gnezdimo. V vzrocu lahko uporabimo tudi spre-menljivke, ki jih potem uporabimo v izrazu.

V vzorcu lahko uporabljamo le konstruktorje ali vrednosti istega tipa.

```
case sez of
  [] => 0
  | x::xs => xs @ [x]
```

```
case opt of
  NONE => 0
  | SOME x => x
```

Vzajemna rekurzija

Pri definiciji funkciji in tipov lahko uporabimo ključno besedo and, da definiramo več funkcij ali tipov hkrati.

```
fun liho n =
  if n = 0 then false else sodo (n-1)
and sodo n =
  if n = 0 then true else liho (n-1)
```

```
datatype a = A of b | Aend
and       b = B of a | Bend
```

Lokalno okolje

```
val x = 10;
let
  val y1 = x + 1 (* y1 = 11 *)
  val x = 1      (* zasencimo globalno x *)
  val y2 = x+1   (* y = 2 *)
  val a = 100
  fun f a = a + y2 (* 'a' vzamemo iz args *)
in
  f y1 (* vrne vrednost 11 + 2 = 13  *)
end
```

Mutacije

```
val x = ref 10; (* x = ref 10 : int ref *)
x := 20; (* x = ref 20 : int ref *)
!x; (* vrne 20 : int *)
```

Izjeme

Svoj tip izjeme definiramo z exception:

```
exception Izjema of string;
```

Izjemo sprožimo z raise.

```
raise (Izjema "Napaka");
```

Izjeme ujamemo z handle:

```
izraz_ki_prozi_izjemo
handle
  vzorec1 => izraz1
  | vzorec2 => izraz2
  | ...
  | vzorecn => izrazn
```

Izjeme so tipa exn.

Moduli

```
structure ImeModula = struct
  (* definicije val, fun, datatype, ... *)
end
```

Do vrednosti v modulu dostopamo z ImeModula.ime. Naprimer:

```
structure MojModul = struct
  val x = 10
  fun pozdravi () = "Zivjo" ^ Int.toString x
end;
```

```
MojModul.x; (* vrne 10 *)
MojModul.pozdravi(); (* vrne "Zivjo10" *)
```

Lahko dodamo podpis modula, ki določa katere vrednosti so vidne izven modula.

```
signature MojModulP = sig
  val pozdravi : unit -> string
end
```

```
structure MojModul :> MojModulP = struct
  val x = 10
  fun add (a, b) = a + b
  fun pozdravi () = "Zivjo" ^ Int.toString x
end
```

```
MojModul.x; (* napaka *)
MojModul.add(1,2); (* napaka *)
MojModul.pozdravi(); (* vrne "Zivjo10" *)
```

Prednosti leksikalnega dosega

- Imena spremenljivk v funkciji so neodvisna od imen zu-nanjih spremenljivk
- Funkcija je neodvisna od imen uporabljenih spremenljivk
- Tip funkcije lahko določimo ob njeni deklaraciji
- Ovojnica shrani podatke, ki jih potrebuje za kasnejšo izvedbo.

