# Vprašanja za ustni izpit pri Programiranju I

## Iztočnice za funkcijsko programiranje

### Najpogostejši tipi v OCamlu

okrajšave tipov

- int
- float
- string
- unit
- char
- bool

Zelo pomembna je razlika med tipoma *int* in *float*. Pri *intih* uporabljamo klasične operatorje, medtem, ko za *floate* tem operatorjem dodamo piko na koncu.

tipi funkcije

$$tip_{arg} \rightarrow tip_{rez}$$

tip naborov [1]

$$tip_1 * tip_2 * \cdots * tip_n$$

tip seznama

$$tip_{el} \quad list$$

OCaml je pri tipih v seznamu zelo dosleden. Hkrati so lahko v seznamu zgolj elementi istega tipa.

Vrednostim, ki imajo v tipih spremenljivke, pravimo parametrično polimorfne. *Using parametric polymorphism, a function or data type can be written generically so that it can handle values identically without depending on their type.* [2]

### Primerjava med statičnimi in dinamičnimi tipi

OCaml uporablja statične tipe, Python pa dinamične.

statični tipi:

- compiler preveri, da so vsi tipi v funkciji pravi, preden poženemo program

[1] Prazen nabor je tipa unit().
[2] $\alpha$ tip

- s tem pridobimo to, da funkcija vedno vrže isti tip ven, kar je zelo uporabno za kompozicije

dinamični tipi:

- programski jezik preveri pravilnost funkcije šele, ko jo kličemo

OCamlovi tipi so bogatejši od Pythonovih, saj nam dajo več podatkov [3]

### Repni klici in repna rekurzija

Klic funkcije je *repen*, če se izvede zadnji. Funkciji, kjer so vsi klici repni, pravimo *repno rekurzivna funkcija*. Pri *repno rekurzivnih funkcijah* spreminjamo samo akumulator, zato si računalniku ni treba zapomniti celotnega procesa ampak zgolj vmesne vrednosti.

OCaml optimizira repne klice.

Python repnih klicev namenoma ne optimizira, da so sporočila o napakah bolj poučna. Tega, da sklad ne raste, ne dosežemo tako, da optimiziramo repno rekurzijo, ampak da optimiziramo repne klice na splošno. Torej poljubna funkcija, ki kliče drugo funkcijo, izgine iz kode. Razlog, zakaj ne želimo po privzetem načinu izvzeti repnih klicev je v tem, da dobimo lep izpis o napakah. Če optimiziramo repne klice in vmesne funkcije izginejo iz zgodbe, se lahko "*izgubimo*". [4]

### Parametrični tipi in polimorfne funkcije

### Funkcije višjega reda in anonimne funkcije

Anonimne funkcije so funkcije, ki jih *ne poimenujemo*. V OCamlu jih zapišemo kar kot *fun* $x \rightarrow$ *fun(x)*. Uporabne so predvsem kadar jih uporabimo zgolj enkrat in na seznamu.

```
List.map (fun x -> x * x) [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

This is handy, as we would otherwise have to name the function first (see let) to be able to use it:

```
let square x = x * x
(* val square : int -> int = <fun> *)

List.map square [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

[3] Za sezname celih števil nam OCaml vrne tip *int list*, medtem, ko Python vrne zgolj *list*.
[4] PRI OBEH NAPIŠI PREDNOSTI IN SLABOSTI

Funkcije nam ob dani spremenljivki vrnejo določene vrednosti. Če te vrednosti uporabimo naprej v drugi funkciji, govorimo o *funkcijah višjega reda.* Torej funkcija višjega reda je vsaka funkcija, ki kot argument uporabi neko drugo funkcijo v določeni spremenljivki.

```
let twice f x = f (f x)
(* twice : ('a -> 'a) -> 'a -> 'a *)
```

Če funkciji, ki sprejme dva argumenta podamo samo en argument, dobimo novo funkcijo. Če mi to funkcijo poimenujemo, jo lahko (kasneje) uporabimo na drugem argumentu.

