

PROGRAMIRANJE

2014/15

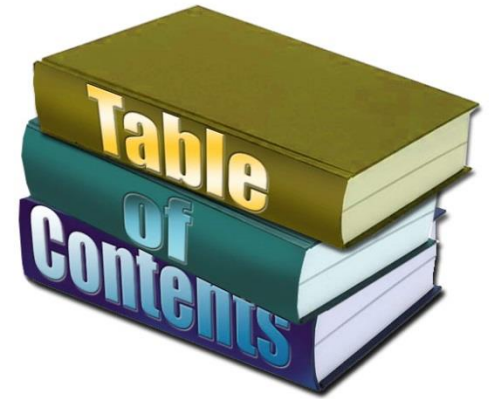
Racket

dinamično tipiziranje

takojšnja in zakasnjena evalvacija

Pregled

- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija
- zakasnitev in sprožitev
- tokovi
- memoizacija



Dinamično tipiziranje

- Racket pri prevajanju ne preverja podatkovnih tipov
- **slabost:** uspešno lahko prevede programe, pri katerih nato pride do napake pri izvajanju (če programska logika pripelje do dela kode, kjer se napaka nahaja)
- **prednost:** naredimo lahko bolj fleksibilne programe, ki niso odvisni od pravil sistema za statično tipiziranje
 - fleksibilne strukture brez deklaracije podatkovnih tipov (npr. sezname in pari)
 - primer spodaj

```
(define (prestej sez)
  (if (null? sez)
      0
      (if (list? (car sez))
          (+ (prestej (car sez)) (prestej (cdr sez)))
          (+ 1 (prestej (cdr sez))))))
```


```
> (prestej (list (list 1 2 (list #f) "lala") (list 1 2 3) 5))
8
```

Lokalno okolje


- različne vrste definiranja lokalnega okolja za različne potrebe

```
let      ; izrazi se evalvirajo v okolju PRED izrazom let  
let*    ; izrazi se evalvirajo kot rezultat predhodnih  
          deklaracij (tako dela SML)  
letrec  ; izrazi se evalvirajo v okolju, ki vključuje  
          vse podane deklaracije (vzajemna rekurzija)  
define ; semantika ekvivalentna kot pri letrec, le drugačna  
          sintaksa
```

- `let in let*`
- pozor: sintaksa `(let ([..]...[..]) (telo))`

```
(define (test-let a)  
  (let ([a 3]   
        [b (+ a 2)]))  
  (+ a b)))
```

```
> (test-let 10)  
15
```

```
(define (test-let* a)  
  (let* ([a 3]  
        [b (+ a 2)]))  
  (+ a b)))
```

```
> (test-let* 10)  
8
```

Lokalno okolje

- `letrec` in `define`: podobno kot vzajemna rekurzija v SML (operator `and`)
- pozor: izrazi se vedno evalvirajo v vrstnem redu, takrat morajo biti spremenljivke definirane; izjema so funkcije: telo se izvede šele ob klicu funkcije
- (globalne) deklaracije v programski datoteki se obnašajo kot `letrec`

```
(define (test-letrec a)
  (letrec ([b 3]
           [c (lambda (x) (+ a b d x))]
           [d (+ a 1)])
    (c a)))
```

```
> (test-letrec 50)
154
```



enakovredno

```
(define (test-define a)
  (define b 3)
  (define c (lambda (x) (+ a b d x)))
  (define d (+ a 1))
  (c a))
```

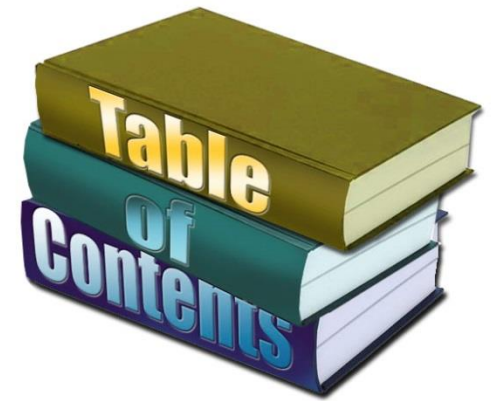
```
(define (test-letrec2 a)
  (letrec ([b 3]
           [c (+ d 1)]
           [d (+ a 1)])
    (+ a d)))
```

```
> (test-letrec2 50)
+: contract violation
  expected: number?
  given: #<undefined>
```

