Predavanje 12

Moduli

Iztok Savnik, FAMNIT

May, 2024.

Literatura

- Emmanuel Chailloux, Pascal Manoury, Bruno Pagano, Developing Applications With Objective Caml, O'REILLY & Associates, 2000 (Chapter 14)
- John Mitchell, Concepts in Programming Languages, Cambridge Univ Press, 2003 (Chapters 9)
- Michael L. Scott, Programming Language Pragmatics (3rd ed.), Elsevier, 2009 (Chapter 9)
- Iztok Savnik, Koncepti programskih jezikov, Skripta (in Slovene), FAMNIT, 2013 (Chapter 6)

Pregled

1. Koncept modula
2. Vrste modulov
3. Moduli v C
4. Moduli v Javi
5. Moduli v Ocaml
6. Jezik modulov v Ocaml
7. Funktorji

Koncept modula

- Modularno načrtovanje razdeli program v več programskih enot, ki jih imenujemo moduli
- Modul se lahko razvije <u>neodvisno</u> od drugih delov sistema
  - Module lahko prevedemo ločeno
  - Programer ne potrebuje izvorno kodo, da bi lahko delal z modulom
- <u>Vmesnik modula</u> definira vrednosti, tipe, funkcije, razrede ali vmesnike, ki jih modul ponuja uporabniku
  - Vmesnik skriva podrobnosti implementacije
  - Vse kar programer potrebuje vedeti je definirano v vmesniku

Moduli

- Moduli so definirani za izbrano “entiteto”
  - Primeri entitet:
    - fizična naprava, meni uporabniškega vmesnika, podatkovna struktura, funkcijska enota aplikacije, …
- Modul definira podatkovno okolje
  - Množico podatkovnih struktur za modeliranje entitete
  - Običajno ena podatkovna struktura predstavlja modul
  - Kasneje to strukturo definiramo kot APT (Abstraktni Podatkovni Tip)
- Modul definira množico operacij
  - Operacije za delo z entiteto
  - Operacije so običajno definirane na primerku APT

Moduli

- Razvijalec modula ima precej svobode pri implementacije modula
  - Implementacija se lahko popolnoma spremeni medtem, ko vmesnik ostane isti
    - Uporabnik modula lahko niti ne opazi razlike
  - Programer ne potrebuje vedeti podrobnosti o ostalih delih sistema v razvoju, da bi lahko implementiral svoj modul
    - Zadosti je da pozna vmesnike preostalih modulov
  - Vmesnik modula skrije implementacijske podrobnosti ki jih programer noče deliti z uporabniki

Pregled

1. Koncept modula
2. <u>Vrste modulov</u>
3. Moduli v C
4. Moduli v Javi
5. Moduli v Ocaml
6. Jezik modulov v Ocaml
7. Funktorji

Modularni jeziki: pregled

- Enote prevajanja
  - Izolacija kode v eno samo konceptualno enoto
  - Konceptualno šibek modularen jezik
- Jezik modulov
  - Jezik modulov je del programskega jezika
  - Vsebuje programske konstrukte in koncepte potrebne za definicjo vmesnika in implementacije modula
- Funktorji
  - Parametrizirani moduli
  - Moduli so lahko parametri modulov
  - Generična koda

Modul kot enota prevajanja

- Več programskih jezikov uporablja module kot enote prevajanja
  - C, C++, Java, Scala, Erlang, ML, Ocaml, Pascal, Modula, Perl, Python
- Modul je običajno predstavljen z eno ali dvema datotekama (lahko tudi z več kot dvemi)
  - C, C++: glave in implementacijske datoteke (m.h, m.c)
  - Java, Scala paketi in moduli:
    - paket=razred (datoteka) v direktorijih; modul=množica paketov
  - ML, Ocaml: dve datoteki (ocaml) ali jezik modulov
  - Erlang: ena ali več datotek
  - Perl, Python: različne datoteke