V OCamlu je navada, da pišemo *fun* $x \rightarrow$ *fun(x).* Operator $\rightarrow$ v tem zapisu je desno asociativen.

### Delno uporabljene funkcije in curryiranje

Če imamo funkcijo, ki sprejme več argumentov, mi pa ne podamo vseh, dobimo *delno uporabljeno funkcijo.* Dobimo torej novo funkcijo, ki ima sprejete že podane argumente, na ostale pa čaka. Ko ji dodamo tudi ostale argumente, deluje identično, kot bi delovala, če ji vse argumente podamo kar na začetku.

Curryirane funkcije [5]

Imamo funkcijo dveh argumentov, ki torej sprejme prvi argument, sprejme drugega in nekaj naredi. Če ji podamo samo prvi element, dobimo novo funkcijo (ki čaka na drug argument).

To vidimo tudi kadar imamo funkcijo $f$, ki sprejme dva argumenta $x$ in $y$. V OCamlu to napišemo kot $fxy$. Ta zapis je enak kot $(fx)y$, torej $(fx)$ je funkcija, ki čaka še drugi argument $y$, ki je v tem primeru že podan.

primer:

```
f x y z = ((f x) y) z

A -> B -> C -> D = A -> (B -> (C -> D))
```

[5] Haskell Curry - po njem se imenuje tudi programski jezik Haskell.

Tipa $A * B \to C$ in $A \to B \to C$ sta si izomorfna. Prvi tip predstavlja *običajno funkcijo dveh argumentov*, drugi pa *Curryirano funkcijo dveh argumentov*. Izomorfizmu med tema dvema funkcijama pravimo *curryiranje*.

$$C^{A \times B} \cong (C^B)^A$$

$$curry\colon C^{A \times B} \to (C^B)^A, uncurry\colon (C^B)^A \to C^{A \times B}$$

**Vsote in induktivni tipi**

**Različice funkcije *fold***

Funkcije *fold* so funkcije višjega reda. Gre za družino funkcij, ki "nabirajo" vrednosti elementov po vrsti. Primer funkcije, ki sodi v družino *fold*, je funkcija max.

Uporabimo v funkcijah, kjer sledimo vzorcu:

- vzami seznam in začetno vrednost akumulatorja
- sprehodi se čez seznam
- za vsak element v seznamu *naredi nekaj* s trenutno vrednostjo akumulatorja in trenutnega elementa
- nadaljuj z iteracijo po seznamu
- ko prideš do konca seznama, vrni akumulator

Ta postopek lahko skrajšamo s funkcijo *fold*, ki sprejme tri argumente. (*funkcijo f, začetno vrednost akumulatorja* in *seznam elementov*)

