Pregled konceptov programskih jezikov

I.Savnik, FAMNIT, 2016/17

Viri

J.C. Mitchell, Concepts in programming languages, Cambridge University

Press, 2003.

I.Savnik, Koncepti programskih jezikov, Zapiski, 2015.

Koncepti programskih jezikov

- Osnovne strukture
- Funkcijski jeziki
- Imperativni jeziki
- Objektno-usmerjeni jeziki
- Modularni jeziki

Osnovne strukture

- Vrednosti
  - Cela števila
  - Realna števila
  - Znaki
  - Nizi
  - Boolove vrednosti
- Seznami
- N-terice

Vrednosti

- Cela števila:

+	seštevanje
-	odštevanje in unarna negacija
*	množenje
/	celoštevilsko deljenje
mod	ostanek celoštevilskega deljenja

```
# 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
```

Vrednosti

- Realna števila:

+.	addition
-.	subtraction and unary negation
*.	multiplication
/.	division
**	exponentiation

```
# 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
```

ceil	
floor	
sqrt	square root
exp	exponential
log	natural log
log10	log base 10

cos	cosine
sin	sine
tan	tangent
acos	arccosine
asin	arcsine
atan	arctangent

Znaki in nizi

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "is a string";;
- : string = "is a string"
# (string_of_int 1987) ^ "is the year Caml was created";;
- : string = "1987 is the year Caml was created"
```

Boolove vrednosti

not	negacija
&&	zaporedni logični in
&	sinonim za &&
or	sinonim za ||
||	zaporedni logični or

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

=	strukturna enakost
==	fizična enakost
<>	negacija =
!=	negacija ==
<	manjše
>	večje
<=	manjše ali enako
>=	večje ali enako

```
# 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' ;;
Characters 4-7:
This expression has type char but is here used with type int
```

Seznami

```
# [] ;;
- : 'a list = []
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two"; 3 ] ;;
Characters 14-17:
This expression has type int list but is here used with type string list
```

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

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

N-terice

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

```
# 12 , "October";;
- : int * string = 12, "October"
The functions fst and snd allow access to the first and second elements of a pair.
# fst ( 12 , "October") ;;
- : int = 12
# snd ( 12 , "October") ;;
- : string = "October"
```

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

Imenski prostori

- Globalna deklaracija
- Lokalna deklaracija

Funkcijski jeziki

- Funkcijski izrazi
- Definicija funkcij
- Rekurzivne funkcije
- Polimorfizem
- Funkcije višjega reda

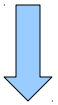
Funkcijski izrazi

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

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

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

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



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

Definicija funkcij

```
# let succ = function x -> x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
```

```
# let succ x = x + 1 ;;
val succ : int -> int = <fun>
```

```
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

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

Rekurzivne funkcije

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

```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
  and     odd n = (n<>0) && ((n=1) or (even (n-1)));;
```

Polimorfizem

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

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

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

Funkcije višjega reda

```
# 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
```

```
# 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>
```

```
# 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
```

Imperativni jeziki

- Spremenljivke
- Sekvenčna kontrola
  - Sekvenca
  - Pogojni stavek
  - Zanke
  - Vzorci
- Implementacija funkcij
- Polja
- Matrike

Spremenljivke

```
int x = 1, y = 2, z = 3;
int *ip;
ip = &x;
```

```
int i;
```

```
y = *ip;
*ip = 0;
ip = &z;
```

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# x ;;
- : int ref = {contents=3}
```

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

Sekvenca

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

```
# 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>
```

Pogojni stavek

```
# if 3=4 then 0 else 4 ;;
- : int = 4
```

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

Zanke

```
# 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 = ()
```

```
# 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 = ()
```

Vzorci

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

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

```
# 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>
```

Polja

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

```
# for i=0 to (n-1) do v.(i)<-i done;;
- : unit = ()
# v;;
- : int array = [|0; 1; 2; 3; 4; 5; 6; 7; 8; 9|]
```

```
# let tmp=ref 0
  in for i=0 to (n/2-1) do
       tmp := v.(i);
       v.(i) <- v.(n-i-1);
       v.(n-i-1) <- (!tmp);
     done;;
- : unit = ()
# v;;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
```

Matrike