Abstraktni podatkovni tipi

- Modul je Abstraktni Podatkovni Tip (APT)
  - OCaml, ML, Haskell, Schema, Scala, Kotlin, Swift, Julija, Erlang, Rust, itd.
  - APT je <u>del funkcijskega programskega jezika</u>
- Kaj je abstraktni podatkovni tip?
  - Podatkovni tip vsebuje podatke in operacije
    - APT predstavlja množico abstraktnih podatkovnih <u>struktur</u>
    - Imamo množico <u>operacij</u>, ki so definirane nad primerki APT
  - APT je abstrakten:
    - Vmesnik (signatura) definira pogled neodvisen od implementcije
      - Signatura je <u>tip modula</u>: tipi operacij nad primerki APT
    - Interna predstavitev APT je <u>uporabniku skrita</u>
      - Moduli so podobni razredom v OO jezikih
  - <u>Podatkovna abstrakcija</u> v funkcijskih jezikih

Funktorji

- Parametrizirani moduli
  - Parametri modulov so manjši moduli
  - Parameter funktorja je podan s tipom modula (signatura)
    - Modul (parameter) uporabljamo ne da bi poznali implementacijo
  - Moduli podani kot parametri
    - Modelirajo podatkovne strukture, ki so sestavni deli abstraktnih podatkovnih tipov (npr. kolekcija in tip elementa)
    - Omogočajo dostop do različnih naprav preko modulov (parametrov)
- Generativnost v funkcijskih jezikih
  - Podobno parametrične polimofizmu nad funkcijami
  - Funktor apliciran na modulih, ki služijo kot parametri, vrne konkreten modul!

Pregled

1. Koncept modula
2. Vrste modulov
3. <u>Moduli v C</u>
4. Moduli v Javi
5. Moduli v Ocaml
6. Jezik modulov v Ocaml
7. Funktorji

Moduli v C

- Programska koda je razdeljena v več različnih datotek.
  - Datoteke s podaljškom *.c vsebujejo dele programa
  - Datoteke s podaljškom *.h vsebujejo vmesnike modula
- Prednosti uporabe C modulov
  - Program je razdeljen v logično smiselne komponente
  - Ločeno prevajanje komponent v ločene objektne datoteke (kasneje povezane v en sam program)
  - Modul je računska komponenta, ki predstavlja model nekega dobro definiranega koncepta
    - Podatkovna struktura (npr. sklad), fizična naprava (npr. zaslon, pomnilniška naprava, gonilnik), itd.

C modul:

set.h
set.c

```
#ifndef SET_H
#define SET_H
/* Constants */
#define SET_LEN  sizeof(set_type) /* Maximal length of a set */
#define SET_MASK 0xffffffff   /* Mask for computing set op. */
#define MaxSetEl 30           /* max number of set elements. */
/* SET type definition */
typedef struct set_type {
  unsigned long lo;
  unsigned long hi;
} set_type;
/*---------------------- Exported functions -----------------------------*/
extern set_type* set_emp( set_type *S );
extern set_type* set_cpy( set_type *S, set_type *S1 );
extern set_type* set_union( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_intsc( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_diff( set_type *S, set_type *S1, set_type *S2 );
extern set_type* set_add( set_type *S, int el );
extern set_type* set_del( set_type *S, int el );
extern boolean   set_elm( int el, set_type *S );
extern boolean   set_subs( set_type *S1, set_type *S2 );
extern boolean   set_equ( set_type *S1, set_type *S2 );
extern int       set_card( set_type *S );
extern boolean   set_next_el( set_type *S, int cEl, int *nEl );
extern void      set_print( set_type *S );
#endif  /* SET_H */
```

```
#include <stdio.h>
#include "config.h"
#include "set.h"

/* Make set S empty */
set_type* set_emp( set_type *S )
{
  (*S).lo = 0; (*S).hi = 0;
  return S;
}/*set_emp*/

/* S = S1; */
set_type* set_cpy( set_type *S, set_type *S1 )
{
  (*S).lo = (*S1).lo;
  (*S).hi = (*S1).hi;
  return S;
}/*set_null*/

/* S = S1 + S2; */
set_type* set_union( set_type *S, set_type *S1, set_type *S2 )
{
  (*S).lo = (*S1).lo | (*S2).lo;
  (*S).hi = (*S1).hi | (*S2).hi;
  return S;
}
```

```
...

/* S = S – {el}; */
set_type* set_del( set_type *S, int el )
{
  if (el < 32) (*S).lo = (*S).lo & ((1 << el) ^ SET_MASK);
  Else (*S).hi = (*S).hi & ((1 << (el-32)) ^ SET_MASK);
  return S;
}

/* Membership test. */
boolean set_elm( int el, set_type *S )
{
  if (el < 32) return (( (*S).lo & (1 << el)) > 0);
  else return (( (*S).hi & (1 << (el-32))) > 0);
}

/* Subsumption test. */
boolean set_subs( set_type *S1, set_type *S2 )
{
  boolean Lo,Hi;
  Lo = (( (*S1).lo & (*S2).lo ) == (*S1).lo );
  Hi = (( (*S1).hi & (*S2).hi ) == (*S1).hi );
  return (Lo && Hi);
}
```

