Uvod u funkcionalno programiranje u ML-u

Andrej Ivašković

Matematička gimnazija, Nedelja informatike

2. april 2015.

Paradigme programiranja

- Imperativno programiranje:
  - programu se "kaže" šta tačno treba da uradi;
  - Java, C, C++, Pascal, Fortran, Python...
- Deklarativno programiranje:
  - program je, u stvari, jedna dovoljno precizna definicija našeg problema;
  - nekoliko podvrsta:
    - jezici za operacije nad bazama podataka: SQL;
    - logičko programiranje: Prolog;
    - funkcionalno programiranje: ???

Paradigme programiranja

- Imperativno programiranje:
  - programu se "kaže" šta tačno treba da uradi;
  - Java, C, C++, Pascal, Fortran, Python...
- Deklarativno programiranje:
  - program je, u stvari, jedna dovoljno precizna definicija našeg problema;
  - nekoliko podvrsta:
    - jezici za operacije nad bazama podataka: SQL;
    - logičko programiranje: Prolog;
    - funkcionalno programiranje: Lisp, Haskell, ML, OCaml, F#.

Karakteristike funkcionalnog programiranja

- Sveprisutnost rekurzije (izbegavaju se ciklusi).
- Programer treba ispravno da *definiše* zadatak.

\+ Fleksibilnost i sloboda.

\+ Podrška za lenja izračunavanja.

∼ Neefikasnost konačnog programa.

! Karakterističan način razmišljanja, često stran običnim smrtnicima.

ML interpreter

- ML je **interpreterski jezik**: sve funkcije i svi upiti se vrše u interaktivnom okruženju.
- Koristićemo Poly/ML.
- Moguće je konsultovati bazu znanja preko:

```
use "baza.sml";
```

- Šta znači sledeće?

```
> val x = 3 + 5;
> val y = x - 9;
```

nakon ovih naredbi imamo ispis:

```
val x = 8 : int
val y = ~1 : int
```

- Promenljive **ne mogu da menjaju vrednost**, mogu samo da se redefinišu!

Primeri funkcija

- Identifikator funkcije je njeno ime, a ne vrsta argumenata!

```
1 fun kvadriraj x = x * x;
```

je ekvivalentno sa

```
1 fun kvadriraj (x : int) : int = x * x;
```

- **Type checking** i **pattern matching**.
- Možemo da imamo par brojeva kao argument:

```
1 fun saberi (x : real, y : real) = x + y;

val saberi = fn : (real * real) -> real
```

- Funkcija ima svoj tip — vratićemo se ovoj ideji kasnije!

if/then/else

- Postoji koncept grananja, ali mora da bude potpuno:

```
1 fun paran n =
2   if n mod 2 = 0 then "paran"
3                  else "neparan";
```

- U dosta primera možemo da koristimo pattern matching i *wildcard* na pametan način:

```
1 fun nula 0 = true
2   | nula _ = false;
```

- Voditi računa o tome da su svi slučajevi pokriveni!

Višestruko grananje

```
1 fun vrsta n =
2   case n mod 3 of 0 => "lud"
3                 | 1 => "zbunjen"
4                 | 2 => "izgubljen";
```

- Ovo imaju i drugi jezici (ali verovatno nikad/retko koristite).
- Veoma korisno kada pravimo enumeracijske tipove...

let

- Šta ako želimo da imamo pomoćna izračunavanja, a ne želimo da pravimo dosta malih i besmislenih funkcija?
- Ispitujemo da li je $f\left(f\left(x\right)\right) > 0$, gde je $f\left(x\right) = x^2 - 3x$:

```
1 fun probaj x =
2   let fun f t = t * t - 3.0 * t
3       val r = f (f x)
4   in r > 0
5   end;
```

- Pomoćni podaci su vidljivi samo u opsegu u okviru kog postoje.
- Korisno u komplikovanijim problemima.
- Loše je preterivati sa `let`-om!

Rekurzivne funkcije

- Ciklusi nemaju smisla u funkcionalnom programiranju!
- Za računanje x^n koristimo $x^0 = 1$, $x^n = x \cdot x^{n-1}$:

```
1 fun stepen (x, 0) = 1
2   | stepen (x, n) = x * stepen (x, n - 1);
```

- Rekurzivno ćemo definisati i strukture podataka.
- Implementirano preko steka.
- Neophodno je biti vrlo oprezan!

