Predavanji 5-6

Imperativni jeziki

Iztok Savnik, FAMNIT

Marec, 2024.

Literatura

- Učbenika:
  - John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 5 and 8)
  - Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapters 6 and 8)
- Več primerov iz:
  - Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, English translation, O'REILLY, 2000

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Imperative languages

- Funkcijski jeziki
  - Abstraktni model je λ-račun
  - Program je funkcija
    - Rezultat se izračuna z redukcijo
  - Funkcija nima stranskih učinkov
    - Ne spreminja stanje spremenljivk izven funkcij
- Imperativni jeziki
  - Abstrakten model je Turingov stroj
  - Program je sekvenca instrukcij
    - Program ima stanja
    - Rezultat se zgradi z izvajanjem inštrukcij
  - Inštrukcije spreminjajo stanje glavnega spomina
    - Rezultat se izračuna, ko automat pride v končno stanje

Imperativno programiranje

- Začetni imperativni programski jeziki
  - Fortran, 1954; Jezik matematičnih enačb; zanke, procedure
  - BASIC, 1960; Splošno-namenska koda sestavljena iz abstraktnih inštrukcij
  - Pascal, 1970; Algoritmi + Podatkovne strukture = Program
  - C, 1972; gradniki C se lepo prevedejo v tipičen nabor inštrukcij
- Trenutno stanje teh jezikov
  - Januar 2021, C je prvi jezik v TIOBE indeksu!
  - C na vrhu drugih lestvic: najbolj iskani, ponudbe zaposlitev, iskanje na spletu
  - Fortran je še vedno med najbolj popularnimi jeziki za numerično procesiranje

Primer

Funkcija, ki izračuna največji skupni delitelj dveh števil.

OCaml

```
let rec gcd x y =
    if y = 0 then x
    else gcd y (x mod y);;
```

C

```
int gcd(int x, int y) {
   while (y != 0) {
        int t = x % y;
        x = y; y = t;
    }
    return x;
}
```

Funkcijski: vrednosti, rekurzija

Imperativni: spremenljivke, zanke, sekvence

Strukturirana kontrola

- Strukturirano programiranje, 1970
  - Velik vpliv na imperativne programske jezike
  - Strukturirana in nestrukturirana kontrola
    - Nestrukturirana: GOTO stavki
    - Strukturirana: sintaktični gradniki definirajo strukt. kontrolo
  - "Revolucija" v programskem inženirstvu
    - Načrtovanje programskih sistemov od zgoraj navzdol (inkrementalne izboljšave)
    - Lepo definirane, abstraktne zanke (while-do, do-while, for)
    - Strukturirani tipi (zapisi, množice, kazalci, več-dimenzionalna polja)
    - Procedure, funkcije in modularizacija kode
    - Opisna imena spremenljivk in konstant
    - Dogovori glede dokumentiranja

Strukturirano programiranje

- Večina idej iz strukturiranega programiranja je bilo implementiranih v imperativnih jezikih
  - Pascal, C, Modula, Ada, Oberon, Java, C#, ...
- Večina klasičnih algoritmov je napisana v imperativnih programskih jezikih!
  - Dijkstra, Floyd, Knuth, itd.

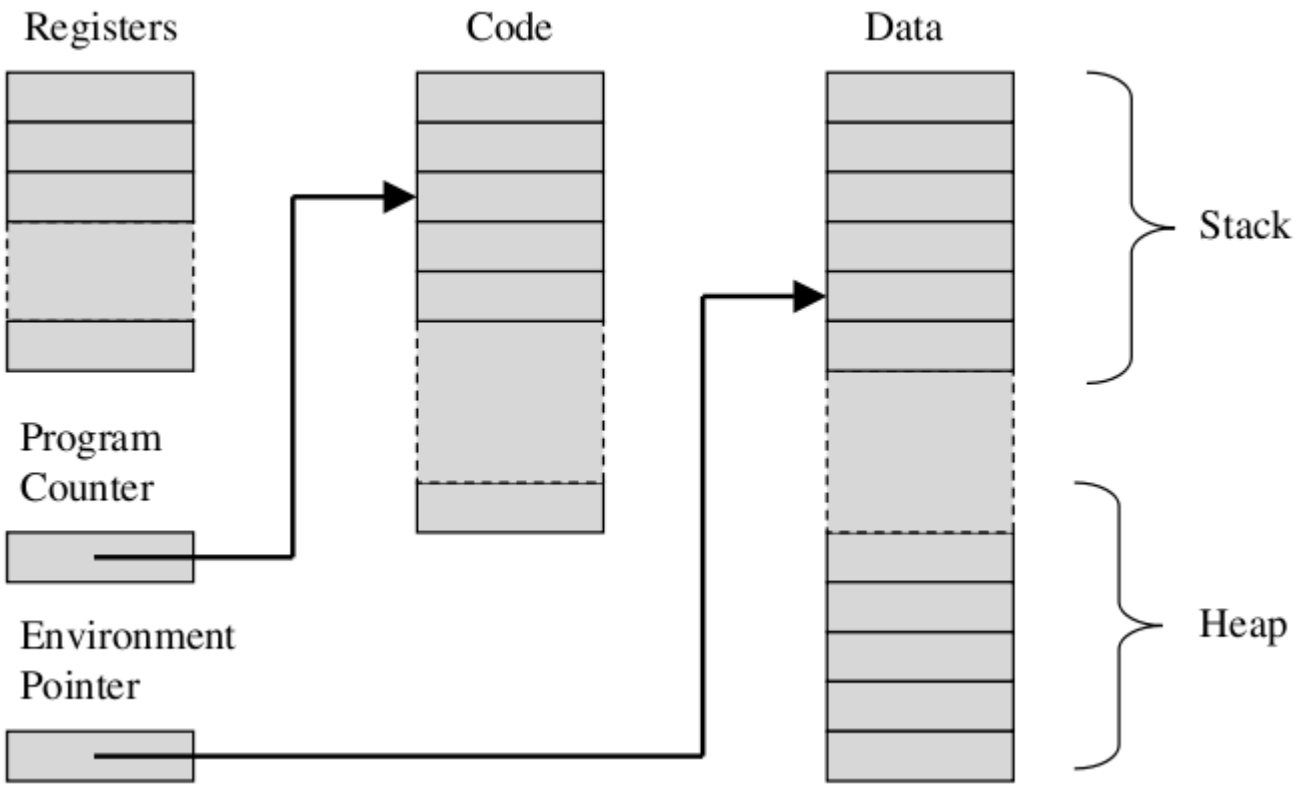
Koncepti imperativnih prog. jezikov

- Brano/pisalni spomin in spremenljivke
- Ukazi in sekvence ukazov
- Bloki
- Pogojni stavki
- Zanke s pogoji; iteracije preko intervalov ali kontejnerjev
- Procedure in funcije
- Zapisi, kazalci in polja
- Množice, unije in slovarji

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Spomin programa



Spremenljivke

- Spomin programa je oraniziran v spominske celice
- Do vsake celice lahko dostopamo z naslovom celice; naslov je celo število iz intervala 0 – (2^{62} − 1)
- Spremenljivka je simbolično ime za neko spominsko celico določene velikosti
  - Do spremenljivke dostopamo preko imena namesto preko naslova spominske celice.
- Imenski prostori programa
  - Programski kontekst (blok, funkcija, modul)
  - Imenski prostor vsebuje podatke o spremenljivkah in funkcijah
    - Ime (identifikator), naslov začetka in velikost spominske celice.
  - Imenski prostori so pogosto organizirani hierarhično

Operacije na spremenljivkah

- Program mora zaseči pomnilniški prostor spremenljivke preden se spremenljivka uporablja.
  - Alokacija je lahko statična ali dinamična.
- Program prebere vsebino pom. prostora spremenljivke, ko dostopamo do imena spremenljivke.
- Vsebina spomina, kjer je shranjena spremenljivka se spremeni pri prireditvi vrednosti.
- Spremenljivka je sproščena na koncu programa, na koncu bloka ali na zahtevo.
- Možni problemi:
  - branje/pisanje nealociranega spomina, vzporedno pisanje, puščanje spomina
  - Predavanje o delu s spominom

