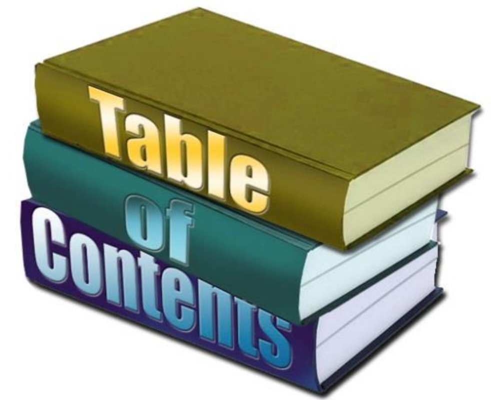
FUNKCIJSKO PROGRAMIRANJE

2023/24

Racket
dinamično tipiziranje
lokalno okolje
zakasnjena evalvacija
memoizacija
makro sistem

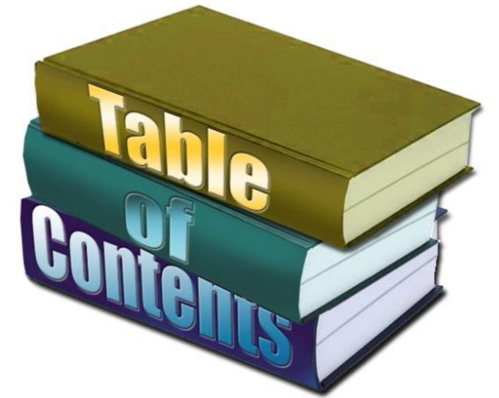
© Zoran Bosnić

Pregled snovi do sedaj



- paradigme programiranja (funkcijsko, objektno, usmerjeno, …)
- sintaksa, semantika, preverjanje tipov
- podatkovni tipi (seznami, terke, zapisi, opcije)
- lokalno okolje, vezave, senčenje
- sinonimi za podatkovne tipe
- deklaracija lastnih alternativnih in rekurzivnih tipov
- ujemanje vzorcev (tudi rekurzivno)
- polimorfizem
- izjeme
- repna rekurzija, funkcijski sklad
- funkcije višjega reda (kot argumenti ali rezultat funkcij), map/filter/fold
- leksikalni doseg, funkcijske ovojnice, dinamični doseg
- currying, delna aplikacija
- statično/dinamično tipiziranje, implicitno/eksplicitno tipiziranje
- mutacija
- določanje podatkovnih tipov, omejitev vrednosti
- vzajemna
- moduli (organizacija in skrivanje programske kode)