Brže stepenovanje

- Uporediti sledeće dve funkcije:

```
fun brzst1 (x, 0) = 1
  | brzst1 (x, n) =
      if n mod 2 = 0 then brzst1 (x, n div 2)
                        * brzst1 (x, n div 2)
                     else x * brzst1 (x, n - 1);
```

```
fun brzst2 (x, 0) = 1
  | brzst2 (x, n) =
      let val pola = brzst2 (x, n div 2)
      in if n mod 2 = 0 then pola * pola
                        else x * pola * pola
      end;
```

- Ovde imamo ideju koja unapređuje složenost sa $O(n)$ na $O(\log n)$, ali imamo i tehnički detalj...

Tail-recursion

- Rekurzija zahteva stek, vremenska i memorijska neefikasnost (razlozi?).
- Uvodimo pojam **akumulatorskog argumenta** i koristimo **tail-recursive** logiku.

```
1 fun stepen (x, n) =
2   let fun st (0, r) = r
3     | fun st (k, r) = st (k - 1, x * r)
4   in st (n, 1)
5   end;
```

Naravno, ovo može da se obavi na više sličnih načina...

- Postoji i u drugim jezicima, efikasno jedino ukoliko kompajler detektuje!

Tail-recursion: još primera

```
1 fun faktorijel n =
2   let fun fakt (k, r) =
3         if k > n then r
4                  else fakt(k + 1, k * r)
5   in fakt (1, 1)
6   end;
```

- Kako bismo izveli brzo stepenovanje?

ML tipovi

- Imamo razne celobrojne tipove: Poly/ML ima *proizvoljnu preciznost* celih brojeva u tipu `int`.
- Floating-point racionalni brojevi: `real`.
- Imamo i tip `string`: `"ovo je string"`.
- Logički tip `bool`, vrednosti su `true` i `false`.
- 8-bitni karakteri, `char`: `#"c"`.
- Provera tipova radi na veoma opštem nivou, te možemo da imamo **statički polimorfizam**.

n-torke

- Predstavljaju Dekartove proizvode skupova kojima pripadaju vrednosti.
- Tip para `(u, v)`, gde je α tip `u`, a β tip `v`, jeste $\alpha * \beta$.
- Dosadašnje funkcije sa "više argumenatašu operisale nad n-torkama.
- Tačno je određena dužina jedne n-torke.
- Postoji i **prazna n-torka, 0-torka**, posebnog tipa `unit`, gde je jedina dopuštena vrednost `()`. Ovo ima smisla da bude povratna vrednost u proceduralnom programiranju...

Liste

- Funkcionalni ekvivalent niza, "n-torka sa proizvoljnim n".
- Rekurzivna definicija:
  - Prazna lista `[]` je lista tipa α.
  - Element tipa α nadovezan sa listom tipa α je lista tipa α. Nadovezivanje zovemo `cons`, koristimo `::`.
- `[1, 2, 3]` $\Longleftrightarrow$ `1::(2::(3::[]))`
- Tip ove liste je `int list`.
- **Glava** (`hd`) liste je njen prvi član.
- **Rep** (`tl`) liste je lista kojoj je izbačen prvi član.
- **Konkatenacija** dve liste (`xs@ys`): `[1, 3, 5] @ [2, 4]` daje `[1, 3, 5, 2, 4]`. Moguće samo ako tipovi odgovaraju!

Neke funkcije nad listama

```
fun zbir [] = 0
  | zbir (x::xs) = x + zbir xs;
```

```
fun nakraj1 (x, []) = [x]
  | nakraj1 (x, y::ys) = y::nakraj1 (x, ys);
```

```
fun nakraj2 (x, l) = rev (x::(rev l));
```

```
fun rev1 [] = []
  | rev1 (x::xs) = (rev1 xs) @ [x];
```

```
fun rev2 l =
  let fun r ([], rez) = rez
        | r (x::xs, rez) = r (xs, x::rez)
  in r (l, [])
  end;
```

```
val nakraj1 = fn : ('a * 'a list) -> 'a list
```

Službena reč type

- Nekada želimo da definišemo svoje tipove i da koristimo malo kompaktniji zapis.

```
type rgbcolour = int * int * int;

fun getR ((r, g, b) : rgbcolour) = r;
fun getG ((r, g, b) : rgbcolour) = g;
fun getB ((r, g, b) : rgbcolour) = b;

val getR = fn : rgbcolour -> int
val getG = fn : rgbcolour -> int
val getB = fn : rgbcolour -> int
```

- U prethodnom primeru smo uspostavili potpuno ekvivalenciju između `rgbcolour` i `int*int*int`.

