SML-NJ

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
         bool -> bool
not
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

Zapisi

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

Sinonimi za tipe

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

```
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 spremenljivke, 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

structure MojModul = struct

Izjeme so tipa exn.

Moduli

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

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

```
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 zunanjih spremenljivk
- Funkcija je neodvisna od imen uporabljenih spremenljivk
- Tip funkcije lahko določimo ob njeni deklaraciji
- Ovojnica shrani podatke, ki jih potrebuje za kasnejšo izvedbo.

Racket

Definicije

Na začetku vsake datoteke moramo dodati: #lang racket

Definicija vrednosti:

(define ime vrednost)

Funkcije

Definicija funkcije:

```
(define (ime arg1 arg2 ... argn) izraz)
```

Definicija funkcije z poljubnim številom argumentov:

```
(define (ime . args) izraz)
(ime 1 2 3); args = '(1 2 3)
```

Definicija funkcije z poimenovanimi argumenti in privzetimi vrednostmi. Nepoimenovani argumenti s privzetimi vrednostmi morajo biti za ostalimi nepoimenovanimi argumenti.

```
(define (f #:ime_a a b #:ime_c c) izraz )
(f #ime_c 3 2 #ime_a 1) ; a = 1, b = 2, c = 3

(define (f #:ime_a [a 1] [b 2]) izraz )
(f) ; a = 1, b = 2
```

Anonimna funkcija:

```
(lambda (arg1 arg2 ... argn) izraz)
(lambda args izraz)
```

Nadzor toka

```
(if pogoj potem sicer)
```

Stavek cond vrne vrednost prvega izraza, pri katerem je resničen let* pogoj (ne vrača #f).

```
(cond [pogoj1 izraz1]
        [pogoj2 izraz2]
        ...
        [else izrazn])
```

Ujemanje vzorcev

```
(match izraz
      (vzorec1 izraz1)
      (vzorec2 izraz2)
      ...
      (else izrazn))
```

Pari in seznami

```
null: null?
Vrednost '()
Ustvari par
                          (cons a d) \rightarrow pair?
Vrni prvi el.
                          (car p) \rightarrow any/c
Vrni drugi el.
                          (cdr p) \rightarrow any/c
                          (pair? v) \rightarrow boolean?
Ali je v par
Ali je v enak '()
                          (null? v) \rightarrow boolean?
Ustvari seznam
                          (list v ...) \rightarrow list?
                          (list? v) \rightarrow boolean?
Ali je v seznam
Dolžina seznama
                          (length lst) \rightarrow integer?
Obrni seznam
                          (reverse lst) \rightarrow list?
Vrni i-ti el. seznama
                         (list-ref lst i) \rightarrow any/c
                         (map proc lst \dots+) \rightarrow list?
Uporabi proc na
vsakem el. v 1st
Zloži iz leve
                          (foldl proc init lst ...+) \rightarrow list?
Zloži iz desne
                          (foldr proc init lst ...+) \rightarrow list?
Vrni seznam brez el- (filter pred lst) \rightarrow list?
ementov za katere
pred vrne #f
```

Seznam je sestavljen iz parov, kjer je drugi element seznam ali '().

```
(list 1 2 3) \equiv (cons 1 (cons 2 (cons 3 '())))
```

Primer uporabe foldl in foldr:

```
(define (f x acc) ...)
(foldl f init (list 1 2 3))
   ; f(3, f(2, f(1, init)))
(foldr f init (list 1 2 3))
   ; f(1, f(2, f(3, init)))
```

Funkcije map, foldl in foldr lahko uporabljamo na več seznamih hkrati.

```
(map + (list 1 2 3) (list 4 5 6))
   ; vrne (list 5 7 9)
(foldl
      (lambda (x y acc) (+ acc (max x y)))
      0 (list 10 2 3) (list 4 20 5)); vrne 35
```

Lokalno okolje

let izrazi se evalvirajo v oklju
 pred izrazom let let* izrazi se evalvirajo po vrsti in okolje se posodablja (kot SML)

letrec izrazi se evalvirajo v okolju, ki že vesebuje vse deklaracije naštete v letrec (kot and v SML) define deluje enako kot letrec, le drugačna sintaksa

Primer uporabe let:

```
(define x 100)
(let ([x 10] [y x]) (cons x y))
; vrne (10 . 100)
```

Primer uporabe let*:

```
(define x 100)
(let* ([x 10] [y x]) (cons x y))
; vrne (10 . 10)
```

Primer uporabe letrec:

```
(letrec
	([x (lambda () y)] [y 123])
	(cons (x) y))
	; vrne (123 . 123)
```

Zakasnjena evalvacija

```
Zakasnjena evalvacija:
```

```
(thunk izraz) \equiv (lambda () izraz)
```

Zakasnitev in sprožitev z uporabo delay in force:

Funkcija delay ustvari objekt tipa promise, ki hrani izračunano (define-syntax ime-makra vrednost. (syntax-rules (kljucr

Tokovi

Tok je funkcija, ki vrača par, v katerem je privi element vrednost drugi pa tok.

Primer:

Funkcija, ki izpiše prvih n elementov toka:

Funkcija, ki izpsuje elemente toka dokler velja pogoj:

Memoizacija

Rezultate funkcije shranimo v tabelo in jih uporabimo, če je funkcija ponovno klicana z istimi argumenti.

Za tabelo lahko uporabimo hash ali seznam parov (kjuč, vrednost).

```
Ustvari hash tabelo (make-hash) → hash?
Ali je v hash tabela (hash? v) → boolean?
Vrni vrednost za (hash-ref h k) → any/c
ključ k
Nastavi vrednost za (hash-set! h k v) → void?
ključ k na v
Ali ima h ključ k
Izbriši ključ k iz h (hash-remove! h k) → void?
```

Makri

vzorec lahko vsebuje spremenljivke, ki se vežejo na vrednosti v izrazu in ključne besede, ki so naštete v syntax-rules.

Primer makra:

```
(define-syntax mojif
   (syntax-rules (then else)
      [(mojif p then et)
        (if p et '())]
   [(mojif p then et else ef)
        (if p et ef)]
   [(mojif p1 then e1t elif p2 e2t else ef)
        (if p1 e1t if p2 e2t ef)]
   ))
(mojif #t then 123 else 456); vrne 123
```

Lastni podatkovni tipi

```
(struct mojtip
     (polje1 polje2 ... poljen)
   #:transparent ; polja so vidna v REPL
    #:mutable ; polja lahko spreminjamo
)
```

Avtomatsko dobimo funkcije:

```
Konstruktor (mojtip v1 v2 ... vn) \rightarrow mojtip
Preverjanje tipa (mojtip? v) \rightarrow boolean?
Vrednost polja (mojtip-polje v) \rightarrow any/c
Nastavi polje* (set-mojtip-polje! v vrednost) \rightarrow void?
* če je #:mutable.
```