`fold_right`

- vrstni red združevanja elementov je desno → levo
- ni repno rekurzivna

`fold_left`

- vrstni red združevanja elementov je levo → desno
- repno rekurzivna

Tipa funkcij sta različna.

```
# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

# List.fold_right;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

### Dokazovanje z indukcijo na seznamih in drevesih

Indukcija na naravnih številih

$$P(0) \wedge (\forall m.P(m) \Rightarrow P(m^+)) \Rightarrow \forall n.P(n)$$

Indukcija na seznamih

$$P([]) \wedge (\forall x, xs.P(xs) \Rightarrow P(x :: xs)) \Rightarrow \forall ys.P(ys)$$

ideja:

Vsak seznam je ali *prazen* ali pa ima *glavo in nek krajši seznam.* [6] Če imamo predikat $P$, ki velja na praznem seznamu in velja, da iz tega, ko $P$ velja na repu $xs$, $P$ velja tudi na glavi in repu $x :: xs$, potem predikat $P$ velja na celem seznamu.
Ne moremo reči, da $P$ velja za $x$, saj je $P$ predikat, ki velja na seznamih, $x$ pa je zgolj element seznama, zato je nesmiselno govoriti o $P(x)$.
Indukcija na seznamih deluje, ker jo lahko prevedemo na indukcijo na naravnih številih. Iz predikata $P$ na seznamih moramo predelati predikat $Q$ na naravnih številih. To storimo tako, da definiramo

$$Q(n) := \forall ws.|ws| = n \Rightarrow P(ws).$$

Z besedami to pomeni *"$Q(n)$ bo veljalo, če za vse sezname dolžine $n$ velja $P(ws)$."* V tem primeru je $Q(0)$ ekvivalentno $P([])$. To, da gremo iz tega, da če velja za rep, velja tudi za glavo, je ravno korak indukcije na naravnih številih. Mi predpostavimo, da velja za sezname $\cdots$ [7].

Indukcija na drevesih

$$P(Empty) \wedge (\forall x, l, d.P(l) \wedge P(d) \Rightarrow P(Node(l, x, d))) \Rightarrow \forall t.P(t)$$

Najprej pokažemo, da predikat velja za prazno drevo. V indukcijskem koraku predpostavimo, da velja za *levega in desnega otroka*. Razlika od indukcije na dobro urejenih množicah je v tem, da smo tam predpostavili, da velja za *vse* manjše. Tu predpostavimo, da velja samo za dve *direktno manjši drevesi*. Pomembno pri indukciji na drevesih je to, da imamo neko *velikost*, nekaj, s čimer lahko indukcijo na drevesih prevedemo na indukcijo na naravnih številih. Dober primer je *globina*.

[6] Ignoriramo neskončne sezname.
[7] DOPIŠI

Indukcija na poljubnem tipu

```
type nek_moj_tip =
| A
| B of nek_moj_tip
```

1. pokažemo, da predikat $P$ velja na $A$
2. pokažemo, da iz tega, da velja za *"nek_moj_tip"*, velja tudi za $B$

$$P(A) \wedge \forall m.P(m) \Rightarrow P(Bm) \Rightarrow \forall n.P(n)$$

Indukcija za bool

$$P(A) \wedge P(B) \Rightarrow \forall b.P(b)^8$$

### Signature in moduli

Modularnost kode nam pomaga, da kodo razbijemo na manjše enote, ki med seboj ustrezno sodelujejo. Osnovno modularnost nam ponuja pisanje funkcij in tipi, dosežemo pa jo lahko tudi z *moduli*.

Modul je zbirka tipov in vrednosti. Vse definicije tipov in funkcij lahko *združimo* v zbirko in tej zbirki pravimo modul. Vsaka datoteka je že sama po sebi modul funkcij in tipov, ki so napisane v tisti datoteki. Do tipov in funkcij modula dostopamo tako, da napišemo *modul.funkcija* oz. *modul.tip*.

Razlika med datoteko funkcij in tipov in modulom je v tem, da lahko imamo v eni datoteki več modulov. Vsebina modula je vse med *struct* in *end*. Tip modula se imenuje *signatura* in pove, kaj je v modulu shranjeno.

Funkcije lahko v modulu tudi skrivamo. Če signaturi nekega modula priredimo novo signaturo, skrijemo vse funkcije, ki niso definirane v novi signaturi. Funkcije lahko še vedno uporabljamo.

## Iztočnice za podatkovne strukture in algoritme

### Računska zahtevnost

### Iskalna drevesa in AVL drevesa

V iskalna drevesa lahko spravljamo zgolj elemente, ki jih lahko primerjamo.

[8] Predpostavki sta samo, da velja za $A$ in za $B$. Nimamo nobenih manjših problemov, s katerimi bi prevedli na "navadno" indukcijo.

**Spremenljive in nespremenljive podatkovne strukture**

**Predstavitev podatkov v pomnilniku**

**Razlika med verižnimi seznami in tabelami**

**Metoda deli in vladaj**

**Fisher - Yatesov algoritem**

**Algoritmi za urejanje**

**Časovna zahtevnost hitrega urejanja**

**Dinamično programiranje in memoizacija**