Enumeracijski tip

- Želimo da jednim tipom obuhvatimo više slučajeva.
- Vozilo može da bude ili bicikl ili automobil, u oba slučaja može da se ukrade:

```
datatype vozilo = Bajs of string
                | Kola of string * int;
```

```
fun ukradi (Bajs s) = s ^ "nema vise bajs!"
  | ukradi (Kola(_, c)) =
      "drpio si kola od " ^ c ^ " funti!"
```

- Sa ovakvim stvarima ćemo moći da budemo vrlo kreativni...

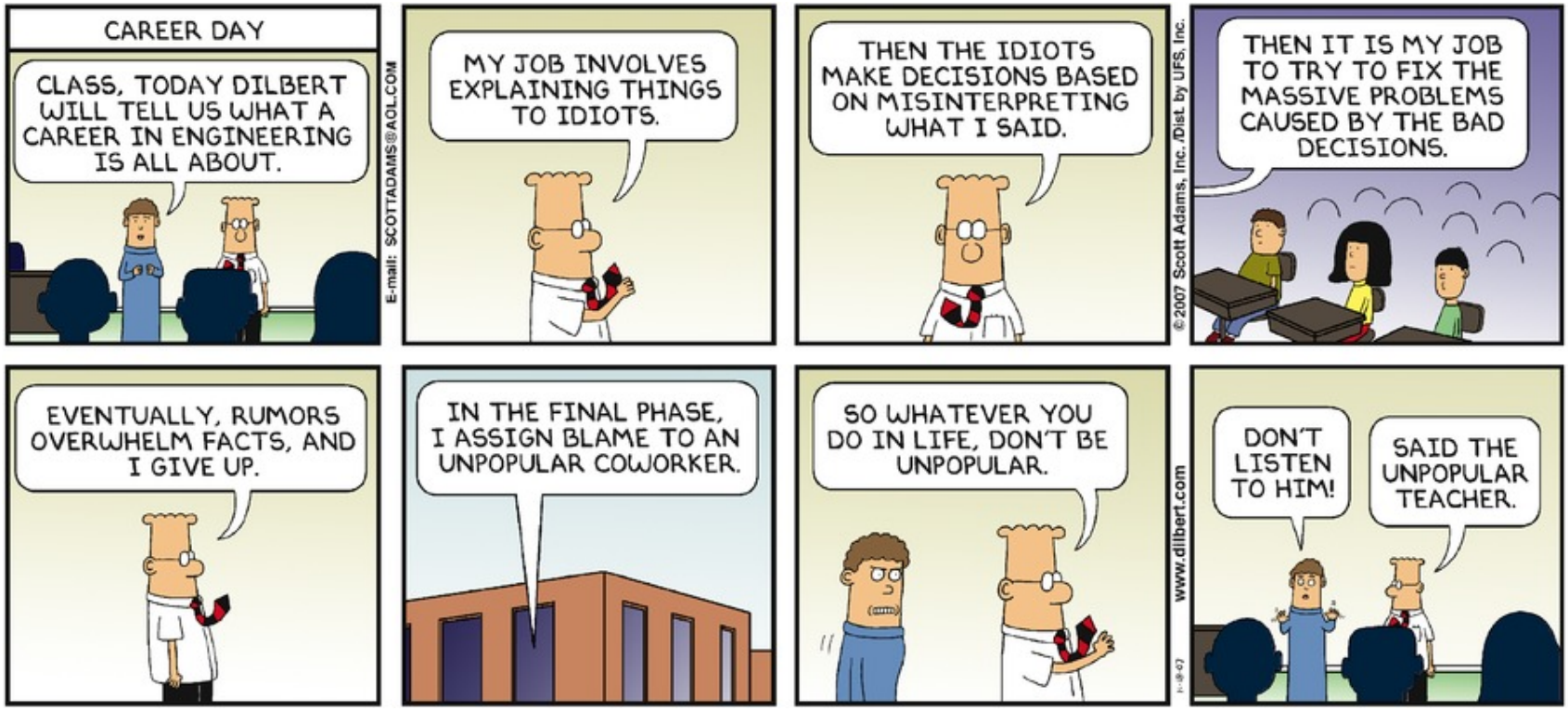
Tip funkcije

- Funkcije imaju tip oblika $\alpha \to \beta$.
- Funkcija takođe može da bude vrednost npr.

```
fun triput x = [x, x, x];
val f = triput;
val l = f 5;
```