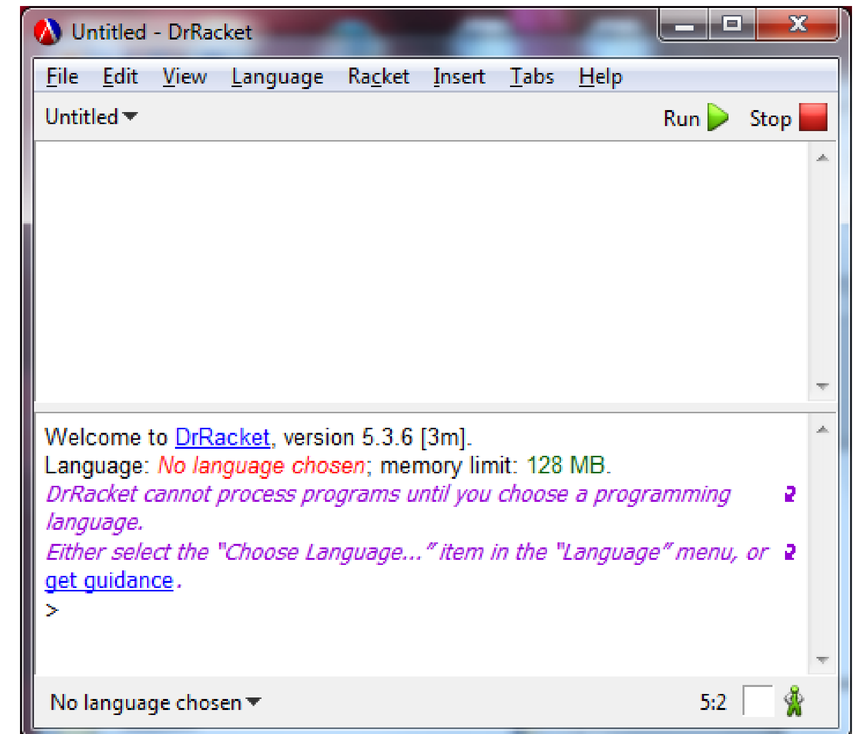
Pregled



- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem

Racket

- Literatura: The Racket Guide, http://racket-lang.org/
- tudi funkcijski jezik
  - vse je izraz, ovojnice, anonimne funkcije, currying
  - je dinamično tipiziran: uspešno prevede več programov, vendar se večina napak zgodi šele pri izvajanju
- primeren za učenje novih konceptov:
  - zakasnjena evalvacija
  - tokovi
  - makri
  - memoizacija
- naslednik jezika Scheme
- razvojno okolje: DrRacket
  - koda in REPL



Oklepaji

- veliko jih je ☺
- primerjava z značkami v sintaksi HTML
- imajo poseben pomen: niso namenjeni samo prioriteti izračunov
- uporabljamo lahko tudi `[]` namesto `()`, morajo biti v pravih parih

Oklepaji

- različni pomeni izrazov v odvisnosti od oklepajev:

```
e      ; izraz
(e)    ; klic funkcije e, ki prejme 0 argumentov
((e))  ; klic rezultata funkcije e, ki prejme 0 argumentov
```

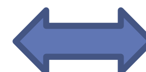
```
(define (potenca x n)
 (if (= n 0)
   1
   (* x (potenca x (- n 1)))))
```

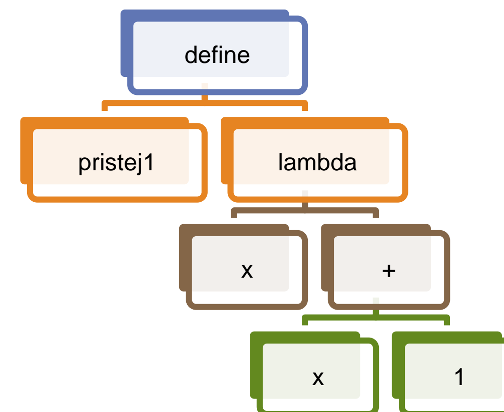
≠

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

- omogočajo nedvoumno sintakso (opredeljujejo prioriteto operatorjev) in predstavitev v drevesni obliki (razčlenjevanje)

```
(define pristej1
  (lambda (x)
    (+ x 1)))
```





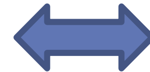
Osnove

- modul je zbirka deklaracij
- `#lang racket` na vrhu datoteke
- deklaracija

```
(define x "Hello world")
```

- deklaracija funkcije z besedo `lambda` (ali sintaktična olepšava)

```
(define sestej1
  (lambda (a b)
    (+ a b)))
```



```
(sintaktična olepšava)
(define (sestej2 a b)
  (+ a b))
```

- stavek `if`

```
(if pogoj ce_res ce_nires)
```

- currying

```
(define potenca2
  (lambda (x)
    (lambda (n)
      (potenca x n))))
```

Osnove

- **izrazi**
  - atomi (konstante in imena spremenljivk): `3.14, 5, #t, #f, x, y`
  - rezervirane besede: `lambda`, `if`, `define`
  - zaporedja izrazov v oklepajih `(e1 e2 … en)`
    - `e1` je lahko rezervirana beseda ali ime funkcije

- **logične vrednosti**
  - `#t` in `#f`
  - vse vrednosti, ki niso `#f`, se obravnavajo kot `#t`
    (to v statično tipiziranih jezikih ni možno!)

```
> (if "lala" "DA" "NE")
"DA"
> (if null "DA" "NE")
"DA"
> (if "" "DA" "NE")
"DA"
> (if 0 "DA" "NE")
"DA"
> (if #f "DA" "NE")
"NE"
```

Seznami in pari

- seznami in pari se tvorijo z istim konstruktorjem (`cons`) ← prednost dinamično tipiziranega jezika (ne potrebujemo ločenih konstruktorjev, ki že pri prevajanju nakazujejo na pravilni tip podatka)

```
cons  ; konstruktor
null  ; prazen "element" (seznam)
null? ; ali je seznam prazen?
car   ; glava
cdr   ; rep
; funkcija za tvorjenje seznama
(list e1 e2 ... en)
```