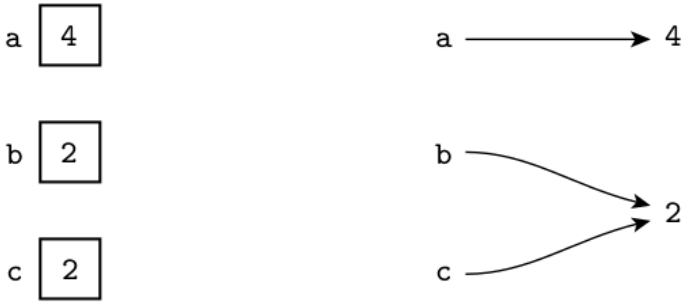
Modeli spremenljivk

- Dva modela spremenljivk
- Vrednostni model spremenljivk
  - Spremenljivka je imenovan kontejner vrednosti
  - Lokacija in vrednost (poglej spremenljivko a)
    - l-vrednost = leva stran prireditvenega stavka
    - r-vrednost = izrazi, ki se ovrednotijo v vrednost
    - Oboje je lahko kompleksen izraz
  - Izraz je bodisi l-vrednost ali r-vrednost, odvisno od konteksta
  - C, Pascal, Java, itd.

```
d = a;
a = b + c;
```

Modeli spremenljivk

- Model referenc spremenljivk



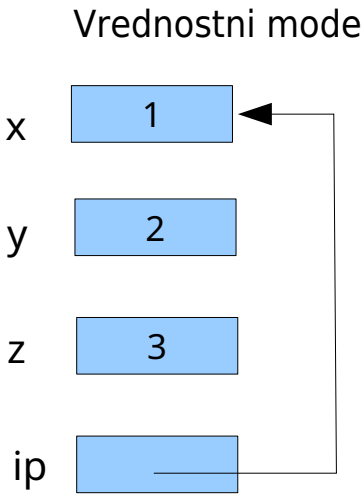

```
b := 2;
c := b;
a := b + c;
```

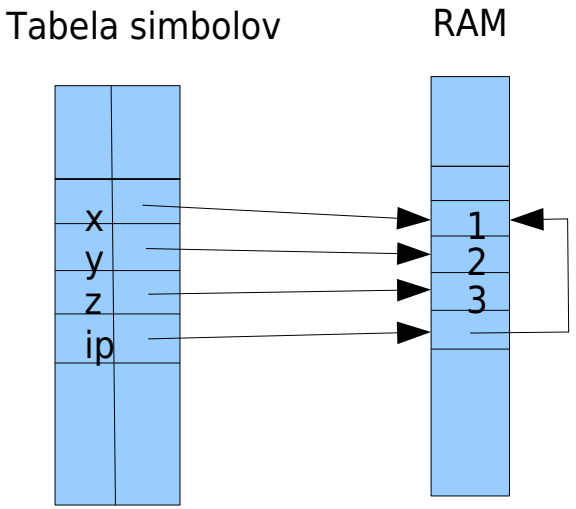
  - Spremenljivka je imenovana referenca na vrednost
    - Vsaka spremenljivka je l-vrednost
    - Spremenljivka mora biti dereferencirana v kontekstu r-vrednosti
    - Dereferciranje je automatsko v večini PJ, vendar ne v ML
  - Model referenc (ni) dražji od modela vrednosti
    - Uporablja več kopij nespremenljivih objektov
  - Algol68, Clu, Lisp/Scheme, ML, Haskell, and Smalltalk

Spremenljivke v C

- Dve pomembni operaciji
  - Operator »&«: vrne naslov spremenljivke
  - Operator »*«: vrne vrednost spremenljivke (iz naslova)

```
int x = 1, y = 2, z = 3;
int *ip;
ip = &x;
y = *ip;
*ip = 0;
*ip = *ip + 10;
y = *ip + 1;
z = *ip * 10;
```

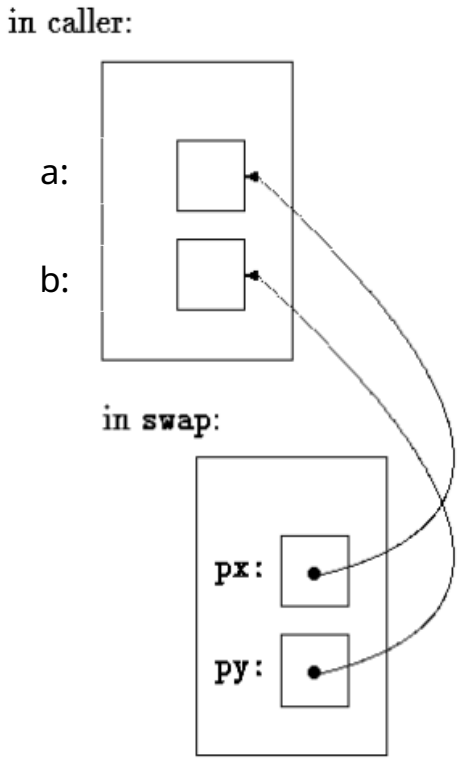




Dve pomembni operacije

- Operator »&«: vrne naslov spremenljivke
- Operator »*«: vrne vrednost spremenljivke

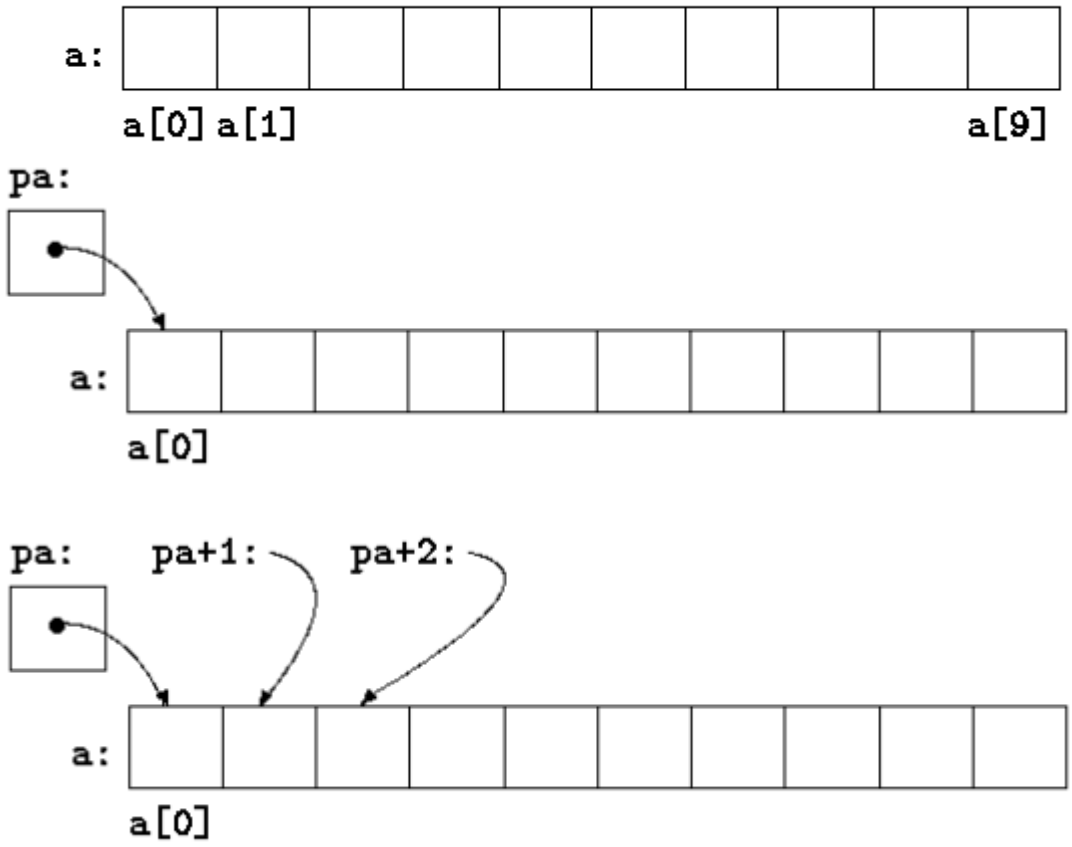
```
swap(&a, &b);
...
void swap(int *px, int *py) {
   int temp;
   /* interchange *px and *py */
   temp = *px;
   *px = *py;
   *py = temp;
}
```