- Funkcije ne mogu da se uporede tj. test jednakosti nije moguć!

```
fun f (x, y, z, n) = 0;
fun g (x, y, z, n) =
  if n > 2 andalso
      step (x, n) + step (y, n) = step (z, n)
    then 1
    else 0;
```


CAREER DAY
CLASS, TODAY DILBERT WILL TELL US WHAT A CAREER IN ENGINEERING IS ALL ABOUT.
E-mail: SCOTTADAMS@AOL.COM
MY JOB INVOLVES EXPLAINING THINGS TO IDIOTS.
THEN THE IDIOTS MAKE DECISIONS BASED ON MISINTERPRETING WHAT I SAID.
© 2007 Scott Adams, Inc. /Dist. by UFS, Inc.
THEN IT IS MY JOB TO TRY TO FIX THE MASSIVE PROBLEMS CAUSED BY THE BAD DECISIONS.
EVENTUALLY, RUMORS OVERWHELM FACTS, AND I GIVE UP.
IN THE FINAL PHASE, I ASSIGN BLAME TO AN UNPOPULAR COWORKER.
SO WHATEVER YOU DO IN LIFE, DON'T BE UNPOPULAR.
www.dilbert.com
DON'T LISTEN TO HIM!
SAID THE UNPOPULAR TEACHER.

Funkcija kao deo argumenta

- Ništa nas ne sprečava da imamo funkciju u okviru argumenta:

```
1 fun primeni (f, g, x) = f (g x);
```

- Tip prethodne funkcije je $(\beta \to \gamma) \times (\alpha \to \beta) \times \alpha \to \gamma$.
- Sa ovim imamo već neke mogućnosti, ali nas ograničava činjenica da koristimo n-torku, te moramo da znamo sve argumente unapred.

Currying

- Da bismo zaista postigli više argumenata, dopustićemo da prosleđivanje samo prvog argumenta već fiksira jednu funkciju:

```
1 fun spoji x y = (x, y);
```

- Ako napišemo `spoji 3 2`, rezultat je par celih brojeva `(3, 2)`.
- Ukoliko posmatramo `spoji 3`, onda dobijemo "instanciranje"funkcije:

```
1 (* skica *) fun (spoji 3) y = (3, y);
```

- Tip funkcije `spoji` je:

```
val spoji = fn : 'a -> 'b -> 'a * 'b
```

- Koji bi bio tip izmenjene varijante `primeni`?

Funkcionali `map` i `filter`

```
map dupliraj [2, 7, 3];
```

```
val it = [4, 14, 6] : int list
```

```
fun map _ [] = []
  | map f (x::xs) = (f x)::map f xs;
```

```
filter paran [1, 2, 3, 4, 5];
```

```
val it = [2, 4] : int list
```

```
fun filter _ [] = []
  | filter p (x::xs) =
      if p x then x::filter p xs
             else filter p xs;
```

Primene map i filter

```
1 fun clan [] _ = false
2   | clan (x::xs) y = if x = y then true
3                                else clan xs y;
4 val pr = [2, 3, 5, 7, 11, 13];
5 val l = [1, 2, 3, 4, 5, 6, 7, 8];
6 val l1 = filter (clan pr) l;
7
8 fun dodaj x l = x::l;
9 val l2 = map (spoji 1) [[3, 4], [2, ~1, 0], [7]];
```

λ zapis funkcija

- Tek uvod u dosta dugu priču u vezi sa **teorijom izračunljivosti**.
- Opisaćemo funkciju $x \mapsto f\,(x)$ bez eksplicitnog davanja imena. Ukoliko želimo da prikažemo $x \mapsto x + 1$, tada pišemo:

$$\lambda x.x + 1$$

- Zbir dva broja se opisuje sa:

$$\lambda x.\lambda y.x + y$$

- Sam račun se vrši na sledeći način:

$$((\lambda x.\lambda y.x + y)\,(5))\,(6) = (\lambda y.5 + y)\,(6) = 5 + 6 = 11$$

- Naravno, možemo da radimo i mnogo zanimljivije stvari:

$$\lambda f.\lambda x.f\,(f\,(f\,(x)))$$

ML anonimne funkcije

- Zameniti λ sa `fn` i . sa =>:

```
1 val f = fn x => x + 1;
2 val g = fn x => fn y => x + y;
```