- konstruktor `cons` oblikuje par (lahko je gnezden – potem par postane terka)
- seznam je samo posebna oblika para/terke, ki ima na najbolj vgnezdenem mestu null

```
> (cons "a" 1)
'("a" . 1)                ; par
> (cons "a" (cons 2 (cons #f 3.14)))
'("a" 2 #f . 3.14)        ; terka
> (cons "a" (cons 2 (cons #f (cons 3.14 null))))
'("a" 2 #f 3.14)          ; seznam
> (list "a" 2 #f 3.14)
'("a" 2 #f 3.14)          ; enak seznam (lepše)
```

Seznami in pari

- razpoznavanje seznama (angl. *proper list*) in parov (angl. *pair*)

```
(list? e)   ; vrne #t, če je e seznam
(pair? e)   ; vrne #t, če je e seznam ali
              par (karkoli narejenega s cons)
```

- kdaj uporabiti par in kdaj seznam?
    - podobno razmišljanje kot pri terkah/seznamih
    - par: hiter zapis števila elementov fiksnega tipa
    - seznam: zapis večjega števila elementov nedorečene velikosti
- dostop do elementov seznama

```
(define p1 (cons "a" 1))
(define p2 (cons "a" (cons 2 (cons #f 3.14))))
(define l1 (cons "a" (cons 2 (cons #f null))))
(define l2 (cons "a" (cons 2 (cons #f (cons 3.14 null)))))
(define l3 (list "a" 2 #f 3.14))
```

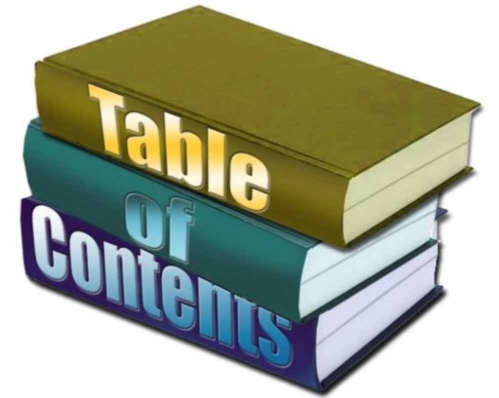
```
> sez
'("a" 2 #f 3.14)
> (car sez)
"a"
> (cdr sez)
'(2 #f 3.14)
> (car (cdr (cdr (cdr sez))))
3.14
```

Primeri

Napiši funkcije za delo s seznami:

1. `seštej elemente v seznamu`
2. `preštej elemente v seznamu`
3. `združi seznam`
4. `odstrani prvo pojavitev elementa v seznamu`
5. `odstrani vse pojavitve elementa v seznamu`
6. `vrni n-ti elementi`
7. `vrni vse elemente razen n-tega`
8. `map`
9. `filter`
10. `foldl (reduce)`

Pregled



- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
    - zakasnitvena funkcija
    - zakasnitev in sprožitev
    - tokovi
- memoizacija
- makro sistem

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. seznami 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
```

Pogojni stavek cond

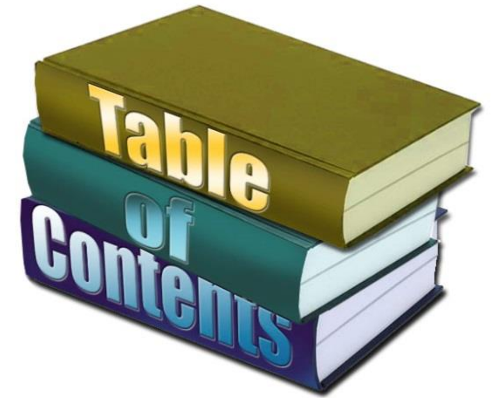
- boljši stil namesto vgnezdenih `if` stavkov

```
(cond  [pogoj1 e1]
       [pogoj2 e2]
        …
       [pogojN eN])
```

- semantika: če velja `pogoj1`, evalviraj izraz `e1` itd.
- oglati oklepaji so le konvencija, niso obvezni (lahko so okrogli)
- smiselno je, da je `pogojN = #t` ("globalni" else)