```
# let n = 3 and m=3;;
val n : int = 3
val m : int = 3
# let a = Array.create_matrix n m 0.0;;
val a : float array array =
  [|[|0.; 0.; 0.|]; [|0.; 0.; 0.|]; [|0.; 0.; 0.|]|]
```

```
# let add_mat a b =
    let r = Array.create_matrix n m 0.0 in
      for i = 0 to (n-1) do
        for j = 0 to (m-1) do
          r.(i).(j) <- a.(i).(j) +. b.(i).(j)
        done
      done ; r;;
val add_mat : float array array -> float array array -> float array array = <fun>
```

Tipi

- Atomični tipi
- Produkti
  - N-terice
  - Zapisi
- Unije
- Podtipi
- Rekurzivni tipi

Atomični tipi

- int
- real
- string
- bool

Produkti

```
# type triple = int*int*int;;
type triple = int * int * int
# let l = 1,2,3;;
val l : int * int * int = (1, 2, 3)
# let t:triple = 1,2,3;;
val t : triple = (1, 2, 3)
```

```
# type ('a,'b,'c) triple = 'a*'b*'c;;
type ('a, 'b, 'c) triple = 'a * 'b * 'c
# let t = 1,'1',"1";;
val t : int * char * string = (1, '1', "1")
# let t:(int,char,string) triple = 1,'1',"1";;
val t : (int, char, string) triple = (1, '1', "1")
```

Zapisi

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

```
# 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}
```

Unije

```
# type suit = Spades | Hearts | Diamonds | Clubs ;;
# type card =
        King of suit
      | Queen of suit
      | Knight of suit
      | Knave of suit
      | Minor_card of suit * int
      | Trump of int
      | Joker ;;
```

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

```
# let string_of_suit = function
        Spades   -> "spades"
      | Diamonds -> "diamonds"
      | Hearts   -> "hearts"
      | Clubs    -> "clubs";;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
        King c              -> "king of "^ (string_of_suit c)
      | Queen c             -> "queen of "^ (string_of_suit c)
      | Knave c             -> "knave of "^ (string_of_suit c)
      | Knight c            -> "knight of "^ (string_of_suit c)
      | Minor_card (c, n) -> (string_of_int n) ^ "of "^(string_of_suit c
      | Trump n             -> (string_of_int n) ^ "of trumps"
      | Joker               -> "joker";;
val string_of_card : card -> string = <fun>
```

Rekurzivni tipi

```
# type 'a rnode = { mutable cont:'a; mutable next:'a rlist }
  and  'a rlist =
          Nil
        | Elm of 'a rnode;;
type 'a rnode = { mutable cont : 'a; mutable next : 'a rlist; }
and 'a rlist = Nil | Elm of 'a rlist
# let l1 = Elm {cont=1;next=Elm {cont=2;next=Nil}};;
val l1 : int rlist =
  Elm {cont = 1; next = Elm {cont = 2; next = Nil}}
```

```
# type 'a bin_tree =
       Empty
     | Node of 'a bin_tree * 'a * 'a bin_tree ;;
# let rec list_of_tree = function
       Empty -> []
     | Node(lb, r, rb) -> (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

Objektno-usmerjeni jeziki

- Razredi
  - Agregacija
- Specializacija
  - Dedovanje
  - Self, super
  - Pretvorba tipov
  - Dinamično povezovanje
- Generativnost
  - Abstraktni razredi
  - Parametrizirani razredi

Razredi

```
# class point (x0,y0) =
      object(self)
        val mutable x = x0
        val mutable y = y0
        val mutable old_x = x0
        val mutable old_y = y0
        method get_x = x
        method get_y = y
        method private mem_pos () = old_x <- x ; old_y <- y
        method undo () = x <- old_x; y <- old_y
        method moveto (x1, y1) = self#mem_pos (); x <- x1; y <- y1
        method rmoveto (dx, dy) = self#mem_pos (); x <- x+dx; y <- y+dy
        method to_string () =
               "("^ (string_of_int x) ^ ","^ (string_of_int y) ^")"
        method distance () = sqrt (float(x*x + y*y))
    end ;;
```

Agregacija

```
# class picture n =
      object
        val mutable ind = 0
        val tab = Array.create n (new point(0,0))
        method add p = tab.(ind)<-p ; ind <- ind + 1
        method remove () = if (ind > 0) then ind <-ind-1
        method to_string () =
          let s = ref "["
          in for i=0 to ind-1 do s:= !s ^ " "^ tab.(i)#to_string () done ;
              (!s) ^ "]"
      end ;;
```

```
# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string () ;;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0) ]"
```