Kazalci in polja v C

- C ima aritmetiko kazalcev

```
int a[10];
// a[0], a[1],..., a[9]
pa = &a[0];
```




```
/* strlen: return length of string s */
int strlen(char *s) {
   int n;
   for (n = 0; *s != '\0', s++)
        n++;
   return n;
}
```

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

Spremenljivke v OCaml

- Spremenljivke so implementirane na osnovi tipa referenca (tip ref)
- OCaml ima šibkejši vendar varnejši model (od C)
  - Reference se inicializirajo ob kreaciji z referencirano vrednostjo
  - Spominski prostor je automatično alociran z uporabo tipa reference
  - Prireditev je posebna funkcija s tipom rezultata unit
  - Rezultat branja ima tip referencirane spremenljivke
- Slabe lastnosti modela
  - Funkcij se ne da referencirati
  - Nimamo polnega dostopa do spomina programa – ni kazalcev, ni aritmetike naslovov

Primeri spremenljivk v Ocaml

```
# let x = ref 2 ;;                                    (* declaration and allocation
val x : int ref = {contents=2}
# !x;;                                                (* reading, notice operator '!'
- : int = 2
# x ;;                                                (* reading of the reference
- : int ref = {contents=2}
# x := 5; !x;;                                        (* assignment
- : int = 5
# x := !x * !x; !x;;                                  (* reading, operation, and assignment
- : int = 25
# let l = ref [1;2;3];;
val l : int list ref = {contents = [1; 2; 3]}
# !l;;
- : int list = [1; 2; 3]
# l := 0::!l; !l;;
- : int list = [0; 1; 2; 3]
```

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Sekvenca

- Sekvenca je osnovna abstrakcija za opisovanje algoritmov
  - Instrukcijski cikel Von Neumann modela

```
LOOP: execute *PC;      // execute instruction referenced by PC
      PC++;             // increment programm counter (PC) by 1
      goto LOOP;        // loop
```

- Sekvenca instrukcij spreminja stanje spremenljivk (v spominu)

```
int t = x % y;
x = y;
y = t;
```

Sekvence v OCaml

- Sintaksa OCaml sekvenc

```
let t = ref 0
let x = ref 42
let y = ref 28 in begin
    t:= x mod y;
    x:=!y;
    y:=!x;
end;;
```

```
<expr1>; <expr2>; <expr3>;; (* list of exprs *)
```

- Vsak izraz v OCaml sekvenci mora biti tipa `unit`.

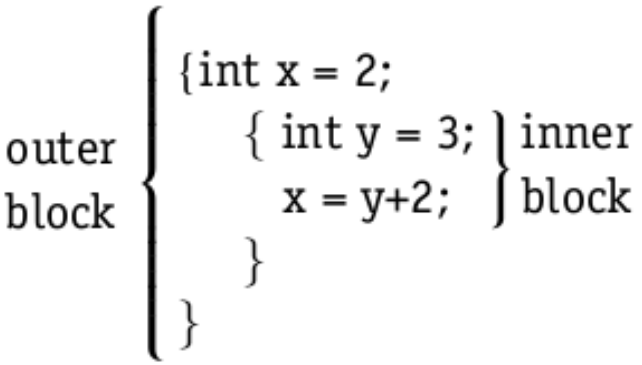
```
# print_int 1; 2; 3;;
Warning 10: this expression should have type unit.
1- : int = 3
```

```
# print_int 2; ignore 4; 6;;
2- : int = 6
```

Bloki

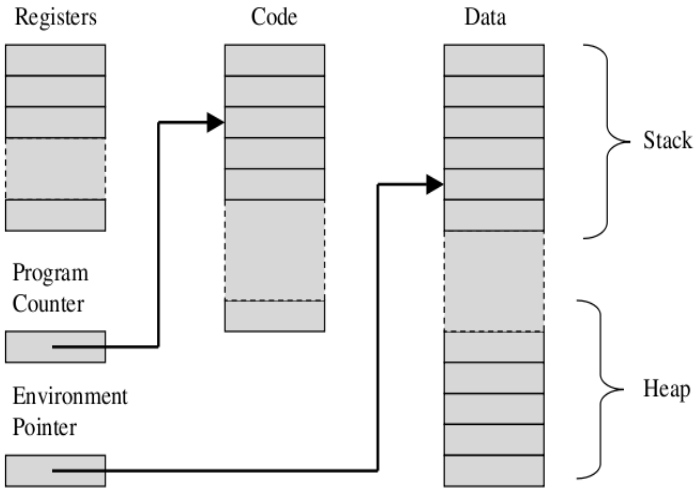
- Imperativni programi so tipično strukturirani v bloke
- Blok je sekvenca stavkov obdana s strukturno jezikovno formo
  - begin-end bloki, blok zanke, telo funkcije, itd.

```
begin
    let t = ref 0
    and x = ref 42
    and y = ref 28 in
    begin
        t:= !x mod !y;
        x:=!y;
        y:=!x;
    end;
end;;
```



Bloki

- Vsak blok je predstavljen z aktivacijskim zapisom
  - Vsebuje parametre in lokalne spremenljivke
  - Vsebuje spominsko lokacijo za vrnejno vrednost
  - Vsebuje kontrolne kazalce (seledeča predavanja)
  - Kontrolni kazalci so uporabljeni za kontrolo izvajanja
- Aktivacijski zapisi so zaseženi na prog. skladu
  - Predstavljeni bodo na predavanju Upravljanje spomina



Pogojni stavki

JE/JZ	Jump if equal/Jump if zero
JNE/JNZ	Jump if not equal/Jump if not zero
JA/JNBE	Jump if above/Jump if not below or equal
JAE/JNB	Jump if above or equal/Jump if not below
JB/JNAE	Jump if below/Jump if not above or equal
JBE/JNA	Jump if below or equal/Jump if not above
JG/JNLE	Jump if greater/Jump if not less or equal
JGE/JNL	Jump if greater or equal/Jump if not less
JL/JNGE	Jump if less/Jump if not greater or equal
JLE/JNG	Jump if less or equal/Jump if not greater
JC	Jump if carry
JNC	Jump if not carry

- Strojni jezik uporablja inštrukcije za pogojne skoke
  - Začetni imperativni pristop
- OCaml sintaksa

```
if <cond_expr> then <expr_true> else <expr_false> ;;
```

- Pogojni stavek je koncept PJ, ki je uporabljen v imperativnem in funkcijskem modelu jezikov
  - Obe veji se morajo ujemati v tipu (Ocaml)
  - Pogojni stavek v Ocamlu ima vrednost!

```
# (if 1 = 0 then 1 else 2) + 10;;
- : int = 12
```

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Zanke – while-do

- Ponavljaj blok ukazov dokler je while pogoj izpolnjen
  - Telo zanke spreminja stanje programa
- Stavek `while` v OCaml

```
while <cond_expr> do
    <sequence>
done
```

```
# let gcd (x,y) =
      let t = ref 0 in
      while !y != 0 do
          t := !x mod !y; x := !y; y := !t
      done; !x;;
val gcd : int ref * int ref -> int = <fun>
# let a = ref 42 and b = ref 28; (!a,!b);
val a : int ref = {contents = 42}
val b : int ref = {contents = 28}
# gcd (a,b);;
- : int = 14
```

Zanke – `for` stavek

- Stavek `for` je klasičen konstrukt imperativnih programskih jezikov.
- Stavek `for` v Ocaml

```
for <sym> = <exp1> to <exp2> do
    <exp3>
done
for <sym> = <exp1> downto <exp2> do
    <exp3>
done
```

```
/* atoi: convert s to integer */
int atoi(char s[])
{
    int i, n, sign;
    for (i = 0; isspace(s[i]); i++) ;        /* skip white space */
    sign = (s[i] == '-') ? -1 : 1;
    if (s[i] == '+' || s[i] == '-')  i++;    /* skip sign */
    for (n = 0; isdigit(s[i]); i++)          /* convert s to integer */
        n = 10 * n + (s[i] - '0');
    return sign * n;
}
```

- for stavek v
  - imperativnih,
  - skriptnih,
  - OO in
  - modularnih

  programskih jezikih.