```
(define (prestej1 sez)
  (cond [(null? sez) 0]
        [(list? (car sez)) (+ (prestej1 (car sez)) (prestej1 (cdr sez)))]
        [#t (+ 1 (prestej1 (cdr sez)))]))
```

Pregled



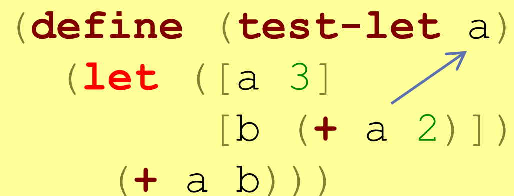
- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
    - zakasnitvena funkcija
    - zakasnitev in sprožitev
    - tokovi
- memoizacija
- makro sistem

Lokalno okolje

- različne vrste definiranj lokalnega okolja za različne potrebe

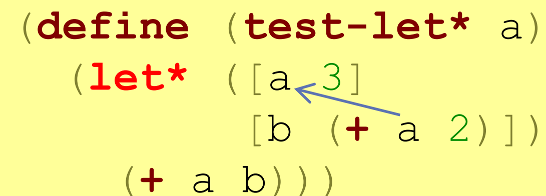
```
let      ; izrazi se evalvirajo v okolju PRED izrazom let
let*     ; izrazi se evalvirajo kot rezultat predhodnih
           deklaracij (tako dela SML)
letrec   ; izrazi se evalvirajo v okolju, ki vključuje
           vse podane deklaracije (vzajemna rekurzija)
define   ; semantika ekvivalentna kot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

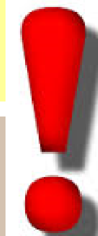
```
> (test-letrec 50)
154
```

enakovredno

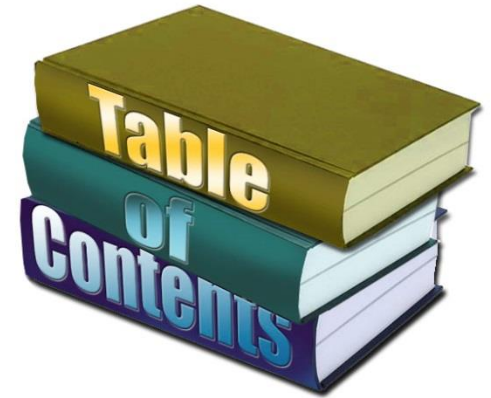
```
(define (test-define a)
  (define b 3)
  (define c (lambda (x) (+ a b d x)))
  (define d (+ a 1))
  (c a))
```

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

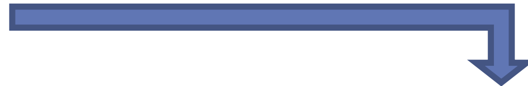
```
> (test-letrec2 50)
; d: undefined;
; cannot use before
initialization
```

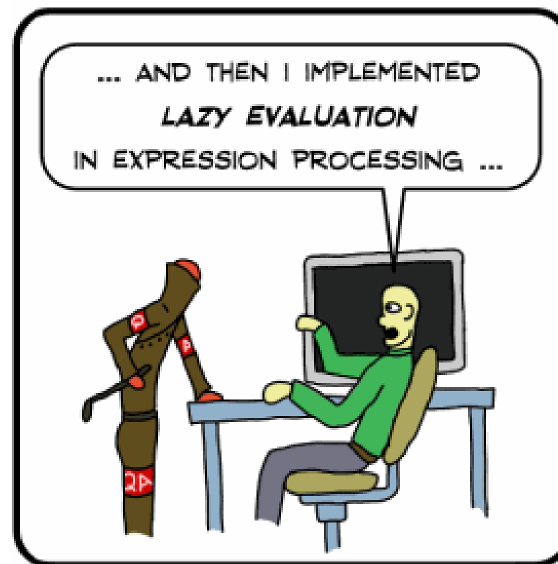


Pregled



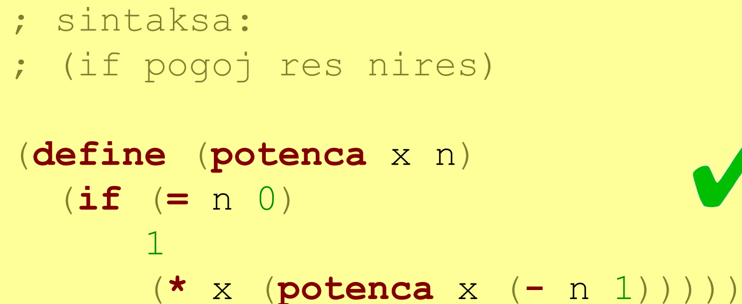
- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem





