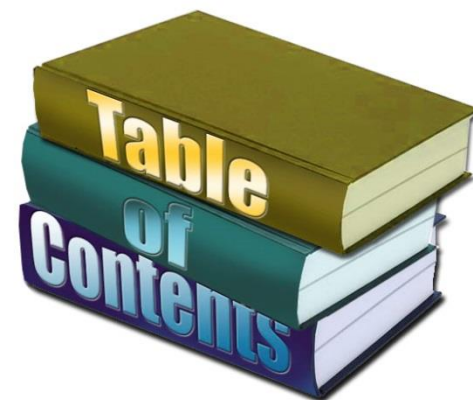


PROGRAMIRANJE

2014/15

interpreter: ovojnice in makri
argumenti funkcij
primerjava ML in Racket

Pregled



- optimizacija ovojnic
- implementacija makro sistema v JAIS
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov

Optimizacija ovojnic

- okolje v ovojnici lahko vsebuje spremenljivke, ki jih funkcija ne potrebuje
 - senčene spremenljivke iz zunanjega okolja
 - spremenljivke, ki so definirane v funkciji in senčijo zunanje
 - spremenljivke, ki v funkciji ne nastopajo
- ovojnice so lahko prostorsko zelo potratne, če so obsežne
- rešitev: zmanjšamo število spremenljivk v okolju ovojnice na nujno potrebne
- primeri nujno potrebnih spremenljivk
 - `(lambda (a) (+ a b c))`
 - `(lambda (a) (let ([b 5]) (+ a b c)))`
 - `(lambda (a) (+ b (let ([b c]) (* b 5))))`



Implementacija makro sistema

- makro sistem:
 - nadomeščanje (neprijazne) sintakse z drugačno (lepšo)
 - širitev sintakse osnovnega jezika
- v našem interpreterju (JAIS) lahko makro sistem implementiramo kar s funkcijami v jeziku Racket
- primeri

```
(define (in e1 e2)
  (ce-potem-sicer e1 e2 (bool #f)))
```

```
> (jais3 (in (bool #f) (bool #f)))
(bool #f)
> (jais3 (in (bool #f) (bool #t)))
(bool #f)
> (jais3 (in (bool #t) (bool #f)))
(bool #f)
> (jais3 (in (bool #t) (bool #t)))
(bool #t)
```

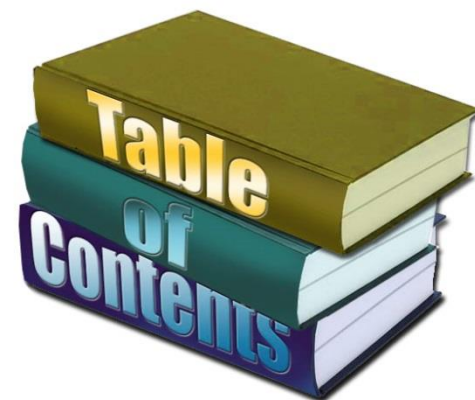
```
(define (vsota-sez sez)
  (if (null? sez)
      (konst 0)
      (sestej (car sez)
               (vsota-sez (cdr sez)))))
```

```
> (vsota-sez (list (konst 3) (konst 5)
                   (konst 2)))

(sestej (konst 3) (sestej (konst 5)
                          (sestej (konst 2) (konst 0))))
```

- je tak makro sistem higieničen?

Pregled



- optimizacija ovojnic
- implementacija makro sistema v JAIS
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov

Funkcije z različnim številom argumentov

- **poljubno število argumentov** podamo z imenom spremenljivke brez oklepaja. V funkciji so vsi ti argumenti podani v seznamu

A lambda expression can also have the form

```
(lambda rest-id  
  body ...+)
```

That is, a lambda expression can have a single *rest-id* that is not surrounded by parentheses. The resulting function accepts any number of arguments, and the arguments are put into a list bound to *rest-id*

```
(define izpisi  
  (lambda sez  
    (displayln sez)))
```

```
> (izpisi 1 2 3 4 5 6)  
(1 2 3 4 5 6)
```


```
(define vsotamulti  
  (lambda stevila  
    (apply + stevila)))
```

```
> (vsotamulti 1 2 3)  
6  
> (vsotamulti 1 2 3 11 33 -4)  
46
```