Ocaml – for -> while stavek

```
let is_digit = function '0' .. '9' -> true | _ -> false;;
let is_white = function ' ' | '\n' | '\t' -> true | _ -> false;;
```

```
let int_of_string s =
   begin
      let i = ref 0 and n = ref 0 in
      while is_white(s.[!i]) do i := !i+1; done;
      let sign = (if s.[!i]='-' then -1 else 1) in
      if s.[!i]='+' || s.[!i]='-' then i := !i+1;
      while is_digit(s.[!i]) do
         n := 10 * !n + (int_of_char(s.[!i]) - int_of_char('0'));
         i := !i+1;
      done;
      sign * !n;
   end;;
```

```
# let s = " -12\n";;
val s : string = " -12\n"
# int_of_string s;;
- : int = -12
```

Zanke – do-while stavek

- Pogoj zanke je na koncu bloka zanke
- Ni vključen v Ocaml!

```
do <sequence>
while <cond_expr>
```

- Stavek repeat-until
  - Kernighan & Ritchie: The programming language C
  - Konverzija `int` v `char*` (`string`)

```
/* itoa: convert n to characters in s */
void itoa(int n, char s[]) {
   int i, sign;
   if ((sign = n) < 0) /* record sign */
      n = -n;              /* make n positive */
   i = 0;
   do {     /* generate digits in reverse order */
      s[i++] = n % 10 + '0';     /* get next digit */
   } while ((n /= 10) > 0);       /* delete it */
   if (sign < 0)
      s[i++] = '-';
   s[i] = '\0';
   reverse(s);
}
```

Ocaml – do-while -> while stavek

```
let string_of_int n =
   begin
      let s = Bytes.make 10 ' ' and sign = n and nr = ref n and i = ref 0 in
      if sign<0 then nr := -n;
      Bytes.set s !i (char_of_int(!nr mod 10 + int_of_char('0')));
      nr := !nr / 10;
      while (!nr > 0) do
         i := !i+1;
         Bytes.set s !i (char_of_int(!nr mod 10 + int_of_char('0')));
         nr := !nr / 10;
      done;
      if (sign < 0) then begin i := !i+1; Bytes.set s !i '-'; end;
      reverse(Bytes.to_string (Bytes.trim s));
   end;;
```

Kontrola zanke

- Kontrola zank v program. jeziku C
  - Skok izven zanke – break
  - Skok na pogoj zanke – continue
  - Ni vsebovano v Ocaml

```
for (i = 0; i < n; i++)
    if (a[i] < 0)
        /* skip negative elements */
       continue;
    ...
        /* do positive elements */
```

  - Kernighan & Ritchie: The programming language C

```
/* trim: remove trailing blanks, tabs, newlines */
int trim(char s[]) {
    int n;
    for (n = strlen(s)-1; n >= 0; n--)
        if (s[n] != ' ' && s[n] != '\t' && s[n] != '\n')
            break;
    s[n+1] = '\0';
    return n;
}
```

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Procedure in funkcije

- Abstrakcija je proces s katerim programer poveže simbol ali vzorec z gradnikom programskega jezika
  - Kontrolne in podatkovne abstrakcije
- Podprogrami so osnovni mehanizem za kontrolno abstrakcijo
  - Del programa z dobro definiranim vhodom in izhodom
    - Podprogram, procedura, funkcija
  - Podprogram izvaja operacijo na zahtevo oz. za klic iz danega okolja
  - Klic podprograma poda argumente z uporabo parametrov
  - Podprogram, ki vrne vrednost je funkcija
  - Podprogram, ki ne vrne vrednosti se imenuje procedura

Procedure in funkcije

- Procedura je prva abstrakcija Algolske družine programskih jezikov
- Procedure in funkcije imajo parametre
  - Formalni in dejanski parametri procedure

```
procedure Proc(First : Integer; Second: Character);
Proc(24,'h');
```

  - Dejanski parametri se preslikajo v formalne parametre
- Metode za prenos parametrov
  - Nekateri jeziki uporabljajo eno samo množico pravil, ki veljajo za vse parametre (C, Java, Fortran, ML, and Lisp)
  - Ostali PJ imajo več različnih metod za prenos parametrov (Pascal, C++, Ada, ...)

Prenos parametrov

- Vhod in izhod procedure je narejen s prenosom parametrov
  - Prenos vrednosti
    - C (samo cbv), Java, Ocaml, C++, Pascal, ...
  - Prenos referenc
    - Pascal, C++, Fortran, …

```
Procedure Square(Index : Integer;
                 Var Result : Integer);
 Begin
   Result := Index * Index;
 End
```

Pascal

- Druga pravila za prenos parametrov
  - Prenos strukturiranih stvari
    - Polja, strukture, objekti
  - Manjkajoče and privzete vrednosti
  - Imenovani parametri
  - Seznam argumentov s spremenljivo dolžino

Prenos po vrednosti

- Najbolj pogosto uporabljena metoda
  - Vrednosti dejanskih param se prepiše v formalne param
  - Java uporablja samo to metodo (polja in strukture se identificirajo z referencami)
- Parameter je viden kot lokalna sprem. procedure
  - Lokalna sprem se inicializira z dejanskimi param

C

```
int plus(int a, int b) {
  a += b;
  return a;
}
int f() {
  int x = 3; int y = 4;
  return plus(x, y);
}
```

Ocaml

```
# let plus ((a:int), (b:int)) : int = a + b;;
val plus : int * int -> int = <fun>
# let f () =
    let x = 3 and y = 4
    in plus (x,y);;
val f : unit -> int = <fun>
# f ();;
- : int = 7
```

Prenos po referenci

- Parameter je referenca na spremenljivko
  - Koda procedure spreminja podano sprem (preko refer.)
  - Vse spremembe se ohranijo po klicu
  - Prenešena spremenljivka in formalni param sta vzdevka
- Najboljša metoda za velike strukture

C

```
void plus(int a, int *b) {
   *b += a;
}
...
int x = 3;
plus(4, &x);
// x == 7
...
```

Ocaml

```
# let plus ((a:int), (b:int ref)) : unit =
    b := a + !b;;
val plus : int * int ref -> unit = <fun>
# let a = 4 and b = ref 3;;
val a : int = 4
val b : int ref = {contents = 3}
# plus (a,b);;
- : unit = ()
# !b;;
- : int = 7
```

Variacije: vrednost in referenca

- C++ reference
  - V C++, so definirane C reference eksplicitno
  - C++ implementira klic-po-referenci

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

- Klic-z-deljenjem (Call-by-sharing)
  - Barbara Liskov, CLU (tudi Smalltalk)
  - Objekti so predstavljeni z referencami oz. identifikatorji
    - V porazdeljenem svetu referenca ni pomnilniški naslov!
  - Ni potrebe po prenosu referenc (na reference)
  - Samo prenesi referenco!

Variacije: vrednost in referenca

- Klic-po-vrednosti/rezultatu
  - Dejanski param se prepišejo v formalne parametre
  - Rezultat se prepiše nazaj v dejanski parameter pred zaključkom
- Bralni parametri
  - Vrednosti parametra se ne smejo spreminjati
  - Modula-3 omogoča bralne parametre
  - Bralni parametri so definirani tudi v C (const)
- Parametri v PJ Ada
  - in, out, in out (tudi v Oracle PL/SQL)
  - Imenovani parametri / pozicijski parametri

Variacije: vrednost in referenca

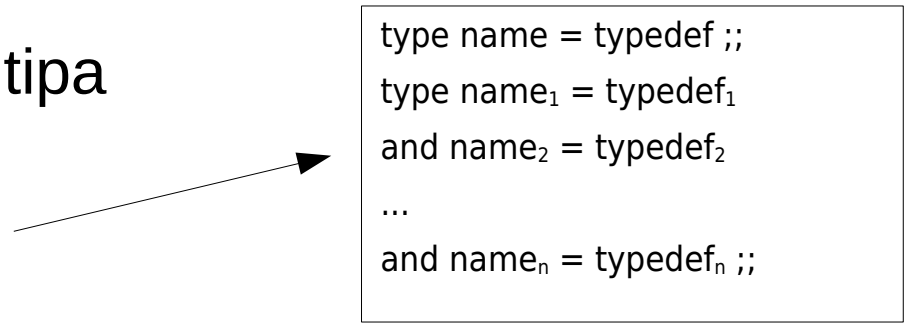
- Privzete vrednosti parametrov
  - Ada, Oracle PL/SQL
- Spremenljivo število parametrov
  - Programski jeziki C, Perl, ...
  - Ni preverjanja tipov, ni kontrole.
  - Gradnik je lahko nevaren.