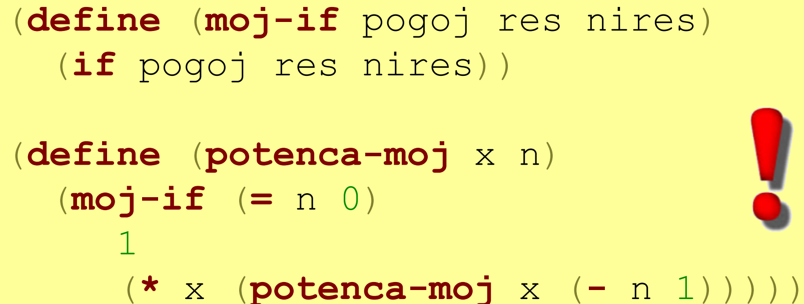
Takojšnja in zakasnjena evalvacija

- 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 ob vezavi, 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-super 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))))
```

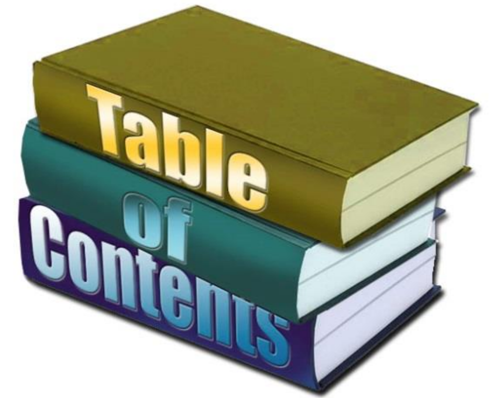
ponavljanje iste evalvacije

```
; 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 največ enkrat – in sicer le v primeru potrebe po vrednosti).

Pregled



- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem

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

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)
   (mcdr prom)
   (begin (set-mcar! prom #t)
          (set-mcdr! prom ((mcdr prom)))
          (mcdr prom))))
```

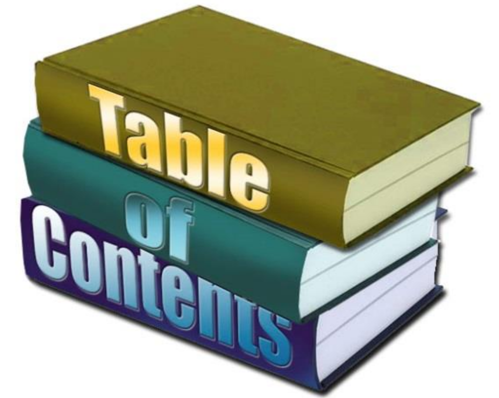
```
> (define md
   (my-delay
   (lambda () (+ 3 2))))

> md
(mcons #f #<procedure>)

> (my-force md)
5

> md
(mcons #t 5)
```

Pregled



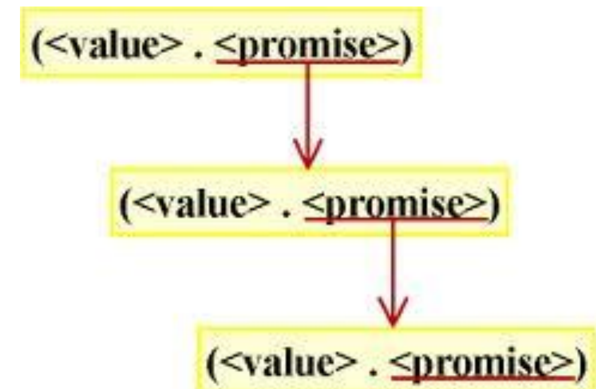
- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem

Tokovi

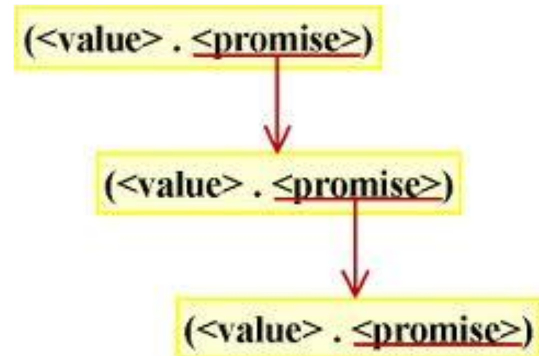
- tok: neskončno zaporedje vrednosti (npr. naravna števila), ki ga ne moremo definirati s podajanjem vseh vrednosti
- ideja: podajmo le (trenutno) vrednost in zakasnimo evalvacijo (*thunk*) za izračun naslednje vrednosti
- definirajmo tok kot par