Pregled

1. Koncept modula
2. Vrste modulov
3. Moduli v C
4. Moduli v Javi
5. Moduli v Ocaml
6. Jezik modulov v Ocaml
7. Funktorji

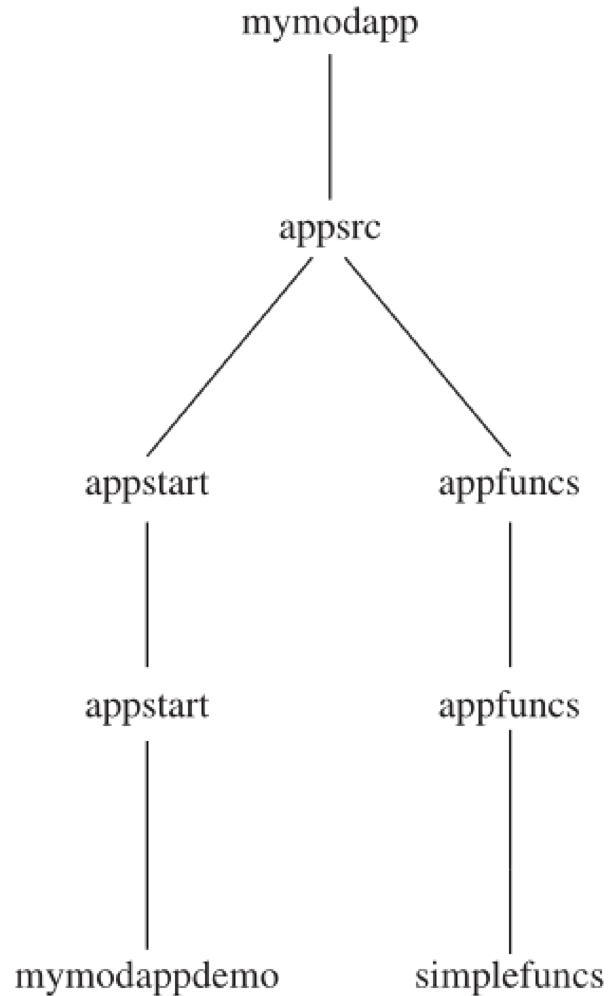
Moduli v Javi

- Programi so organizirani v množice paketov
  - Člani paketa so razredi in vmesniki
  - Paket lahko vsebuje pod-pakete (rekurzivno)
- Vsak paket ima svoj nabor imen razredov in vmesnikov
  - Struktura imenovanja paketov je hierarhična
- Če je množica paketov zadosti kohezivna potem lahko pakete združimo v modul
  - Modul lahko izvozi nekatere ali vse pakete
  - Modul je lahko odvisen (eksplicitno) od drugega modula
    - Lahko uporablja pakete drugega modula

Moduli v Javi

- Moduli omogočajo opisa relacij in odvisnosti v kodi, ki sestavlja aplikacijo
  - Deklaracija modula (module-info.java in direktorij modula)
- Moduli kontrolirajo kako paketi modula uporabljajo druge module
  - Odvisnost med moduli opišemo z `requires`
- Moduli kontrolirajo kako drugi moduli uporabljajo njihove pakete
  - Določijo kateri paketi bodo izvoženi z uporabo `exports`