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Definicija tipa

- Uporabniški tip lahko definiramo iz bolj enostavnih tipov
  - Z uporabo konstruktorjev tipa
  - *, |, record, list, array, ...
- Definicija tipov v Ocaml
- V imperativnih jezikih nimamo parametriziranih (polimorfičnih) tipov!
  - Samo konkretni tipi
- Zapisi, kazalci in polja! Značilne strukture imperativnih prog. jezikov
  - Tipi imperativnih prog. jezikov so predstavljeni v tem poglavju



Zapisi

- Zapisi omogočajo shranjevanje in rokovanje s povezanimi podatki, ki imajo različne tipe.
- Zapisi v programskih jezikih
  - Originalno predstavljeni v Cobol
  - V Algolu 68 so jih imenovali strukture (tudi v C)
    - Uporabljajo ključno besedo `struct`
    - Kasneje v Fortran 90 jih imenujejo zapisi (*tip zapis*)
  - C++ strukture so poseben primer razredov
  - Java nima struktur (samo razrede)
  - C# in Swift uporabljata model referenc za razrede in model vrednosti za strukture (`struct`); brez dedovanja.

Zapisi v C

- V C definiramo enostaven zapis na sledeč način:
- Komponente zapisa imenujemo tudi polja.
- Za dostop do polja zapisa večina PJ uporablja notacijo s piko.

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```

```
element copper;
const double AN = 6.022e23;     /* Avogadro's number */
...
copper.name[0] = 'C'; copper.name[1] = 'u';
double atoms = mass / copper.atomic_weight * AN;
```

Zapisi v Ocaml

- Zapis je produkt z imenovanimi komponentami.
- Definicija tipa in konstrukcija vrednosti:

type name = { $name_1$: t_1 ; . . . ; $name_n$: t_n }
{ $name_1$ = $expr_1$; . . . ; $name_n$ = $expr_n$ }

- Komponente zapisov so lahko spremenljive.
  - Prireditev vrednosti komponenti

type name = { ...; mutable $name_i$: t_i ; ... }

expr1.name <- expr2

```
# type complex = { mutable re:float;
                   mutable im:float } ;;
type complex = { mutable re : float;
                 mutable im : float; }
# let c = {re=2.;im=0.} ;;
val c : complex = {re=2; im=0}
```

```
# c.im <- 3.;;
- : unit = ()
# c;;
- : complex = {re = 2.; im = 3.}
# c = {im=3.;re=3.} ;;
- : bool = true
```

Zapisi v Ocaml

- Operacije:
  - Dostop do komponent
  - Ujemanje vzorcev

expr.name

{ $name_i = p_i$; . . . ; $name_j = p_j$ }

```
# 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}
```

Primer v Ocaml

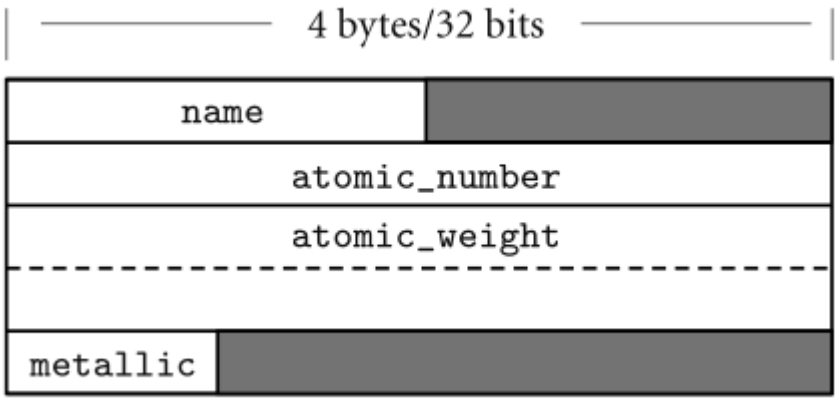
```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
# p.xc <- 3.0 ;;
- : unit = ()
```

```
# let moveto p dx dy =
    begin
      p.xc <- p.xc +. dx;
      p.yc <- p.yc +. dy;
    end;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

Postavitev spomina za zapise

- Polja zapisa so običajno shranjena na zaporednih lokacijah v spominu.
- Prevajalnik si zapomni odmik vseh polj znotraj zapisa.
- Model vrednosti (sprem.)
  - Vsebina polj je vgnezdena v nadrejeni zapis.
- Model referenc
  - Polja so kazalci na podatke na neki drugi lokaciji v spominu.

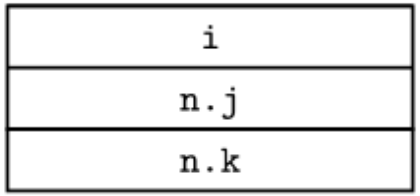
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```



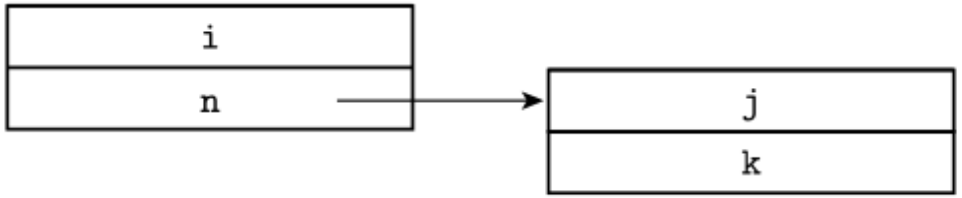
Postavitev spomina za zapise

- Postavitev spomina za vgnezdene strukture (struct, class) v C in Javi (spodaj).

```
struct T {
    int j;
    int k;
};
struct S {
    int i;
    struct T n;
};
```