Funkcije z različnim številom argumentov

- definiramo lahko tudi funkcijo z **zahtevanim** in poljubnim številom **dodatnih neobveznih** argumentov

```
(lambda gen-formals  
  body ...+)  
  
gen-formals = (arg ...)  
              | rest-id  
              | (arg ...+ . rest-id)
```



```
(define mnozilnik  
  (lambda (ime faktor . stevila)  
    (printf "~a~a~a~a"  
            "Zivjo "  
            ime  
            ", tvoj rezultat je: "  
            (map (lambda (x) (* x faktor)) stevila))))
```

```
> (mnozilnik "Frodo" 42 1 4 5 2 3)  
Zivjo Frodo, tvoj rezultat je: (42 168 210 84 126)
```

Funkcije z imenovanimi argumenti

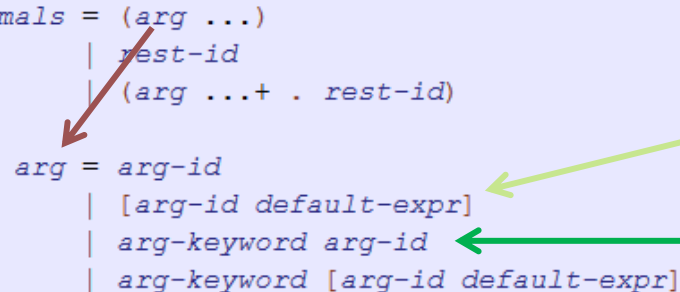
- argumente lahko podamo s ključnimi besedami
 - notacija: *#:beseda*
 - takšni argumenti se pri klicu funkcije upoštevajo glede na ključno besedo in ne glede na podani vrstni red
- sintaksa

A lambda form can declare an argument to be passed by keyword, instead of position. Keyword arguments can be mixed with by-position arguments, and default-value expressions can be supplied for either kind of argument:

```
(lambda gen-formals
  body ...+)

  gen-formals = (arg ...)
                | rest-id
                | (arg ...+ . rest-id)

  arg = arg-id
        | [arg-id default-expr]
        | arg-keyword arg-id
        | arg-keyword [arg-id default-expr]
```



2. sintaksa za podajanje s
privzetimi vrednostmi

1. sintaksa za podajanje s
ključnimi besedami

kombinacija 1. in 2.

Imenovani argumenti in privzete vrednosti

- podajanje s ključnimi besedami

```
(define pozdrav  
  (lambda (#:ime ime #:voscilo voscilo)  
    (printf "~a~a~a~a" voscilo ", " ime "!")))
```

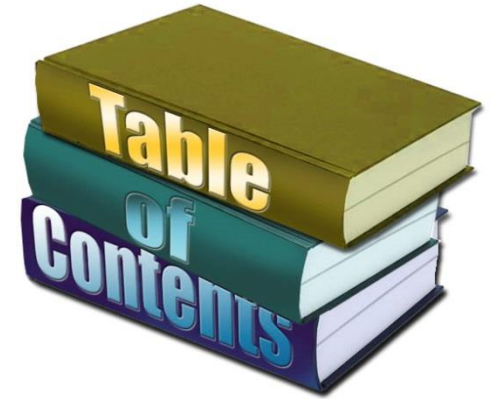
```
> (pozdrav #:ime "Helga" #:voscilo "Auf Wiedersehen")  
Auf Wiedersehen, Helga!  
> (pozdrav #:voscilo "Auf Wiedersehen" #:ime "Helga")  
Auf Wiedersehen, Helga!
```

- podajanje s ključnimi besedami in/ali privzetimi vrednostmi

```
(define mix  
  (lambda ([ime "Frodo"] #:starost [starost 32])  
    (printf "~a~a~a~a" ime " je star " starost " let.")))
```

```
> (mix)  
Frodo je star 32 let.  
> (mix "Jack")  
Jack je star 32 let.  
> (mix "Jack" #:starost 25)  
Jack je star 25 let.  
> (mix #:starost 25)  
Frodo je star 25 let.  
> (mix #:starost 25 "Janez")  
Janez je star 25 let.
```