```
'(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

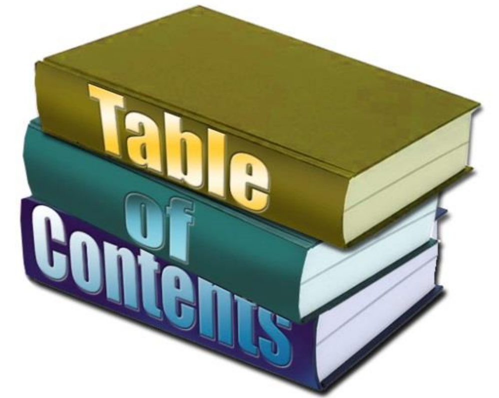
Definiraj naslednje tokove:

1. `zaporedje samih enic`
2. `zaporedje naravnih števil`
3. `zaporedje 1, -1, 1, -1, …`
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*

Pregled

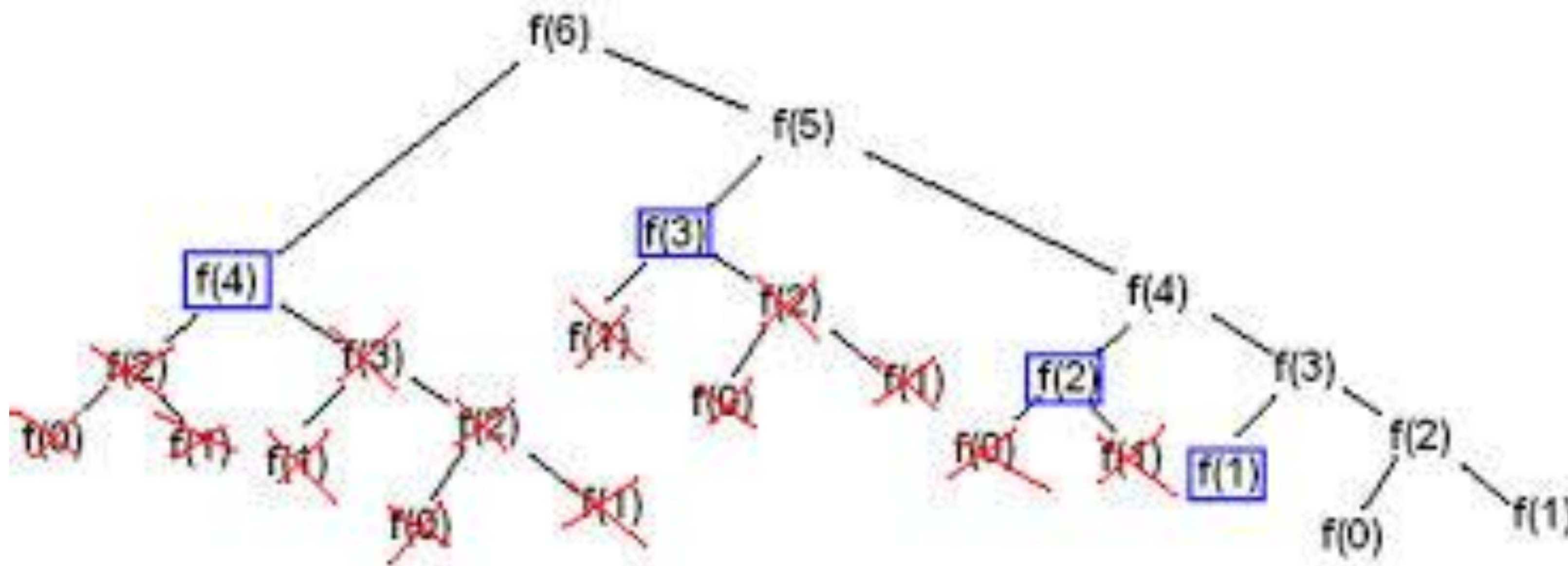


- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem

Memoizacija

- če funkcija pri istih argumentih vsakič vrača isti odgovor (in nima stranskih učinkov), lahko shranimo odgovore za večkratno rabo
- smotrnost?
  - ali je shranjevanje hitrejše od ponovnega računanja?
  - ali bodo shranjeni rezultati kdaj uporabljeni?

- primer: Fibonaccijeva števila, poenostavitev eksponentne časovne zahtevnosti?