```
class T {
    public int j;
    public int k;
}
class S {
    public int i;
    public T n;
}
```



Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Kazalci

- Kazalec je referenca na objekt v spominu
  - Včasih jih imenujejo reference
  - Kazalec je običajno predstavljen s pomnilniškim naslovom
  - Kazalec ima lahko tip (ML, C++)
    - PJ ve na kaj kaže kazalec
    - S tipom je znana struktura in velikost objekta
  - Kazalec je lahko naslov brez tipa (Lisp, C)
    - Programer mora vedeti kam kaže kazalec
    - Prevajalnik ne pozna strukture objekta torej ne more preveriti pravilnost dostopa, interpretacijo, itd.

Kazalci in rekurzivne strukture

- Rekurzivna podatkovna struktura vsebuje vsaj eno referenco na objekt istega tipa.
  - Rekurzivno podat. strukturo lahko definiramo z uporabo struktur, ki vsebujejo komponente
    - Zapisi, n-terice, polja, seznami ali unije.
- Jeziki z modelom referenc
  - Komponente vsebujejo kazalce na druge objekte
  - Ni potrebno definirati kazalcev (definirani implicitno)
- Jeziki z modelom vrednosti
  - Komponente morajo vsebovati kazalce na objekte istega tipa in ne vrednosti objektov.

Kazalci

- Vrste referenciranih lokacij
  - V nekaterih jezikih lahko kazalci kažejo samo na objekte v kopici (Java, Pascal, Ada, Modula)
    - Objekt kreiramo z operacijo `new( )`, ki vrne kazalec
    - To je edini način za kreiranje kazalca
  - Drugi jeziki uporabljajo kazalce, ki lahko kažejo na poljubno lokacijo (Algol,C,C++, itd.)
    - Ti jeziki uporabljajo operator “&” (vrni-naslov)
- Sproščanje zaseženih objektov
  - Nekateri jeziki z operatorjem new() morajo eksplicitno sprostiti zasežen objekt (C,C++,Pascal)
    - Možne napake: dostop do sproščenega objekta, puščanje spomina
  - Dugi uporabljajo automatsko čiščenje spomina (Java,C#)

Kazalci

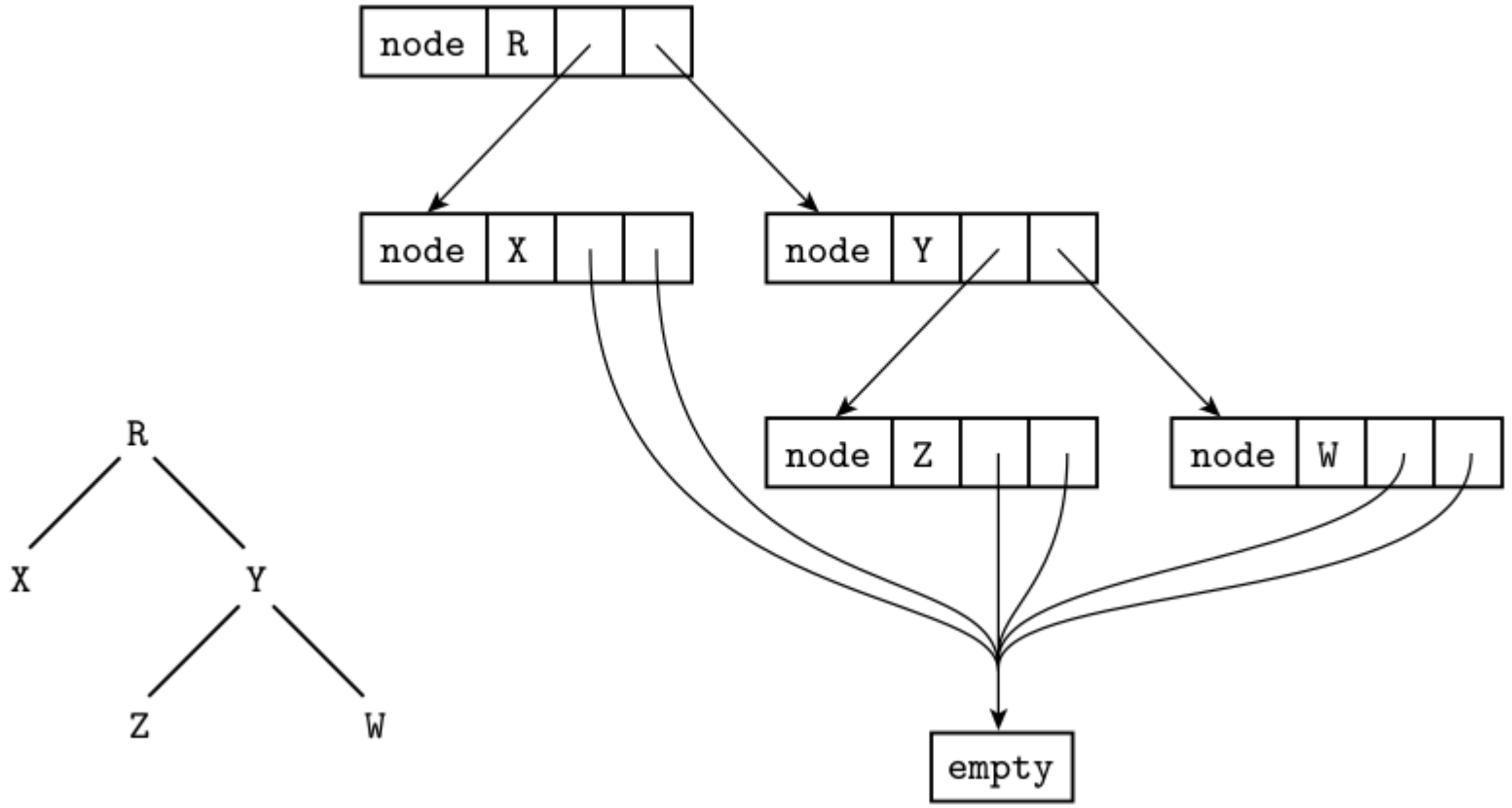
- Operacije na kazalcih so odvisne od modela spremenljivk
  - Zaseganje in sproščanje objektov na kopici
  - De-referenciranje kazalca za dostop do komp. objekta
  - Prirejanje kazalca drugemu kazalcu
- Funkcijski jeziki (model referenc)
  - Automatično zaseganje in sproščanje objektov
- Imperativni jeziki (model vrednosti in/ali referenc)
  - Eksplicitno delo z zaseganjem in sproščanjem (C,C++)
  - Java in C# sta nekje vmes: model vrednosti in referenc

Model referenc

- Rekurzivne podatkovne strukture vsebujejo kazalce na strukture istega tipa
- Primer rekurzivne podatkovne strukture v Ocaml
  - ML implementira reference na strukture (n-terke, zapisi, unije, seznami)

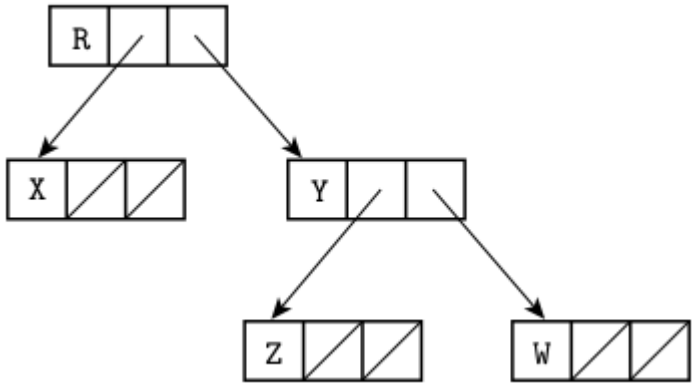
```
# type chr_tree = Empty | Node of char * chr_tree * chr_tree;;
type chr_tree = Empty | Node of char * chr_tree * chr_tree
# let t = Node('R',Node(‘X’,Empty,Empty),
                        Node(‘Y’,Node(‘Z’,Empty,Empty),
                                 Node(‘W’,Empty,Empty)))
```

Rekurzivni tipi v OCaml



Model vrednosti in kazalci

- Rekurzivne strukture s kazalci
  - C, C++
- Primer v Pascalu in C




```
type chr_tree_ptr = ^chr_tree;
     chr_tree = record
        left, right : chr_tree_ptr;
        val : char
     end;

new(my_ptr);
```

```
struct chr_tree {
    struct chr_tree *left, *right;
    char val;
};

my_ptr = malloc(sizeof(struct chr_tree));
```

Primer: implementacija seznamov v Ocaml

```
# 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 rnode
```

```
# let l1 = Elm {cont = 1; next = Elm {cont = 2; next = Nil}};;
val l1 : int rlist =  Elm {cont = 1; next = Elm {cont = 2; next = Nil}}
# let cons v l = Elm {cont=v; next=l};;
val cons : 'a -> 'a rlist -> 'a rlist = <fun>
```

```
# let ( ** ) v l = cons v l;;
val ( ** ) : 'a -> 'a rlist -> 'a rlist = <fun>
# let l2 = cons 3 (cons 4 Nil));;
val l2 : int rlist =  Elm {cont = 3; next = Elm {cont = 4; next = Nil}}
# let l3 = 5**6**Nil;;
val l3 : int rlist = Elm {cont = 5; next = Elm {cont = 6; next = Nil}}
```

Primer: implementacija seznamov v Ocaml

```
# exception EmptyList;;
exception EmptyList
# let head l = match l with Nil -> raise EmptyList | Elm r -> r.cont;;
val head : 'a rlist -> 'a = <fun>
# let tail l = match l with Nil -> raise EmptyList | Elm r -> r.next;;
val tail : 'a rlist -> 'a rlist = <fun>
# head l1;;
- : int = 1
# tail l1;;
- : int rlist = Elm {cont = 2; next = Nil}
```

```
# let rec length l = match l with Nil -> 0 | Elm {next=t} -> 1+length t;;
val length : 'a rlist -> int = <fun>
# length l1;;
- : int = 2
```

Primer: implementacija seznamov v Ocaml

```
# let rec append l1 l2 = match l1,l2 with
  Elm r1,_ -> Elm {cont=r1.cont; next=append r1.next l2}
| Nil,Elm r2 -> Elm {cont=r2.cont; next=append Nil r2.next}
| Nil,Nil -> Nil;;
val append : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append l1 l2;;
- : int rlist =
Elm {cont=1; next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist = Elm {cont=1;  next=Elm {cont=2; next=Nil }}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil }}
```

Primer: implementacija seznamov v Ocaml

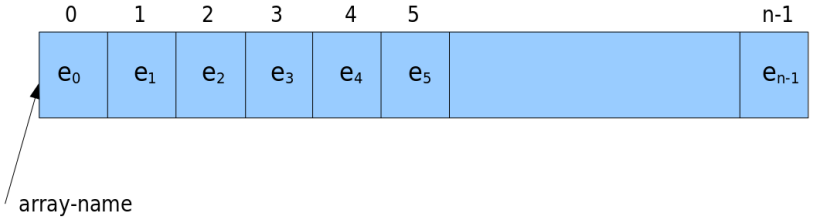
```
# let rec append1 l1 l2 = match l1 with
  Nil -> l2
| Elm r when r.next=Nil -> r.next <- l2; l1
| Elm r -> ignore (append1 r.next l2); l1;;
val append1 : 'a rlist -> 'a rlist -> 'a rlist = <fun>
```