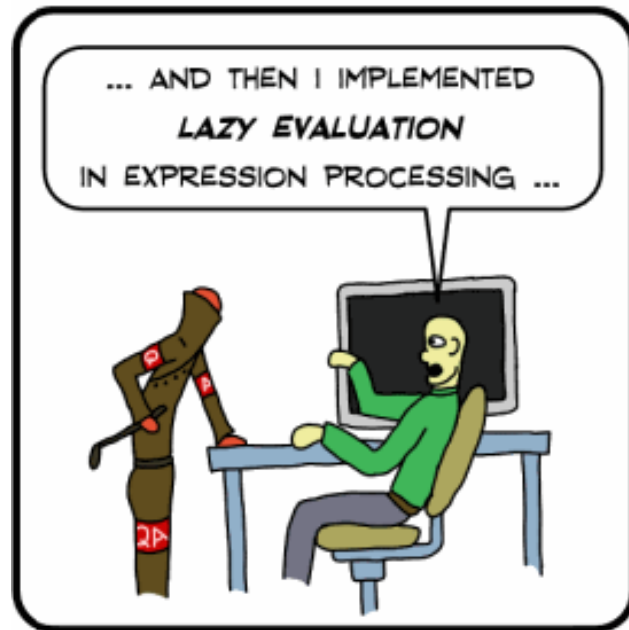


Pregled

- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija
- zakasnitev in sprožitev
- tokovi
- memoizacija



Pregled




Takojšnja in zakasnjena evalvacija

- semantika programskega jezika mora opredeljevati, kdaj se izrazi evalvirajo
- spomnimo se primera deklaracij `(define x e)`:
 - če je `e` aritmetični izraz, se ta evalvira takoj, v `x` se shrani rezultat (**takojšnja ali zgodnja evalvacija**, angl. *eager evaluation*)
 - če je `e` funkcija, torej `(lambda ...)`, se telo evalvira šele ob klicu `(x)` (**zakasnjena evalvacija**, angl. *delayed evaluation*)
- kako je s pogojnim stavkom `(if pogoj res nires)`?
Izraza `res` in `nires` se evalvirata šele po evalvaciji pogoja in vedno samo eden
- zakaj desni primer ne deluje?


```
; sintaksa:  
; (if pogoj res nires)
```

```
(define (potenca x n)  
  (if (= n 0)  
      1  
      (* x (potenca x (- n 1))))))
```



```
(define (moj-if pogoj res nires)  
  (if pogoj res nires))
```

```
(define (potenca-moj x n)  
  (moj-if (= n 0)  
          1  
          (* x (potenca-moj x (- n 1))))))
```



Takojšnja in zakasnjena evalvacija

- ideja:
 - če želimo zakasniti evalvacijo, zapišemo izraz v funkcijo (lahko brez parametrov)
 - `(lambda () e)`
 - kadar želimo izvesti evalvacijo izraza, funkcijo pokličemo
- angl. *thunking*

thunk functional programming

Web definitions

In computer science, a thunk is a parameterless closure created to prevent the evaluation of an expression until forced at a later time.

```
(define (moj-if-super pogoj res nires)
  (if pogoj (res) (nires)))

(define (potenca-super x n)
  (moj-if-super (= n 0)
    (lambda () 1)
    (lambda () (* x (potenca x (- n 1))))))
```

Zakasnjena evalvacija

- za zakasnitev evalvacije, kodo ovijemo v funkcijo brez parametrov (angl. *thunk*); evalvacija se izvede ob klicu funkcije,
- koristno je vedeti, kolikokrat se bo izraz evalviral

```
; izraz znotraj x se evalvira 0 krat  
(define (fun1 x)  
  (if #t "zivjo" (x)))
```

```
; izraz se evalvira 1 krat  
(define (fun2 x)  
  (if #f "zivjo" (x)))
```

```
; izraz se evalvira 0 do n krat  
(define (fun3 x)  
  (begin  
    (if pogoj1 xxx (x))  
    (if pogoj2 xxx (x))  
    (if pogoj3 xxx (x))  
    ...  
    (if pogojn xxx (x))))
```

ponavljanje
iste
evalvacije

```
; izraz se evalvira 1 krat  
(define (fun4 x)  
  (let* ([t (x)])  
    (begin  
      (if pogoj1 xxx t)  
      (if pogoj2 xxx t)  
      ...  
      (if pogoj xxx t))))
```

kaj pa, če to
sploh ni
potrebno?

- ideja: izvedimo **leno evalvacijo** – naredimo mehanizem, ki evalvira izraz takrat, ko ga prvič potrebujemo. Pri nadaljnjih klicih vrnemo že evalvirano vrednost (izraz torej evalviramo natanko enkrat).

Potrebovali bomo...

- zaporedje izrazov
 - zaporedje vrne vrednost zadnjega izraza v zaporedju

```
(begin e1 e2 ... en)
```

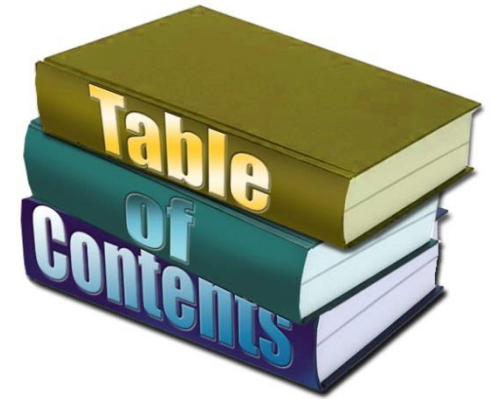
- par, katerega komponente lahko spreminjamo
 - `cons` ne podpira mutacije
 - novi konstruktor `mcons` (mutable cons)

```
mcons      ; konstruktor  
mcar       ; glava  
mcdr       ; rep  
mpair?     ; je par?  
set-mcar!  ; nastavi novo glavo  
set-mcdr!  ; nastavi novi rep
```