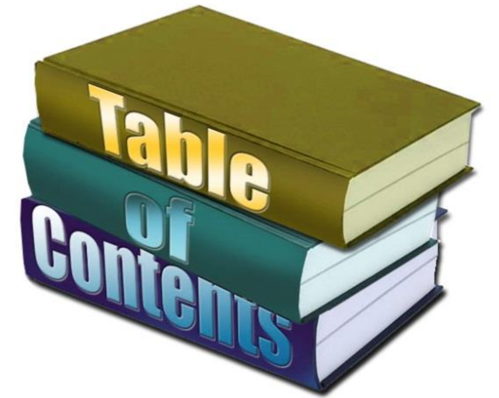


Memoizacija

- implementacija:
  - **uporabimo seznam** parov dosedanjih rešitev `'((arg1, odg1), …, (argn, odgn))`
    - ne želimo, da je globalno dostopen
    - ne sme biti v rekurzivni funkciji, ker bo spraznil z vsakim klicem
  - **če rešitev obstaja**, jo beremo iz seznama
    - pomagamo si lahko z vgrajeno funkcijo `assoc`
  - **če rešitve še ni**, jo izračunamo → dopolnimo seznam rešitev
    - za dopolnitev seznama potrebujemo mutacijo (`set!`)

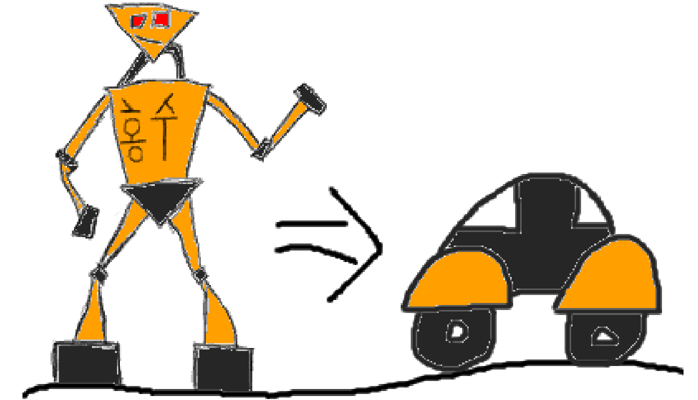
```
(define fib3
  (letrec ([resitve null]
           [pomozna (lambda (x)
                      (let ([ans (assoc x resitve)])                 ; poiscemo resitev
                        (if ans
                            (cdr ans)                                ; vrnemo obstojeco resitev
                            (let ([nova (cond [(= x 1) 1]            ; resitve ni
                                              [(= x 2) 1]
                                              [#t (+ (pomozna (- x 1))        ; izracun resitve
                                                     (pomozna (- x 2)))])])
                              (begin
                                (set! resitve (cons (cons x nova) resitve))   ; shranimo resitev
                                nova)))))])                                   ; vrnemo resitev
    pomozna))
```

Pregled



- uvod v Racket
- dinamično tipiziranje
- lokalno okolje
- zakasnjena evalvacija
  - zakasnitvena funkcija
  - zakasnitev in sprožitev
  - tokovi
- memoizacija
- makro sistem

Makri



- **makro** definira, kako sintakso v programskem jeziku preslikamo v drugo sintakso
    - orodje, ki ga ponuja programski jezik
    - razširitev jezika z novimi ključnimi besedami
    - implementacija sintaktičnih olepšav
- programski jeziki (Racket, C, …) imajo posebno sintakso za definiranje makrov
- postopek razširitve makro definicij (angl. *macro expansion*) se izvede pred prevajanjem in izvajanjem programa
- primeri:
    - lasten stavek if: `(moj-if pogoj then e1 else e2)`
    - trojni if: `(if3 pog then e1 elsif pogoj2 then e2 else e3)`
    - elementi toka: `(prvi tok)`, `(drugi tok)`, `(tretji tok)`
    - komentiranje spremenljivk: `(anotiraj xyz "trenutni stevec")`

Definicija makrov

- rezervirana beseda `define-syntax`
- preostale ključne besede opredelimo s `syntax-rules`
- v `[ … ]` podamo vzorce za makro razširitev
- primeri:

```
(define-syntax if-trojni
  (syntax-rules (then elsif else)
    [(if-trojni e1 then e2 elsif e3 then e4 else e5)
     (if e1 e2 (if e3 e4 e5))]))
```

```
(define-syntax tretji
  (syntax-rules ()
    [(tretji e)
     (car ((cdr ((cdr e)))))]))
```

```
(define-syntax anotiraj
  (syntax-rules ()
    [(anotiraj e s)
     e]))
```