- Korisno kada se ukombinuje sa funkcionalima. Šta radi naredna funkcija?

```
1 fun allcons x l = map (fn y => x::y) l;
```

- Neki interesantni primeri:

```
1 map (fn x => "Dr " ^ x) ["Jovanovic", "Nedeljkovic"];
2 fun cr 0 = []
3   | cr n = n::cr (n - 1);
4 fun crt n = rev (cr n);
5 map (fn x => crt x) (crt 3);
```

Koji je tip x5?

```
1 val pair = fn x => (fn y => (fn z => (z x y)));
2 val x1 = fn y => (pair y y);
3 val x2 = fn y => (pair x1 x1);
4 val x3 = fn y => (pair x2 x2);
5 val x4 = fn y => (pair x3 x3);
6 val x5 = fn y => (pair x4 x4);
```

Potreba za lenjom listom i definisanje

- **Lenja izračunavanja** podrazumevaju da se računanje ne vrši sve do momenta kada su nam ti podaci potrebni.
- Neki jezici sva izračunavanja rade na lenj način, u ML-u to možemo (koliko-toliko) lako da simuliramo!
- Potencijalno liste beskonačne dužine!
- Lenja lista je ili prazna lista ili spoj glave i repa. Koristićemo funkciju koja slika u rep:

```
1 datatype 'a stream = Nil
2                    | Cons of 'a * (unit -> 'a stream);
```

- Primer lenje liste:

```
Cons(1, fn () => Cons(2, fn () => Nil))
```

Primeri funkcija sa lenjim listama

```
1  fun lazify [] = Nil
2    | lazify (x::xs) = Cons(x, fn () => lazify xs);
3
4  fun unlazify [] = Nil
5    | unlazify (Cons(x, xs)) = x::unlazify (xs());
6
7  fun append Nil l = l
8    | append (Cons(x, xs)) l =
9        Cons(x, fn () => append (xs()) l);
10
11 fun interleave Nil l = l
12   | interleave (Cons(x, xs)) l =
13       Cons(x, fn () => interleave l (xs()));
14
15 fun from n = Cons(n, fn () => from (n + 1));
```

Funkcionali na lenjim listama

```
1  fun mapq _ Nil = Nil
2    | mapq f (Cons(x, xs)) =
3        Cons(f x, fn () => mapq f (xs()));
4
5  val parni = mapq (fn x => 2 * x) (from 1);
6
7  fun filterq p Nil = Nil
8    | filterq p (Cons(x, xs)) =
9        if p x then Cons(x, fn () => filterq p (xs()));
10               else filterq p (xs());
```

- Zašto je naredni poziv problematičan?

```
val ups = filterq (fn x => x < 0) (from 1);
```

Pažnja!

- Veoma lako možemo da pogrešimo i da naše izračunavanje uopšte ne bude lenjo!

```
1 fun lose x = append (Cons(x, fn () => Nil))
2                     (lose (x + 1));
```

- Nekoliko načina da ovo zaobiđemo:
  1. Pravimo se da problem ne postoji.
  2. Smislimo drugačije rešenje zadatka (u ovom slučaju jednostavno).
  3. Zarad maksimalne lenjosti, možemo da uvedemo neelegantnost i UVEK tražimo da nam se prosledi funkcija koja slika u ostatak liste.

Zašto je ovo korisno

- Neka rešenja su vrlo elegantna! Vrednost *programerskog* i *programskog* vremena.
- Prirodan način razmišljanja ukoliko je priroda našeg problema rekurzivna.
- Dokazivači teorema mogu da se sastave na relativno elegantan način. Primer profesionalnog softvera: Isabelle theorem prover.

Šta ML dalje podržava?

- Obrada izuzetaka.
- Referentni tipovi.
- Sintaksa sa proceduralno/imperativno programiranje.
- Biblioteke za rad sa strukturama podataka, ulazom i izlazom...

Pitanja za razmišljanje

1. Kako bi mogla da se implementiraju binarna stabla pretrage u ML-u? Šta je (možda) problematično?
2. Kako bi se implementirale funkcije višeg reda u C-u? Koja ograničenja postoje pri pravljenju ekvivalenta `map` (ukoliko pretpostavimo da je ekvivalent liste — niz)?
3. ML deluje kao da je jezik u kom nemamo ništa, ali se ispostavlja da već sada imamo previše stvari. Konkretno, brojevi su suvišni jer bismo mogli da ih *simuliramo*. Na koji način?