Example: Java modules



SimpleMathFuncs.java

appsrc\appfuncs\appfuncs\simplefuncs

module-info.java

```
// Module definition for the functions module.
module appfuncs {
  // Exports the package appfuncs.simplefuncs.
  exports appfuncs.simplefuncs;
}
```

appsrc\appfuncs

```
// Some simple math functions.

package appfuncs.simplefuncs;

public class SimpleMathFuncs {

  // Determine if a is a factor of b.
  public static boolean isFactor(int a, int b) {
    if((b%a) == 0) return true;
    return false;
  }

  // Return the smallest positive factor that a and b have in common.
  public static int lcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = 2; i <= min/2; i++) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }
    return 1;
  }

  // Return the largest positive factor that a and b have in common.
  public static int gcf(int a, int b) {
    // Factor using positive values.
    a = Math.abs(a);
    b = Math.abs(b);

    int min = a < b ? a : b;

    for(int i = min/2; i >= 2; i--) {
      if(isFactor(i, a) && isFactor(i, b))
        return i;
    }

    return 1;
  }
}
```

MyModAppDemo.java

appsrc\appstart\appstart\mymodappdemo

module-info.java

```
// Module definition for the main application module.
module appstart {
  // Requires the module appfuncs.
  requires appfuncs;
}
```

appsrc\appstart

```
// Demonstrate a simple module-based application.
package appstart.mymodappdemo;

import appfuncs.simplefuncs.SimpleMathFuncs;

public class MyModAppDemo {
  public static void main(String[] args) {

    if(SimpleMathFuncs.isFactor(2, 10))
      System.out.println("2 is a factor of 10");

    System.out.println("Smallest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.lcf(35, 105));

    System.out.println("Largest factor common to both 35 and 105 is " +
                       SimpleMathFuncs.gcf(35, 105));

  }
}
```

Pregled

1. Koncept modula
2. Vrste modulov
3. Moduli v C
4. Moduli v Javi
5. <u>Moduli v Ocaml</u>
6. Jezik modulov v Ocaml
7. Funktorji

Moduli v Ocaml

- Osnovna oblika modulov v Ocaml je podobna modulom v program. jeziku C
  - Datoteka s kodo ima končnico “.ml”
  - Vmesnik je datoteka s podaljškom “.mli”
  - Podobno C, modul je lahko ena sama datoteka
- Prvi primer
  - Modul je shranjen v datoteki Stack1.ml
  - Implementacija sklada osnovana na seznamih
  - To je del standardne knjižnice

```
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
```

Moduli v Ocaml

- Sklad v prejšnjem primeru je definiran kot abstraktni podatkovni tip
  - Modul definira abstraktno strukturo Stack1.t, ki je center modula
  - Prvo kreiramo Stack1.t, ki jo potem podajamo kot parameter vsak funkciji definirani v vmesniku modula
  - APT ima veliko skupnega z razredi v OO svetu
- Primer uporabe modula Stack1
  - Najprej se kreira sklad
  - Uporaba funkcij modula
  - Koda je shranjena v datoteki Example1.ml

```
let s = Stack1.create ();;
Stack1.push 1 s; Stack1.push 2 s;
Stack1.push 3 s;;
let a = Stack1.pop s
and b = Stack1.pop s
and c = Stack1.pop s
in Printf.printf "Stack elements: %i, %i, %i\n" a b c;;
```

