Vprašanja za ustni izpit pri Programiranju I

Contents

1 Iztočnice za funkcijsko programiranje

1.1 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]

1.2 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
- s tem pridobimo to, da funkcija vedno vrže isti tip ven, kar je zelo uporabno za kompozicije

[1] Prazen nabor je tipa unit().
[2] α tip

dinamični tipi:

- programski jezik preveri pravilnost funkcije šele, ko jo kličemo

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

1.3 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]

1.4 Parametrični tipi in polimorfne funkcije

1.5 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] *)
```

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.

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

```
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* $x \to$ *fun(x)*. Operator $\to$ v tem zapisu je desno asociativen.

1.6 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*.

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

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

1.7 Vsote in induktivni tipi

1.8 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>
```

1.9 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

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

1.10 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.

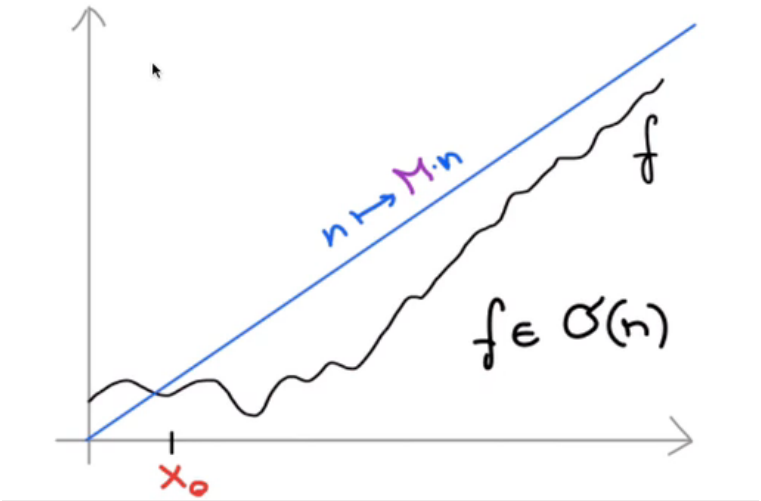
[8] Predpostavki sta samo, da velja za A in za B. Nimamo nobenih manjših problemov, s katerimi bi prevedli na "navadno" indukcijo.

2 Iztočnice za podatkovne strukture in algoritme

2.1 Računska zahtevnost

Računsko zahtevnost računamo s pomočjo velikega O.

$$\exists M, x_0. \forall x > x_0. |f(x)| \leq M \cdot |g(x)|$$



Ker z linearno funkcijo omejimo navzgor in ker je ničla linearne funkcije v $x = 0$, nas računska zahtevnost zanima zgolj za velike x. Časovna zahtevnost nam pove, koliko časa potrebuje program, da nekaj izvede, prostorska zahtevnost pa koliko pomnilnika pri tem porabi.

S $T(n)$ označimo čas, ki ga porabi funkcija za seznam dolžine n. Velja $T(0) = O(1)$, kar pomeni, da za prazen seznam porabi konstanten čas. Čas, ki ga porabimo za seznam dolžine $n + 1$ pa je $T(n + 1) =$. Seznam je potrebno razpakirati (ločimo na glavo in rep), nato izvedemo funkcijo na repu, ki je dolžine n. Pri tem porabimo torej $T(n)$ časa. Na koncu damo še glavo nazaj, za kar porabimo $O(1)$ časa, potem pa še staknemo. Pri stikanju gre za funkcijo na dveh seznamih. Časovna zahtevnost je v odvisnoti od levega. [9] [10]

[9] Dela rekurzivno na levem.
[10] DODAJ ŠE SKICO IZRAČUNA ČASOVNE ZAHTEVNOSTI STIKANJA