```
# append1 l1 l2;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l1;;
- : int rlist =
Elm {cont=1;  next=Elm {cont=2; next=Elm {cont=3; next=Elm {cont=4; next=Nil}}}}
# l2;;
- : int rlist = Elm {cont=3;next=Elm {cont=4;next=Nil}}
```

Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Polja



- Poje je podatkovna struktura, ki hrani končno število elementov določenega tipa
- Polje definira preslikavo med indeksnim tipom in tipom elementa polja.
  - Indeks je običajno celo število; precej PJ uporablja diskretni tip
- Polje je ena izmed bolj pomembnih podatkovnih struktur imperativnih prog. jezikov
  - C, C++, Java, Fortran, Pascal, ...
  - Podobna vloga v imp. PJ kot jo imajo seznami v fun. PJ
- Elementi polja so spremenljivi
  - Velikost je fiksna (lahko ga ponovno zasežemo).

Polja

- Dostop do elementov polja
  - V večini PJ se uporablja ime polja, ki mu sledi indeks elementa znotraj neke oblike oklepajev (a(), a[], a{}, ...)
  - Indeksi polja so običajno celo število lahko pa je indeks poljuben naštevni (diskretni) tip.
- Definicija polja
  - Indeksi so v večini prog. jezikov definirani z območjem
  - Indeksi v C se začnejo z 0

```
char[] upper;          /* Java */
char upper[];          /* alternative declaration */
upper = new char[26];
char upper[26];      /* C */
character, dimension (1:26) :: upper  /* Fortran */
character (26) upper     /* shorthand notation */
var upper : array ['a'..'z'] of char;    /* Pascal */
```

Dimenzije, meje in alokacija

- V prejšnjih primerih je oblika polja (vključno z mejami) specificirana pri deklaraciji.
- Pomnilnik za polja statične oblike se upravlja na običajen način.
  - Statična alokacija polj katerih življenska doba je celoten program.
  - Alokacija na skladu za polja katerih življenska doba je vezana na podprogram.
  - Alokacija v kopici za bolj splošno življensko dobo.
- Delo s spominom je bolj zapleteno za bolj kompleksna polja
  - Katerih oblika ni znana do časa kreacije (deklaracije).
  - Katerih oblika se lahko spremeni med izvajanjem.

Dimenzije, meje in alokacija

- Dinamična polja:
  - C++ (vectors), Java, C# (ArrayList), Go (slices), Rust (Vec)
- Za dinamična polja mora prevajalnik
  - Alocirati prostor in omogočiti podatke o obliki v času izvajanja
  - Nekateri PJ dovolijo, da so število in meje dimenzij dinamični, drugi dovolijo samo dinamične meje.
- Alokacija dinamičnih polj
  - Lokalno polje je še vedno lahko na skladu.
    - Oblika je znana ob kreaciji
  - Polje katerega velikost se lahko spremeni se alocira na kopici.
- Opisni (dope) vektorji vsebujejo podatke o obliki polja v času izvajanja
  - Odmiki polj zapisov, spodnja meja, velikost in zgornja meja za vsako dimenzijo.
  - Opisni vektor se lahko shrani v aktoivacijski zapis na skladu ali skupaj s poljem v kopici.

Postavite polj v spominu

- Polja so v večini PJ shranjena na zaporednih lokacijah v spominu.
  - Eno-dimenzionalno polje: element za elementom
  - Več-dimenzionalno polje: po vrsticah/po stolpcih
    - Vgnezdene zanke z dostopom do vseh elementov več-dimenzionalnega polja.
    - Hitrost zank je zelo odvisna od učinkovitega predpomnenja
  - Prava več-dimen. polja so shranjena na zveznih lokacijah

```
for (i = 0; i < N; i++) {       /* rows */
    for (j = 0; j < N; j++) {   /* columns */
        ... A[i][j] ...
    }
}
```

- Postavitev kazalec-za-vrstico
  - Postavitev ni zvezna; bloki z 1D polji
  - Prednosti: vrstice razl. dolžine, inicializirano iz delov
  - C, Ocaml, Java, C# (nekateri uporabljajo obe postavitve)

Polja v OCaml

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

- Kreiranje polja:
  elemente lahko naštejemo med [|...|]
- Polja so integrirana v Ocaml
  - Niso tako zelo povezana z Ocaml kot seznami
- Podobno seznamom, imamo na razpolago modul Array, ki vsebuje vse potrebne operacije.
- Kreiranje polja
- Dostop/popravek elementa polja
  - Dostop do elementa
  - Prirejanje nove vrednosti elementu

```
# let v = Array.create 3 3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

$expr_1$.($expr_2$)

$expr_1$.($expr_2$) <- $expr_3$

Polja v OCaml

- Primer:

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

- Indeks polja ne sme uiti preko meja

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

- Preverjanje mej indeksa je drago
  - Nekateri jeziki privzeto ne preverjajo indeksov (npr. C)

Funkcije na poljih

```
# let n = 10;
val n : int = 10
# let v = Array.create n 0;;
val v:int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

```
# 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 reverse v =
      let tmp=ref 0
      and n = Array.length(v)
      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 = ()
# reverse(v);;
- : int array = [|9; 8; 7; 6; 5; 4; 3; 2; 1; 0|]
```

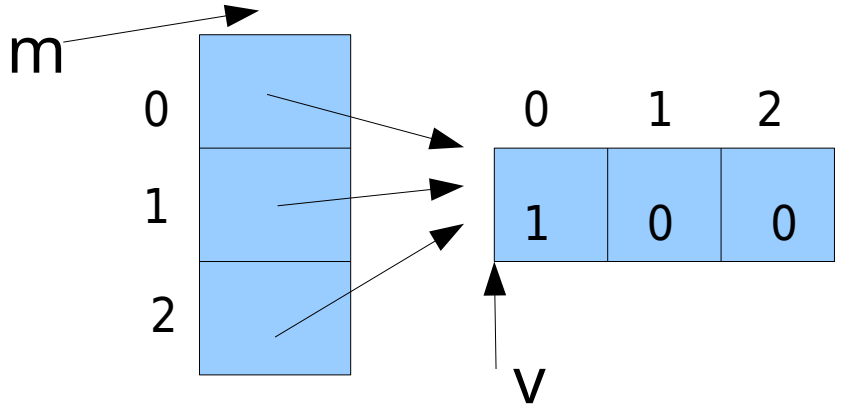
```
# let u = [|2;3|];;
val u : int array = [|2; 3|]
# let m = 2;;
val m : int = 2
# let subarray u v =
      let found = ref false
      and i = ref 0
      in while ((!i<=(n-m)) && not !found) do
            found := true;
            for j=0 to (m-1) do
                if v.(!i+j) != u.(j) then
                   found := false
            done;
            i := !i+1
         done;
      !found;;
val subarray : 'a array -> 'a array -> bool = <fun>
# subarray u v;;
- : bool = true
```

Primer: subarray()

```
# let prefix u v i =
    let found = ref true
    and m = Array.length(u)
    in for j=0 to (m-1) do
          if v.(i+j) != u.(j) then
             found := false
          done;
       !found;;
val prefix : 'a array -> 'a array -> int ->
                  bool = <fun>
# prefix u v 0;;
- : bool = false
# prefix u v 2;;
- : bool = true
```

```
# let subarray u v =
    let found = ref false
    and i = ref 0
    and m = Array.length(u)
    and n = Array.length(v)
    in while ((!i<=(n-m)) && not !found) do
          found := prefix u v !i;
          i  := !i+1
       done;
    !found;;
val subarray : 'a array -> 'a array ->
                     bool = <fun>
```

Implementacija polj v OCaml




```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
val m : int array array =
        [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array =
    [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