Moduli v Ocaml

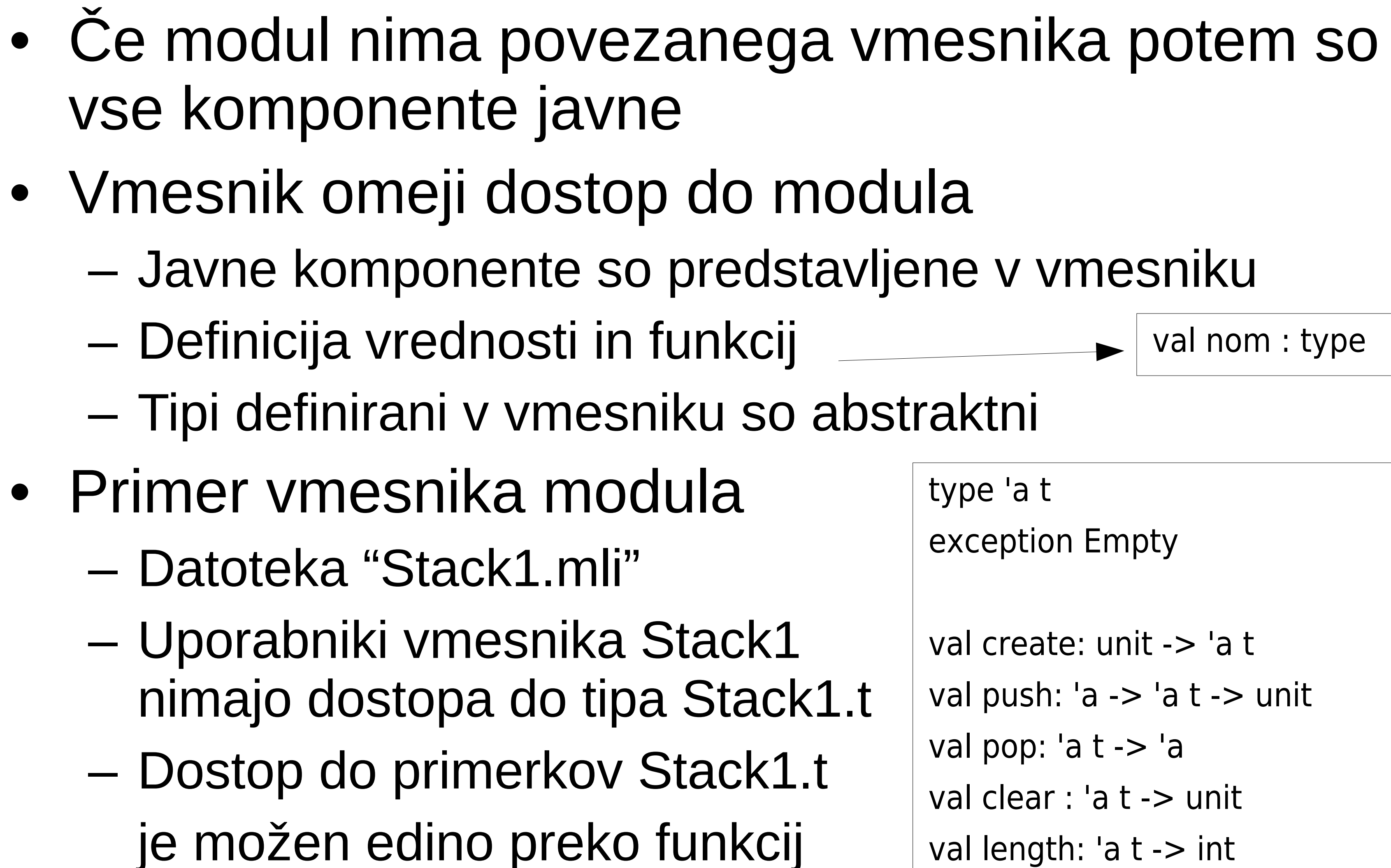
- Dostop do komponent modula
  - Do komponent lahko dostopamo z uporabo notacije s piko
    - Module.identifier
  - Komponente modula lahko uvozimo v neko drugo okolje
    - open Module;;
- Prevajanje
  - Program lahko prevedemo s prevajalnikom ocamlc

```
open Stack1;;
let s = create ();;
push 1 s; push 2 s; push 3 s;;
let a = pop s and b = pop s
and c = pop s
in Printf.printf "Stack elements: \
                    %i, %i, %i\n" a b c;;
```

s

```
$ ocamlc -o example1 Stack1.ml Example1.ml
$ ./example1
elements: 3, 2 and 1
```