2.2 Iskalna drevesa in AVL drevesa

Dvojiško[11] drevo je *iskalno*, če

- so vsi elementi *levega* otroka *manjši* od korena
- so vsi elementi *desnega* otroka *večji* od korena
- sta oba otroka *tudi iskalni drevesi*

V iskalna drevesa zelo lahko dodajamo in iščemo elemente. Pri odstranjevanju se nam lahko zgodi, da pobrišemo koren nekega iskalnega poddrevesa. V tem primeru koren nadomestimo z največjim manjšim elementom, torej elementom, ki je skrajno desni na levi strani ali najmanjšim izmed največjih, torej s skrajno levim na desni strani.

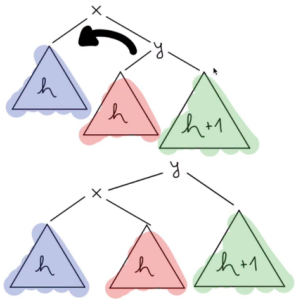
Očitno lahko v iskalna drevesa spravljamo zgolj elemente, ki jih lahko med seboj primerjamo.

Operacije v iskalnem drevesu imajo časovno zahtevnost $O(h)$. [12] Velja le, če je drevo *uravnoteženo*. Primeri uravnoteženih dreves so *AVL drevesa*.

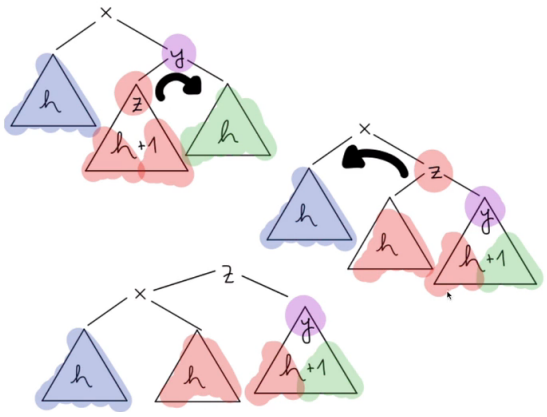
Iskalno drevo je AVL drevo, če

- je razlika višin obeh otrok *največ* 1
- sta oba otroka *tudi AVL drevesi*

Dodajanje in brisanje elementov AVL dreves jih lahko pokvari na enega izmed dveh načinov. Lahko je globina levega poddrevesa za 2 večja od globine desnega ali obratno. Vsaka nadaljna neuravnoteženost ima dve različici. Več kot dveh ne more imeti, saj AVL drevesa tvorimo tako, da jih sproti urejamo, zaradi česar je razlika globin obeh otrok ob vsakem trenutku največ 1. To neuravnoteženost lahko popravimo s pravo rotacijo. [13]



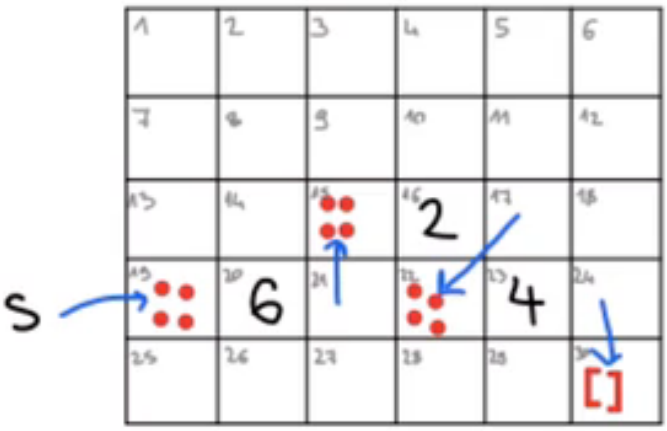
[11] Ali je *prazno*, ali ima pa *koren in levega in desnega otroka.*
[12] h pomeni height
[13] RAZLOŽI, KDAJ UPORABIMO KAKŠNO ROTACIJO + POPRAVI MESTA SLIK



2.3 Spremenljive in nespremenljive podatkovne strukture

2.4 Predstavitev podatkov v pomnilniku

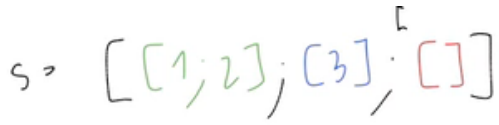
V OCamlu so seznami implementirani z verižnimi seznami. To pomeni, da vsak seznam sestavljata glava in rep. V vsakem prostoru je napisan en podatek.

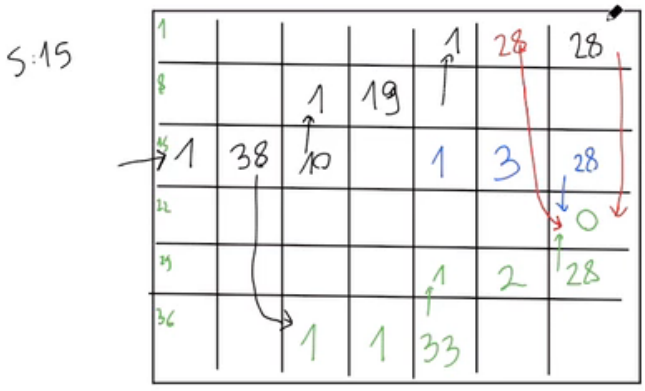


Vidimo, da je seznam *s* shranjen na mestu 19. V pomnilniku je nato shranjeno tudi, da je to seznam, sestavljen iz glave in repa. Na naslednjih dveh mestih so podatki o glavi in repu. Glava oz. prvi element je torej 6. Na naslednjem mestu je spravljen podatek o repu, ampak tudi repa ne moremo spraviti v zgolj en prostor, lahko si pa zapomnimo, kje je shranjen rep. Postopek se tako nadaljuje.

