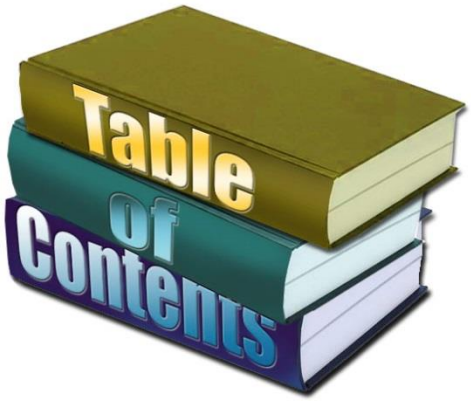
# PROGRAMIRANJE
# 2014/15

*vzajemna rekurzija*

*moduli*

*Racket*

# Pregled




- vzajemna rekurzija
- moduli
- uvod v Racket
- seznami in pari

# Vzajemna rekurzija

- omogočati uporabo funkcij in podatkovnih tipov, ki so deklarirani za trenutno deklaracijo

```
fun fun1 par1 = <telo>
and fun2 par2 = <telo>
and fun3 par3 = <telo>
```

```
datatype tip1 = <definicija>
and tip2 = <definicija>
and tip3 = <definicija>
```

- primer:

```
fun sodo x =
    if x=0
    then true
    else liho (x-1)
and liho x =
    if x=0
    then false
    else sodo (x-1)
```

- v praksi uporabno za opisovanje stanj končnih avtomatov

# Vzajemna rekurzija

- izpitna naloga 2013/14

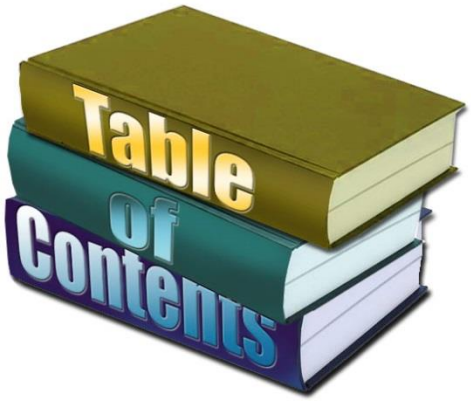
V jeziku SML napiši program check, ki preverja pravilnost vhodnega seznama `sez`. Za vhodni seznam morajo veljati naslednja pravila:

- program naj sprejme prazen seznam,
- seznam hrani vrednosti podatkovnega tipa `datatype datum = A of int | B of int list`
- v seznamu se izmenjujeta podatka, narejena s konstruktorjem `A` in konstruktorjem `B`,
- seznam se mora obvezno začeti z elementom, ki je narejen s konstruktorjem `A` in se lahko konča s poljubnim elementom (konstruktor `A` ali `B`),
- seznami tipa `int list`, ki so argument konstruktorja `B`, vsebujejo elemente z vrednostima 3 in 4,
- seznami tipa `int list`, ki so argument konstruktorja `B`, se morajo vedno končati na 4, njihov začetek pa ni pomemben.

Primeri/:

```
- check [A 1, B [3,4], A 3];
val it = true : bool
- check [A 9, B [3,4], A 4, B [4,3,4,3,4], A 2, B [4]];
val it = true : bool
- check [B [3,4], A 1, B [4,3]];
val it = false : bool        (* has to start with A *)
- check [A 1, B [3,4,3]];
val it = false : bool        (* list given with B has to end with 4 *)
```

# Pregled




- vzajemna rekurzija
- moduli
- uvod v Racket
- seznami in pari

# Moduli

- omogočajo:
  - organiziranje programske kode v smiselne celote
  - preprečevanje senčenja (isto ime je lahko deklarirano v več modulih)
- znotraj modula se sklicujemo na deklarirane objekte enako kot smo se v prej v "zunanjem" okolju (brez posebnosti)
- iz "zunanjega" okolja se na deklaracije v modulu sklicujemo z uporabo predpone "*ImeModula.ime*"
- sintaksa za deklaracijo modula:

```
structure MyModule =
struct
   <deklaracije val, fun, datatype, …>
end
```

```
structure Nizi =
struct
  val prazni_niz = ""
  fun prvacrka niz =
      hd (String.explode niz)
end
```