- funkcij za navadne pare (`cons`) ne moremo uporabljati na `mcons`

Pregled

- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija
- zakasnitev in sprožitev
- tokovi
- memoizacija



Zakasnitev in sprožitev

- zakasnitev (angl. *delay*), sprožitev (angl. *force*)
- mehanizem je že vgrajen v Racket (mi ga sprogramiramo sami)
- *delay* prejme zakasnitveno funkcijo in vrne par s komponentama:
 - bool: indikator, ali je izraz že evalviran
 - zakasnitvena funkcija ali evalviran izraz

```
; ZAKASNITEV
(define (my-delay thunk)
  (mcons #f thunk))
```

```
; SPROŽITEV
(define (my-force prom)
  (if (mcar prom)
      (mcd r prom)
      (begin (set-mcar! prom #t)
              (set-mcdr! prom ((mcd r prom)) )
              (mcd r prom)))))
```

```
> (define md
    (my-delay
     (lambda () (+ 3 2))))
```

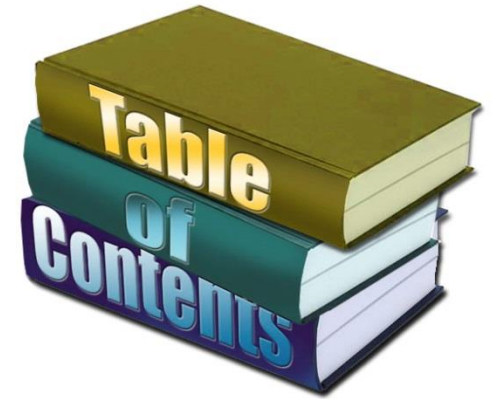
```
> md
(mcons #f #<procedure>)
```

```
> (my-force md)
5
```

```
> md
(mcons #t 5)
```

Pregled

- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija
- zakasnitev in sprožitev
- tokovi
- memoizacija

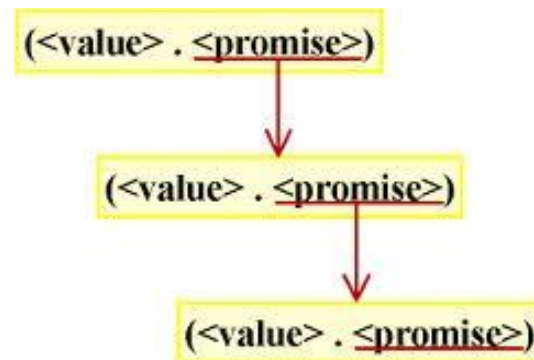


Tokovi

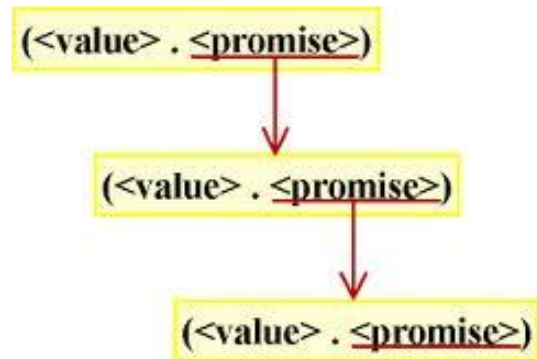
- tok: neskončno zaporedje vrednosti (npr. naravna števila), ki ga ne moremo definirati s podajanjem vseh vrednosti
- ideja: podajmo le naslednjo vrednost in zakasnimo evalvacijo (*thunk*) za izračun naslednje vrednosti
- definirajmo tok kot par

```
'(vrednost . funkcija-za-naslednji)
```

- v paru:
 - zakasnjena funkcija (*thunk*) generira naslednji element v zaporedju, ki je tudi par enake oblike,
 - zakasnjena funkcija lahko vsebuje tudi rekurzivni klic, ki se izvede šele ob klicu funkcije



Tokovi



- dostop do elementov:

```
(car s) ; prvi element  
(car ((cdr s))) ; drugi element  
(car ((cdr ((cdr s))))) ; tretji element
```


Primeri



Definiraj naslednje tokove:

1. zaporedje samih enic
2. zaporedje naravnih števil
3. zaporedje $1, -1, 1, -1, \dots$
4. zaporedje potenc števila 2

Zapiši funkcije za delo s tokovi:

1. izpiši prvih n števil v toku
2. izpisuj tok, dokler velja *pogoj*
3. izpiši, koliko števil je v toku, preden velja *pogoj*



Makro sistem