Vmesnik modula



Povezovanje modulov in vmesnikov

- Modul je sestavljen iz dveh delov
  - Implementacije, ki vsebuje definicije tipov, spremenljivk in funkcij
  - Vmesnika, ki vsebuje deklaracije definiranih komponent vidnih od zunaj
    - Vmesnik lahko deklarira podmnožico definicij komponent!
    - Modul lahko vsebuje dodatne (pomožne) tipe, vredn. in funkcije
    - Te funkcije niso dostopne izven modula
- Deklaracije morajo biti konsistentne z definicijami
  - Vmesnik lahko omeji tipe komponent
- Vmesnik je ločen od implementacije
  - Lahko imamo več implementacij kot tudi več vmesnikov

Povezovanje modulov in vmesnikov

- Modul je shranjen v Stack2.ml
- Implementacija sklada s poljem
- Vmesnik lahko ostane isti kot za impl. s seznamom
- Uporabnik modula ne rabi vedeti, da se je implementacija spremenila

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [||] }
let clear s = s.sp <- 0; s.c <- [||]
let size = 5
let increase s v =
  s.c <- Array.append s.c (Array.make size v)
let push x s =
  if s.sp >= Array.length s.c then increase s x;
  s.c.(s.sp) <- x;
  s.sp <- s.sp+1
let pop s =
  if s.sp = 0 then raise Empty
  else let () = s.sp <- s.sp-1 in s.c.(s.sp)
let length s = s.sp
let iter f s = for i = s.sp-1 downto 0 do f s.c.(i) done
```

Povezovanje modulov in vmesnikov

- Primer povezovanja enega vmesnika in dveh implementacij
  - Stack1.mli in Stack2.mli sta enaki datoteki
    - Jezik modulov zna ponovno uporabiti Stack1.mli
  - example{1|2}.ml se razlikuje samo v uporabi Stack{1|2}

```
$ ocamlc -o example1 Stack1.mli Stack1.ml example1.ml
$ ./example1
Stack elements: 3, 2, 1
$ ocamlc -o example2 Stack2.mli Stack2.ml example2.ml
$ ./example2
Stack elements: 3, 2, 1
```

Pregled

1. Koncept modula
2. Vrste modulov
3. Moduli v C
4. Moduli v Javi
5. Moduli v Ocaml
6. <u>Jezik modulov v Ocaml</u>
7. Funktorji

Jezik modulov v Ocaml

- Jezik modulov je podmnožica gradnikov programskega jezika, ki se ukvarjajo z moduli
- Vmesnik modula se imenuje <u>signatura</u> in implementacija modula se imenuje <u>struktura</u>
- Sintaksa definicije signatur in struktur modula
  - Ime modula se mora začeti z veliko začetnico

```
module type NAME = sig declarations end
module Name = struct definitions end
```

- Signatura in struktura modula ni potrebno, da je imenovana
  - Anonimna signatura in struktura

```
sig declarations end
struct definitions end
```

Jezik modulov

```
module Name : signature = structure
module Name = (structure : signature)
```

- Definicija modula v Ocaml
  - Struktura = implem. modula
  - Signatura = vmesnik modula
- Vsaka struktura ima privzeto signaturo
  - Deklaracije vseh definicij iz strukture
  - Deklaracije uporabijo najbolj splošen tip
- Postopek za preverjanje tipov preveri signaturo in strukturo
  - Definicije strukture so lahko bolj splošne
  - Signatura lahko omeji privzeto signaturo strukture

Primer modula

- Podatkovna struktura vsebuje par seznamov
- Vrsta + sklad

```
# let q = PairOfLists.create ();;
val q : ('_a list * '_b list) ref = {contents = ([], [])}
# PairOfLists.enqueue 1 q; PairOfLists.push 2 q;;
- : unit = ()
# q;;
- : (int list * '_a list) ref = {contents = ([2; 1], [])}
# PairOfLists.dequeue q;;
- : int = 1
# q;;
- : (int list * int list) ref = {contents = ([], [2])}
# PairOfLists.pop q;;
- : int = 2
```