Specializacija

```
# class colored_point (x,y) c =
      object
        inherit point (x,y)
        val mutable c = c
        method get_color = c
        method set_color nc = c <- nc
        method to_string () = "("^ (string_of_int x) ^
                              ","^ (string_of_int y) ^ ")"^
                              "["^ c ^ "]"
      end ;;
```

Pretvorba tipov

```
(name : sub_type :> super_type)
(name :> super_type)
```

```
# let cp = new colored_point (1,1) "red";;
val cp : colored_point = <obj>
# let p = (cp :> point);;
val p : point = <obj>
# p#get_color ();;
Error: This expression has type point
       It has no method get_color
# p#to_string ();;
- : string = "(1,1)[red]"
```

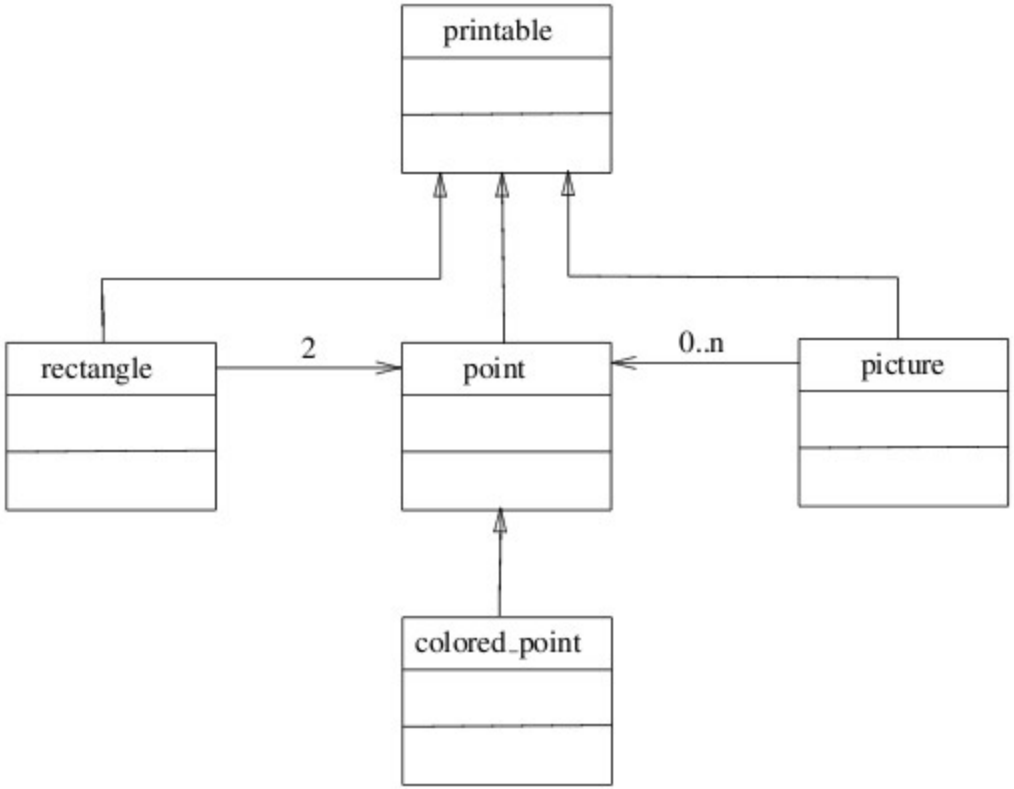
Dinamično povezovanje

```
# class picture n =
      object
        val mutable ind = 0
        val tab = Array.create n (new point(0,0))
        method add p = tab.(ind)<-p ; ind <- ind + 1
        method remove () = if (ind > 0) then ind <-ind-1
        method to_string () =
          let s = ref "["
          in for i=0 to ind-1 do s:= !s ^ " "^ tab.(i)#to_string () done ;
              (!s) ^ "]"
      end ;;
```

```
# let pic = new picture 3;
>> Creation of point: (0,0)
val pic : picture = <obj>
# pic#add (new point (1,1));
  pic#add ((new colored_point (2,2) "red") :> point);
  pic#add ((new verbose_point (3,3)) :> point);;
- : unit = ()
# pic#to_string () ;;
- : string = "[ (1,1) (2,2)[red] point=(3,3),distance=4.24264068712]"
```