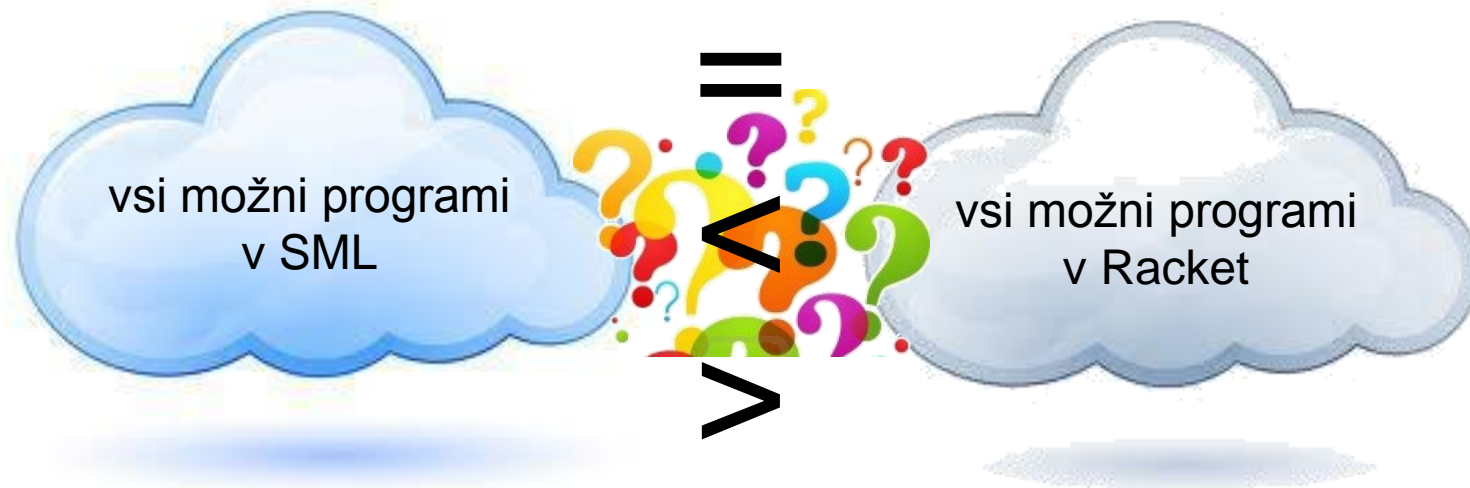
Pregled



- optimizacija ovojnic
- implementacija makro sistema v JAIS
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov

Primerjava ML in Racket

- razlike?
 - sintaksa
 - statična / dinamična tipizacija
 - ujemanje vzorcev / funkcije za preverjanje tipov in dostop do podatkov
- kakšna je relacija med številom veljavnih programov v obeh jezikih?



- zakaj?
 - statični tipizator zavrne programe, ki ne ustrezajo semantičnim pravilom

Primerjava ML in Racket

- pozabimo na razlike v sintaksi in premislimo, kako iz enega od jezikov gledamo na lastnosti drugega
- denimo da spodnja programa (oba sta veljavna v Racketu) implementiramo v SML

```
(define (fun1 x) (+ x (car x)))
```



v SML
neveljaven



```
(define (fun2 x) (if (> x 10)
                     #t
                     (list 1 "zivjo")))
```



v SML
neveljaven



- ➔ ML zavrača torej številne napačne programe (ki jih Racket ne zavrne), a na račun tega, da zavrne tudi programe, ki bi lahko bili veljavni