Matrike v Ocaml

```
# let m = Array.create_matrix 4 4 0;;
val m : int array array = [|[|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]; [|0; 0; 0; 0|]|]
# for i=0 to 3 do m.(i).(i) <- 1; done;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0; 0|]; [|0; 1; 0; 0|]; [|0; 0; 1; 0|]; [|0; 0; 0; 1|]|]
# m.(1);;
- : int array = [|0; 1; 0; 0|]
```

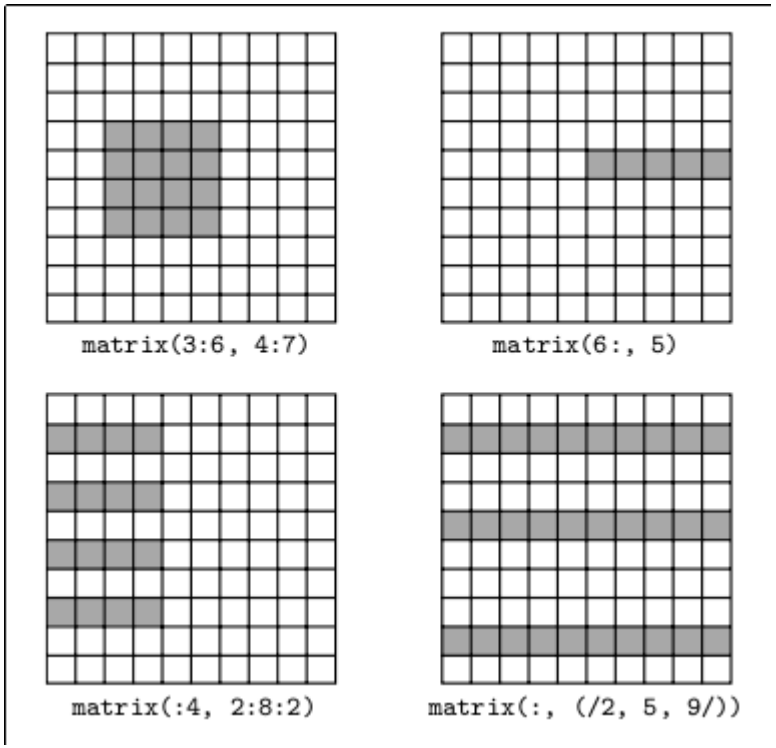
Operacije na matrikah

```
# 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>
# a.(0).(0) <- 1.0; a.(1).(1) <- 2.0; a.(2).(2) <- 3.0;;
- : unit = ()
# b.(0).(2) <- 1.0; b.(1).(1) <- 2.0; b.(2).(0) <- 3.0;;
- : unit = ()
# add_mat a b;;
- : float array array = [|[|1.; 0.; 1.|]; [|0.; 4.; 0.|]; [|3.; 0.; 3.|]|]
```

Matrike

- Več-dimenzionalna polja
  - Definicija
  - Polje polj
    - C, C++,ML,Java, ...
  - Dvo-dimenzionalno polje
    - Ada, Fortran, ...
- Rezine (angl. slices)
  - Rezina je pravokotno področje v polju
  - R, Fortran, Python

```
/* C */
double mat[10][10];
/* Ocaml */
type 'a matrix = array array 'a;;
/* Modula-3 */
VAR mat : ARRAY [1..10] OF ARRAY [1..10] OF REAL;
/* Ada */
mat1 : array (1..10, 1..10) of real;
```



Potek

- Uvod
- Spomin in spremenljivke
- Sekvence, vejitve in bloki
- Zanke
- Procedure in funkcije
- Zapisi
- Kazalci
- Polja
- Množice, unije, slovarji

Množice

- Množica shranjuje vrednosti brez uporabe urejenosti
- Osnovne operacije
  - Osnovne: create, delete, add_element, delete_element
  - Boolove: membership, subset, equality, disjoint
  - Algebra: union, difference, intersection
- Implementacija
  - Obstaja več različnih načinov implementacije množic
    - Vsaka ima slabe lastnosti v nekaterih primerih
    - Za vsak poseben namen ni težko implementirati množice z obstoječimi podatkovnimi strukturami

Množice

- Implementacija
  - Seznami polja (neučinkovito)
  - Bitni nizi (prostorsko učinkovito, inštrukcije procesorja)
  - Binarna iskalna drevesa (knjižnica: Ocaml, Haskell)
  - Razpršilne tabele
  - Predstavitev množic s slovarji
- Množice v programskih jezikih
  - Knjižnice: C++, Java, .NET, Ruby, Ocaml, Swift, Erlang
  - Gradnik: Javascript, Python, Pascal

Množice v Python

- Primeri:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)

thisset.add("orange")
thisset.remove("banana")
thisset.discard("banana")  # not ∃ -> no error
del thisset                # delete complete set

set2 = {1, 2, 3}
set3 = set1.union(set2)    # {3, 'b', 'a', 2, 1, 'c'}
```

Množice v Python

- add() Adds an element to the set
- remove() Removes the specified element
- discard() Remove the specified item
- pop() Removes an element from the set
- clear() Removes all the elements from the set
- copy() Returns a copy of the set
- union() Return a set containing the union of sets
- intersection() Returns a set, that is the intersection of two other sets
- difference() Returns a set contains the difference betw two or more sets
- isdisjoint() Returns whether two sets have a intersection or not
- Issubset() Returns whether another set contains this set or not
- issuperset() Returns whether this set contains another set or not
-
- … and more

Unije

- Tip konstruiran z unijo
  - Nov tip je narejen z unijo obstoječih tipov
- Unija v Ocaml
  - Definicija tipa
  - Konstrukcija instance
  - Ujemanje vzorcev
- Unija v drugih jezikih
  - Označena unija
    - ML-družina, Haskell
    - Pascal, Ada, Modula2
    - Drugo ime: spremenljivi zapisi (variant rec.) v Pascal
  - Neoznačena unija: C, C++

type name = ...
| $Name_i$ ...
| $Name_j$ of t_j ...
| $Name_k$ of t_k * ...* t_l ... ;;

Unije v C

- Tip omogoča shranjevanje več različnih vrednosti v isti pom. prostor
  - velikost = velikost največje komponente

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

```
union Data {
   int i;
   float f;
   char str[20];
} data;
```

```
int main( ) {
   union Data data;
   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");
   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);
   return 0;
}
```

Spremenljivi zapisi v Pascal

- Deli zapisov so spremenljivi
  - Oznaka tipa razloči med variantami
  - Zapisi morajo vsebovati največjo verzijo

```
type paytype=(salaried, hourly);
var employee: record
        id: integer;
        dept: array [1...3] of char;
        age: integer;
        case payclass: paytype of
            salaried: (monthlyrate: real; stardate: integer);
            hourly (hourrate: real; reghours: integer; overtime: integer);
    end;
```

Slovarji

- Alternativna imena
  - Asociativno polje; map; tabela simbolov
- Shramba parov ključ/vrednost
  - Ključi in vrednosti so poljubnega tipa
  - Definirane operacije
    - Create, access, update, delete, to-list, keys, ...
- Programski jeziki
  - Začetne implementacije
    - TMG (1965, compiler-compiler), SETL (1960s), Snobol (1969)
  - Skriptni jeziki
    - AWK, Rexx, Perl, PHP, Tcl, JavaScript, Python, Ruby, Go, Lua

Slovarji

  - Ostali jeziki
    - C++, Java, Scala, Erlang, OCaml, Haskell
- Implementacija slovarja
  - Razpršilna tabela
    - `O(1)`
  - Iskalno drevo
    - Binarno iskalno drevo, B+-drevo, ...
- Zelo popularna in uporabna podatkovna struktura
  - Lahko predstavi katerokoli drugo podatkovno strukturo

Operacije nad slovarji v Python

- Kreacija
  - `D = {}, D = {'key1':value1,'key2':value2, … }`
  - `dict(name1=value1,name2=value2, …)`
  - Iz seznama: parov, imen, ...
- Dostop s ključem
  - `D['name'], D['name1']['name2'], 'name' in D`
  - `D.get(key)`
- Popravljanje in brisanje ključa
  - `D.update(D1)`
  - `del D[key], D.pop(key)`
- Branje ključev, vrednosti in parov kluč/vrednost
  - `D.keys(), D.values(), D.items()`