Abstraktni razredi

```
# class virtual printable () =
        object(self)
          method virtual to_string : unit -> string
          method print () = print_string (self#to_string () )
        end ;;
```



Parametrizirani tipi

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
      object
        val x = x0
        val y = y0
        method fst = x
        method snd = y
      end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

Modularni jeziki

- Moduli kot enota prevajanja
- Jezik modulov
  - Skrivanje informacij
  - Več pogedov na modul
- Funktorji
  - Generativnost

Moduli kot enota prevajanja

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with hd :: tl -> s.c <- tl; hd
            |   []   -> raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

```
type 'a t
exception Empty

val create: unit -> 'a t
val push: 'a -> 'a t -> unit
val pop: 'a t -> 'a
val clear : 'a t -> unit
val length: 'a t -> int
```

```
open Stack;;
let s = create ();;
push 1 s; push 2 s; push 3 s;;
Printf.printf "elementi: %i, %i in %i\n"
              (pop s) (pop s) (pop s);;
```

Jezik modulov

```
# module PairOfLists = struct
    type 'a t = ('a list * 'a list) ref
    exception Empty
    let create () = ref ([], [])

    let enqueue x queue =
      let front, back = !queue in
          queue := (x::front, back)

    let rec dequeue queue =
      match !queue with
        (front, x :: back) -> queue := (front, back);
                              x
      | ([], []) -> raise Empty
      | (front, []) -> queue := ([], List.rev front);
                       dequeue queue

    let push x queue = enqueue x queue

    let rec pop queue =
      match !queue with
        (x::front,back) -> queue := (front,back); x
      | ([],[]) -> raise Empty
      | ([],back) -> queue := (List.rev back,[]);
                     pop queue
  end;;
```

Skrivanje informacij

```
# module type Stack =
      sig
        type 'a t
        exception Empty
        val create: unit -> 'a t
        val push: 'a -> 'a t -> unit
        val pop: 'a t -> 'a
      end ;;
```

```
# module Stack1 = (PairOfLists:Stack);;
module Stack1 : Stack
```

```
# let s = Stack1.create ();;
val s : '_a Stack1.t = <abstr>
# Stack1.push 1 s;;
- : unit = ()
# Stack1.push 2 s;;
- : unit = ()
# Stack1.pop s;;
- : int = 2
# Stack1.pop s;;
- : int = 1
```

Več pogedov na modul

```
# module type Queue =
      sig
        type 'a t
        exception Empty
        val create: unit -> 'a t
        val enqueue: 'a -> 'a t -> unit
        val dequeue: 'a t -> 'a
      end ;;
```

```
# module Queue1 = (PairOfLists:Queue);;
module Queue1 : Queue
# let v = Queue1.create ();;
val v : '_a Queue1.t = <abstr>
# Queue1.enqueue 1 v; Queue1.enqueue 2 v;;
- : unit = ()
# Queue1.dequeue v;;
- : int = 1
# Queue1.dequeue v;;
- : int = 2
```

Funktorji

```
# type comparison = Less | Equal | Greater;;
type comparison = Less | Equal | Greater

# module type ORDERED_TYPE =
   sig
     type t
     val compare: t -> t -> comparison
   end;;
module type ORDERED_TYPE = sig type t val compare : t -> t -> comparison end

# module Set =
   functor (Elt: ORDERED_TYPE) ->
     struct
       type element = Elt.t
       type set = element list
       let empty = []
       let rec add x s =
         match s with
           [] -> [x]
         | hd::tl ->
            match Elt.compare x hd with
              Equal   -> s         (* x is already in s *)
            | Less    -> x :: s    (* x is smaller than all elements of s *)
            | Greater -> hd :: add x tl
       let rec member x s =
         match s with
           [] -> false
         | hd::tl ->
             match Elt.compare x hd with
               Equal   -> true     (* x belongs to s *)
             | Less    -> false    (* x is smaller than all elements of s *)
             | Greater -> member x tl
     end;;
module Set :
```

Drugi koncepti

- Vzporednost
  - Procesi in niti
  - Sinhronizacija
  - Korutine