```
# 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

- Skrivanje abstraktnega tipa modula
- Primer: Sklad
  - Ko je sklad kreiran ga Ocaml označi kot abstrakten
  - Uporabnik Stack1 ne vidi definicije Stack.t
  - Stack.t je skrit s signaturo
- S primerkom sklada se lahko dela samo preko funkcij modula
  - Kot v OO svetu

```
# 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 pogledov na modul

- Omejevanje modulov s signaturami omogoča kreiranje različnih pogledov na eno samo strukturo
  - Na primer, lahko definiramo še en pogled na PairOfLists
  - Isti modul je uporabljen za implementacijo sklada in vrste

```
# 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
```

Prioritetna vrsta

- Element z najnižjo prioriteto je na vrhu
- Funkcija insert zamenjuje vlogi levega in desnega pod-drevesa
  - ... drevo ostane uravnoteženo

```
# module PrioQueue =
  struct
    type priority = int
    type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
    let empty = Empty
    let rec insert queue prio elt =
      match queue with
        Empty -> Node(prio, elt, Empty, Empty)
      | Node(p, e, left, right) ->
          if prio <= p
          then Node(prio, elt, insert right p e, left)
          else Node(p, e, insert right prio elt, left)
    exception Queue_is_empty
    let rec remove_top = function
        Empty -> raise Queue_is_empty
      | Node(prio, elt, left, Empty) -> left
      | Node(prio, elt, Empty, right) -> right
      | Node(prio, elt, (Node(lprio, lelt, _, _) as left),
                        (Node(rprio, relt, _, _) as right)) ->
          if lprio <= rprio
          then Node(lprio, lelt, remove_top left, right)
          else Node(rprio, relt, left, remove_top right)
    let extract = function
        Empty -> raise Queue_is_empty
      | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
  end;;
```

```
# PrioQueue.insert PrioQueue.empty 1 "hello";;
- : string PrioQueue.queue =
PrioQueue.Node (1, "hello", PrioQueue.Empty,
PrioQueue.Empty)
```

Pregled

1. Koncept modula
2. Vrste modulov
3. Moduli v C
4. Moduli v Javi
5. Moduli v Ocaml
6. Jezik modulov
7. Parametrizirani moduli (funktorji)

Parametrizirani moduli

- Parametrizirani moduli ali funktorji
- Parametrizirani moduli so generični moduli osnovani na modulih, ki so podani kot parameteri
  - Modul (APT) se poda kot parameter gostujočemu modulu in s tem zagotovi podatkovne tipe in operacije definirane s parametrom gostitelja
  - Relacija med parametriziranimi moduli in moduli je podobna relaciji med funkcijami in vrednostmi
  - Podobno kot klic funkcije konstruira vrednost, invokacija parametriziranega modula konstruira nov modul
  - Funktorji so funkcije, ki slikajo iz modulov v module

Parametrizirani moduli

- Funktorji razširijo programske jezike s konstrukti, ki izboljšajo ponovno uporabnost kode
- Sintaksa definicije funktorja je blizu sintakse za definicijo funkcije
  - … tudi za okrajšavo

```
functor ( Name : signature ) -> structure
module Name1 ( Name2 : signature ) = structure
```

- Primer enostavnega funktorja Couple, ki uporablja kot parameter modul Q
  - Q definira tip Q.t
  - V modulu Couple definiramo tip Couple.t = Q.t * Q.t

```
# module Couple = functor ( Q : sig type t end ) ->
    struct type couple = Q.t * Q.t end ;;
```

Parametrizirani moduli

- Funktorji z več kot enim parametrom (modulom)
  - Osnovna sintaksa
  - Okrajšana sintaka

```
functor ( Name1 : signature1 ) ->
        ...
        functor ( Namen : signaturen ) -> structure
```

```
module Name (Name1:signature1) . . . (Namen:signaturen) = structure
```

  - Aplikacija funktorja

```
module Name = <functor> (structure1) . . . (structuren)
```

- Zaprti funktor je funktor, ki ne uporablja drugih funktorjev razen definiranih parametrov
  - Uporaba zaprtih funktorjev izboljša generativnost kode

Primer funktorja

```
# type comparison = Less | Equal |
                          Greater;;
type comparison = Less | Equal | Greater
# 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 OrderedString =
   struct
     type t = string
     let compare x y = if x=y then Equal else if x < y then Less else Greater
   end;;
module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end

# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```