Prednost takega seznama je v tem, da lahko elemente dodamo v seznam v konstantnem času, saj samo poiščemo prostor v pomnilniku (potrebujemo dva kvadratka) za ta element in *kazalec*, ki kaže na rep seznama. Ko OCaml gleda pomnilnik ve, kdaj številka zgolj kaže na mesto nadaljevanja seznama in kdaj predstavlja element v seznamu. To mu omogočajo tipi. V Pythonu stvar deluje malo drugače. Seznam je shranjen v enem kosu, kar pomeni, da če želimo dodati nov element v seznam, moramo v pomnilniku poiskati dovolj prostora za seznam dolžine $n-1$ in element, ki ga želimo dodati temu seznamu, nato pa cel seznam preslikamo na teh n mest. Časa porabimo linearno.

Primer pomnilnika za določeni seznam.





2.5 Razlika med verižnimi seznami in tabelami

Verižni seznami so opisani že v zgornjem subsectionu.

2.6 Metoda deli in vladaj

Filozofija metode deli in vladaj je v tem, da problem razbijemo na manjši problem. Izvedemo jo v treh korakih

- nalogo razdelimo na manjše podnaloge

- podnaloge rekurzivno rešimo
- dobljene rešitve združimo v rešitev prvotne naloge

2.7 Fisher - Yatesov algoritem

2.8 Algoritmi za urejanje

urejanje z mehurčki [14]

Vsakič pogledamo dva sosednja elementa. Če je prvi manjši od drugega, ju pustimo pri miru, sicer ju zamenjamo. Tako se sprehajamo čez seznam, dokler ne dobimo urejenega seznama. Časovna zahtevnost tega algoritma je kvadratna. [15]

urejanje z izbiranjem

Zapeljemo se čez seznam in poiščemo najmanjšega oz. največjega in ga damo na začetek oz. na konec. Nato se ponovno zapeljemo čez preostanek seznama. Časovna zahtevnost je kvadratna, saj je seznam vedno manjši, ampak je manjši linearno.

urejanje z vstavljanjem

Imamo neurejen seznam in ga urejamo enega po enega. Vsakič, ko dobimo nov element, pogledamo kam spada in ga vstavimo tja. Bisekcije ne uporabljamo, ker sicer pravo mesto najdemo hitro (v logaritemskem času), ampak moramo potem vse elemente prestaviti, za kar porabimo linearno časa. Če iščemo enega po enega, s tem hkrati že premikamo. Pri bisekciji bi morali vse operacije ponoviti še enkrat. Časovna zahtevnost je približno $\frac{n^2}{2}$.

Čeprav so to načeloma precej neoptimizirani algoritmi, niso nujno neuporabni. Zelo lahko jih napišemo, na urejenih seznamih delujejo hitreje [16], tretja prednost pa je v *lokalnosti pomnilnika*. Če procesor deluje vedno na manjšem kosu, recimo primerja dva sosednja elementa, je to precej hitreje, kot če bi vsakega primerjal z vsakim. Na večjih seznamih zato navadno začnemo z zahtevnejšimi seznami, nato pa za majhne sezname uporabimo enega izmed zgornjih algoritmov.

2.9 Časovna zahtevnost hitrega urejanja

2.10 Dinamično programiranje in memoizacija

[14] intuicija - mehurčki v vodi plavajo navzgor, prav tako gredo večji elementi proti vrhu seznama

[15] n-krat gremo čez in vsakič imamo enega manj

[16] Urejanje z mehurčki bi šlo čez urejen seznam zgolj enkrat.