- lastnosti:
  - definiramo lahko lastne rezervirane besede (`then`, `elsif`)
  - možne sintaktične napake:
    - pri uporabi sintakse za makro
    - pri uporabi sintakse, v katero se makro razširi

Lastnosti makrov

- makro zamenjuje ključne besede (sintaksne žetone) in ne posameznih črk (torej pravilo "or → uta" ne naredi zamenjave v izrazih "(+ c minor)" → "(+ c minuta)"

- **posebno pozornost je potrebno posvetiti:**
  1. ali je makro sploh potreben (morda zadošča funkcija)?
  2. prioriteta izračunanih izrazov
  3. način evalvacije izrazov v makrih
  4. semantika dosega spremenljivk; uporabljamo dve okolji:
     - okolje v definiciji makra,
     - okolje, kjer se makro razširi v programsko kodo

1. Makro: primernost uporabe

- primer: `my-delay` in `my-force`
- pri `my-delay` smo morali podati zakasnjeno funkcijo (*thunk*):

```
(my-delay (lambda () (+ 3 2)))
```

- denimo, da želimo ta zapis poenostaviti v zapis brez besede lambda ():

```
(my-delay (+ 3 2))
```

- brez makrov ne obstaja način, da ta zapis poenostavimo, saj se argumenti evalvirajo takoj ob klicu funkcije!
- rešitev: uporabimo makro

```
(define-syntax my-delaym
  (syntax-rules ()
    [(my-delaym e)
     (mcons #f (lambda() e))]))
```

- `my-force` nima implementacijskih težav, primeren je v obliki funkcije
  - pravzaprav: makro ne bi deloval, kot želimo (o tem malo kasneje)! ☺

2. Makro: prioriteta izračunov

- primer makra v C++:
  `#define ADD(x,y) x+y`
  ta makro opravi zamenjavo izraza:
  `ADD(1,2)*3` → `1+2*3`
  (rešitev je 7 in ne morda 9)
- za pravilno delovanje moramo makro definirati kot
  `#define ADD(x,y) ((x)+(y))`

- Racket teh težav nima, ker uporabljamo **prefiksno notacijo**, ki jasno **opredeljuje prioriteto** operacij
- primer: makro `(sestej a b)` → `(+ a b)`
  pravilno opravi razširitev izraza
  `(* (sestej 1 2) 3)` → `(* (+ 1 2) 3)`

3. Makro: način evalvacije izrazov

- potrebno je posvetiti pozornost temu, kolikokrat se določen izraz evalvira
- primer makrov, ki nista ekvivalentna

```
(define-syntax dvakrat3
  (syntax-rules ()  [(dvakrat3 x)(+ x x)]))
```

```
(define-syntax dvakrat4
  (syntax-rules ()[(dvakrat4 x)(* 2 x)]))
```

- večkratne evalvacije lahko preprečimo z uporabo lokalnih spremenljivk (stavek `let`)

```
(define-syntax dvakrat5
  (syntax-rules()
    [(dvakrat5 x)(let ([mojx x]) (+ mojx mojx))]))
```

4. Makro: semantika dosega

- kaj se zgodi, če makro uporablja iste spremenljivke, ki nastopajo že v funkciji?
- naivna makro razširitev (uporabljata jo C/C++; je enakovredna `find&replace`) lahko povzroči nepričakovane rezultate
- primer:

```
(define-syntax swap
  (syntax-rules ()
    ((swap x y)
     (let ([tmp x])
       (set! x y)
       (set! y tmp)))))
```

```
> (let ([tmp 5]
        [other 6])
    (let ([tmp tmp])
      (set! tmp other)
      (set! other tmp))
    (list tmp other))
'(5 6)
```

naivna makro razširitev klica `(swap tmp other)`

4. Makro: semantika dosega

- v sistemih z naivnimi razširitvami se to rešuje z uporabo redkih imen spremenljivk (čudna imena, samo velike črke)

- vendar pa makro definicije tudi uporabljajo leksikalni doseg
  - uporaba vrednosti spremenljivk v kontekstu, kjer je makro definiran
  - samodejno preimenovanje lokalnih spremenljivk

higiena makro sistema

```
> (let [tmp 5]
       [other 6])
   (let ([tmp tmp])
     (set! tmp other)
     (set! other tmp))
   (list tmp other))
'(5 6)
```

naivna makro razširitev

```
> (let ([tmp 5]
        [other 6])
   (swap tmp other)
   (list tmp other))
'(6 5)
```

Racket

Lastni podatkovni tipi,
interpreter