Statično preverjanje

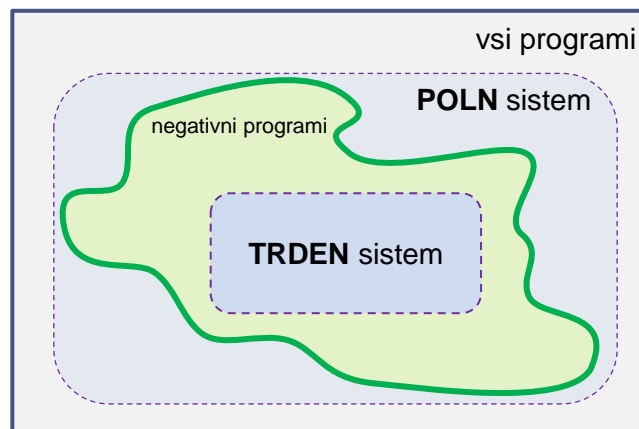


- **statično preverjanje** so postopki za zavrnitev nepravilnega programa, ki so izvedeni po uspešni razčlenitvi programa in pred njegovim zagonom
 - statično preverjanje:
 - pravilna uporaba aritmetičnih izrazov
 - pravilna semantika programskih konstruktov
 - nedefinirane spremenljivke
 - ujemanje vzorcev z vzorcem, ki se ponovi na dveh mestih
 - statično preverjanje NE obsega:
 - preverjanje, ali bo prišlo do izjeme
 - nepravilne aritmetične operacije (deljenje z 0)
 - preverjanje semantičnih napak
- **dinamično preverjanje**: postopki za zavrnitev nepravilnega programa, ki se izvajajo med izvajanjem programa

Trdnost in polnost sistema tipov

- terminologija:
 - pozitiven primer** programa: program, ki ima napako (+)
 - negativen primer** programa: program brez napake (-)
- sistem je TRDEN** (angl. *sound*), če nikoli ne sprejme pozitivnega programa
 - ima lažno pozitivne primere (= pravilni programi, ki jih zaznamo kot nepravilne)
- sistem je POLN** (angl. *complete*), če nikoli ne zavrne negativnega programa
 - ima lažno negativne primere (= nepravilni programi, zaznani kot pravilni)

analiziran kot program	P	N
P	PP	LN
N	LP	PN



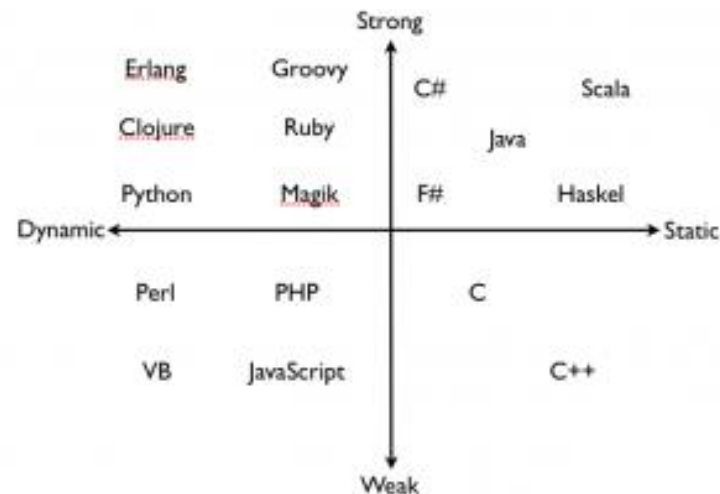
Zakaj nepolnost?

- sistem tipov v ML: trden, vendar ni poln
 - trdni/nepolni sistemi tipov so praksa
 - primer lažno pozitivnega primera (zavrneni primeri, ki ne povzročajo težav)?

```
fun f x = if true then 0 else 4 div "hello"
```

- problem izvajanje statične analize, ki ugotovi vse od naslednjega, je **NEODLOČLJIV** (matematični teorem):
 - ugotovi, ali se program vedno ustavi
 - je trdna
 - je polna
- v praksi izberemo 2 od 3:
 - odločimo se za statično analizo, ki preverja **ustavljivost in je trdna** (ne sprejme nepravilnih programov)
 - kaj, če bi se odločili za analizo, ki preverja ustavljivost in je polna (torej ni trdna, sprejme tudi nepravilne programe → sistem s šibkim tipiziranjem (weak typing))

Šibko tipiziranje



- šibko tipiziranje (angl. *weak typing*)
- sistem, ki:
 - je POLN (dopušča lažne negativne primere)
 - med izvajanjem se lahko zgodi napaka (potrebno preverjanje)
 - sistem izvaja malo statičnih ali dinamičnih preverjanj
 - rezultat: neznan?
 - C / C++
- prednosti šibko tipiziranih sistemov
 - "omogočajo večje programersko mojstrstvo?"
 - lažja implementacija programskega jezika (ni avtomatskih preverjanj)
 - večja učinkovitost (ni porabe časa za preverjanja; ni porabe prostora za oznake podatkovnih tipov)



FP v Pythonu

relacija med OUP in FP