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
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}
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
```

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 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.

PYTHON

Anonimna funkcija

```
f = lambda x: x + 1
f = lambda x, y: x + y
```

Dekoratorji

```
def decor(f):
    def wrapper(*args, **kwargs):
        print(args, kwargs)
        f(*args, **kwargs)
        print('end')
    return wrapper

@decor
def say_hi(name, times=1):
    print(f"hi␣{name}!␣"*times)

say_hi('mom', times=3)

# ('mom',) {'times': 3}
# hi mom! hi mom! hi mom!
# end
```

Iteratorji

Z razredom:

```
class MojIterator:
    def __init__(self, args):
        # inicializacija

    def __iter__(self):
        return self

    def __next__(self):
        # vrne naslednji element
          ali pa raise StopIteration

for x in MojIterator(args):
    pass
```

Z generatorjem:

```
def moj_generator(args):
    # inicializacija
    while True:
        # yield naslednji element
        # ali pa return za konec

for x in moj_generator(args):
    pass
```

Z sestavljanjem iteratorjev:

```
moj_iterator = (x**2 for x in range(10))

for x in moj_iterator:
    pass
```

Povezava med funkcijskim in objektnim prog.

Funkcijsko	Objektno
Če dodamo novo funkcijo, moramo v njej pokriti vse konstruktorje za podatkovni tip.	Če dodamo novo funkcijo, jo moramo implementirati za vse podrazrede.
Če dodamo nov konstruktor (podtip), ga moramo v vseh funkcijah pokriti.	Če dodamo nov (pod)razred moramo v njem implementirati vse funkcije.
Združujemo po funkcijah	*Združujemo po razredih (tipih)*

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)
```

Uporabne funkcije

Aritmetika

`(+ a1 ... an)`	$a_1 + \cdots + a_n$
`(- a1 ... an)`	$a_1 - \cdots - a_n$
`(* a1 ... an)`	$a_1 * \cdots * a_n$
`(/ a1 ... an)`	$a_1 * \cdots * a_n$
`(remainder a b)`	ostanek pri deljenju a/b
`(modulo a b)`	$a \mod b$
`(abs x)`	$\lvert x \rvert$
`(max a1 ... an)`	$\max(a_1, \ldots, a_n)$
`(min a1 ... an)`	$\min(a_1, \ldots, a_n)$
`(gcd a b)`	največji skupni delitelj
`(lcm a b)`	najmanjši skupni večkratnik
`(round x)`	zaokroži
`(floor x)`	zaokroži navzdol
`(ceiling x)`	zaokroži navzgor
`(truncate x)`	odreži decimalni del
`(sqrt x)`	$\sqrt{x}$
`(expt x y)`	x^y
`sin ... atan`	trigonometrične funkcije

Primerjava:

`(eq? a1 ... an)`	primerjava po referenci
`(= a1 ... an)`	$a_1 = \cdots = a_n$
`(> a1 ... an)`	$a_1 > \cdots > a_n$
`(>= a1 ... an)`	$a_1 \geq \cdots \geq a_n$
`(< a1 ... an)`	$a_1 < \cdots < a_n$
`(<= a1 ... an)`	$a_1 \leq \cdots \leq a_n$

Logične operacije:

`(not a)`	$\neg a$
`(and a1 ... an)`	$a_1 \wedge \cdots \wedge a_n$
`(or a1 ... an)`	$a_1 \vee \cdots \vee a_n$
`(xor a1 ... an)`	$a_1 \oplus \cdots \oplus a_n$

Nizi:

`(string? a)`	ali je a niz
`(string-length a)`	dolžina niza
`(string-ref a i)`	i-ti znak niza
`(string-append a1 ... an)`	združi nize

Nadzor toka

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

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

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

Ujemanje vzorcev

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

Pari in seznami

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

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

```
(list 1 2 3) ≡ (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) ≡ (lambda () izraz)
```

Zakasnitev in sprožitev z uporabo `delay` in `force`:

```
(define x
    (delay (begin
        (print "dolg izracun")
        "rezultat")))
(force x)
    ; izpise "dolg izracun" in vrne "rezultat"
(force x)
    ; vrne (shranjen) "rezultat"
```

Funkcija `delay` ustvari objekt tipa `promise`, ki hrani izračunano vrednost.

Tokovi

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

```
(define enke (cons 1 (lambda () enke)))
(define naravna
    (letrec (
        [f (lambda (x)
            (cons x (thunk (f (+ 1 x)))))])
        (f 1)))
```

Funkcija, ki izpiše prvih `n` elementov toka:

```
(define (izpisi n tok)
    (if (> n 1)
        (begin
          (displayln (car tok))
          (izpisi (- n 1) ((cdr tok))))
        (displayln (car tok))))
```

Funkcija, ki izpsuje elemente toka dokler velja `pogoj`:

```
(define (izppog tok pogoj)
    (cond [(pogoj (car tok)) (begin
            (displayln (car tok))
            (izppog ((cdr tok)) pogoj))]
          [#t #t]))
```

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 (ključ, vrednost).

```
(define f (letrec
    ([memo (make-hash)])
    (lambda (x)
        (if (hash-has-key? memo x)
            (hash-ref memo x)
            (let ([rezultat ...])
                (hash-set! memo x rezultat)
                rezultat
            )))))
```

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

Makri

```
(define-syntax ime-makra
    (syntax-rules (kljucna-beseda1 ...)
        [(ime-makra vzorec) izraz]
        ...
    )
)
```

`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)` → `mojtip`
Preverjanje tipa	`(mojtip? v)` → `boolean?`
Vrednost polja	`(mojtip-polje v)` → `any/c`
Nastavi polje*	`(set-mojtip-polje! v vrednost)` → `void?`

** če je* `#:mutable`.

Preverjanje

statično	preverjanje pred izvajanjem programa (ne preveja semantičnih napak, izjem, pravilnosti artimetičnih operatij (deljenje z 0), ...)
dinamično	preverjanje med izvajanjem programa

Trdnost in polnost sistema tipov

pozitiven	program z napako
negativen	program brez napake
trden (sound)	sistem nikoli ne sprejme pozitivnega programa (ima pa lažno pozitivne primere)
poln (comoplete)	sistem nikoli ne zavrne negativnega programa (ima pa lažno negativne primere)

Šibki in močni sistemi tipov

šibki sistem tipov ne preverja oziroma jih pogosto zanemarja (C, JavaScript)
močni sistem tipe preverja bodisi statično ali dinamično (SML, Racket)