# Javno dostopne deklaracije

- modulu lahko določimo, katere deklaracije so na razpolago "javnosti" in katere so zasebne (*public* in *private* v Javi?)
- seznam javnih deklaracij strnemo v *podpis modula* (signature), nato podpis pripišemo modulu

```
signature PolinomP =
sig
    datatype polinom = Nicla | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val mnozi : polinom -> int -> polinom
    val izpisi : polinom -> string
end
```

deklaracija podpisa

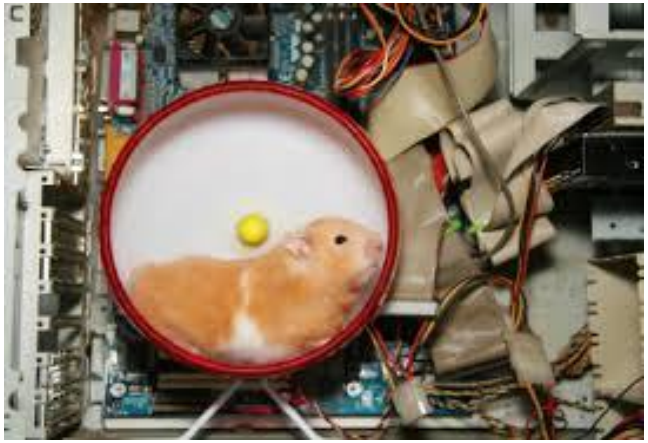
```
structure Polinom :> PolinomP =
struct
    ... deklaracije ...
end
```

podpis pripišemo modulu; uporabimo operator :>

- v podpisu določimo samo podatkovne tipe deklaracij (type, datatype, val, exception)
- podpis mora biti skladen z vsebino modula, sicer preverjanje tipov ne bo uspešno

# Skrivanje implementacije

- uporaba podpisov modulov je koristna, ker z njim skrivamo implementacijo, kar je lastnost dobre in robustne programske opreme!
- s skrivanjem implementacije dosežemo:
  1. uporabnik ne pozna načina implementacije operacij; lahko jo tudi kasneje spremenimo brez vpliva na preostalo kodo,
  2. uporabniku onemogočimo, da uporablja modul na napačen način



# Primer

Specifikacija lastnosti našega modula za delo s polinomi:

- za izdelavo novega polinoma se uporablja funkcija `novipolinom`
- funkcija za množenje ni vidna navzven, je na razpolago (morebitnim) internim funkcijam
- koeficienti polinoma so zapisani v padajočem vrstnem redu glede na potenco neodvisne spremenljivke
- vse potence neodvisne spremenljivke so pozitivne
- če je koeficient enak 0, ga ne izpišemo

```
signature PolinomP =
sig
    datatype polinom = Nicla | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val mnozi : polinom -> int -> polinom
    val izpisi : polinom -> string
end
```

ali ta podpis sedaj ustreza zgornji specifikaciji?

žal ne… poglejmo si primer

# Skrivanje podrobnosti

- uporabnik lahko kvari delovanje, predvideno v specifikaciji (glej primer)

```
signature PolinomP =
sig
    datatype polinom = Nicla
         | Pol of (int * int) list
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

**KORAK 1:**
**skrijemo funkcijo za množenje**

```
signature PolinomP2 =
sig
    type polinom
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

**KORAK 2:**
**definiramo abstraktni podatkovni tip, ki ne razkriva podrobnosti implementacije uporabniku:**

- skrijemo, da je polinom datatype
- uporabnik še vedno lahko računa s polinomi

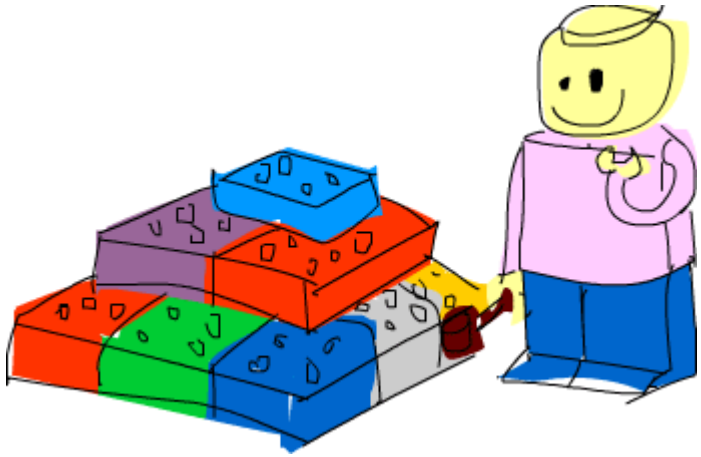
```
signature PolinomP3 =
sig
    type polinom
    val Nicla : polinom
    val novipolinom : int list -> polinom
    val izpisi : polinom -> string
end
```

**KORAK 3:**
**vendar pa ni nič narobe, če razkrijemo samo del podatkovnega tipa (vrednost Nicla) in skrijemo samo konstruktor Pol**

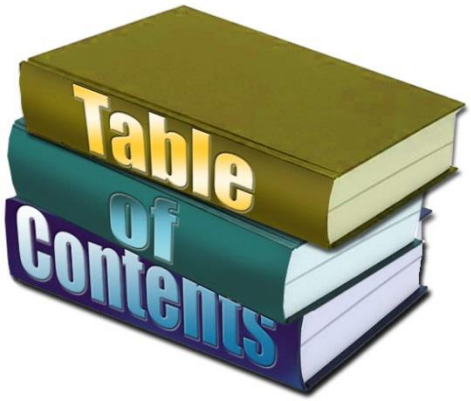
# Ustreznost modula in podpisa

Podpis lahko uspešno pripišemo modulu (`Modul :> podpis`), če velja:

1. modul vsebuje vse ne-abstraktne tipe, navedene v podpisu
2. modul vsebuje implementacijo abstraktnih tipov iz podpisa (`datatype`, `type`)
3. vsaka deklaracija vrednosti (`val`) v podpisu se nahaja v modulu (v modulu je lahko bolj splošnega tipa)
4. vsaka izjema (`exception`) v podpisu se nahaja tudi v modulu



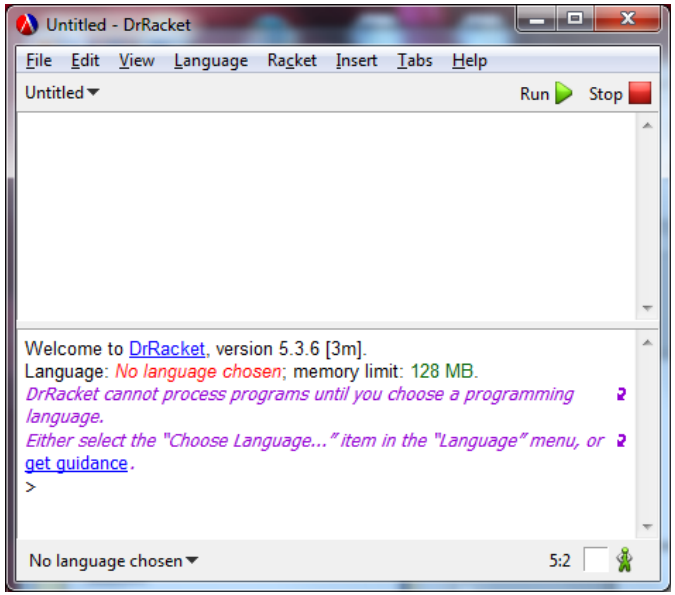
# Pregled




- vzajemna rekurzija
- moduli
- uvod v Racket
- seznami in pari

# 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


```
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)))))
```

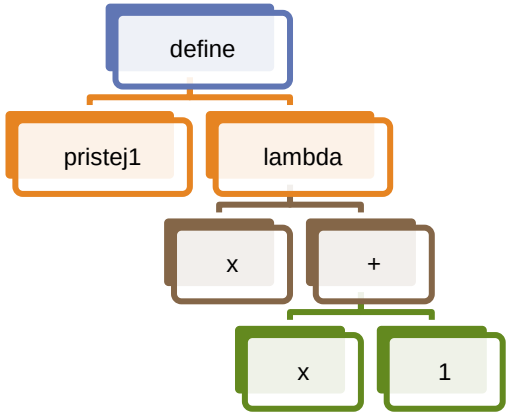
≠

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

# Oklepaji

- uporabljamo lahko tudi `[]` namesto `()`, morajo biti v pravih parih
- 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

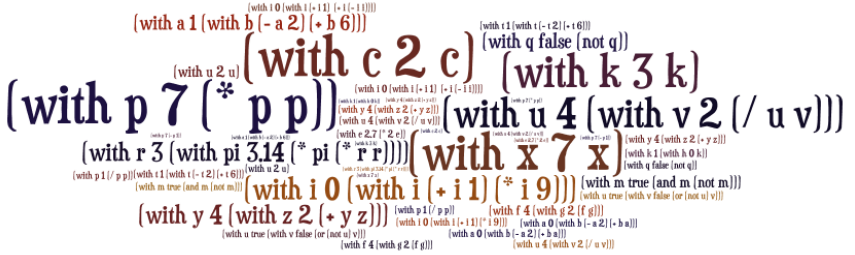
```
(define potenca2
  (lambda (x)
    (lambda (n)
      (potenca x n))))
```

# Osnove

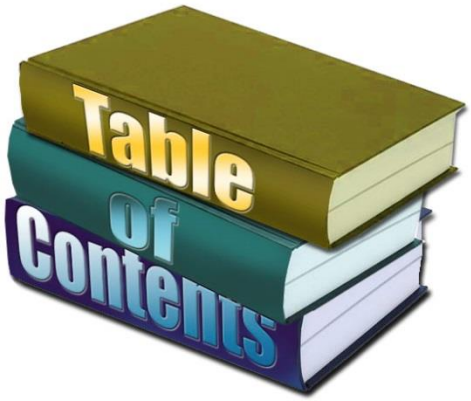
- **izrazi**
  - atomi (konstante in imena spremenljivk): `3.14, 5, #t, #f, x, y`
  - rezervirane besede: `lambda`, `if`, `define`
  - zaporedja izrazov v oklepajih `(e1 e2 … en)`
    - `e1` je lahko rezervirana beseda ali ime funkcije

- **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"
```



# Pregled




- vzajemna rekurzija
- moduli
- uvod v Racket
- seznami in pari

# 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))
```

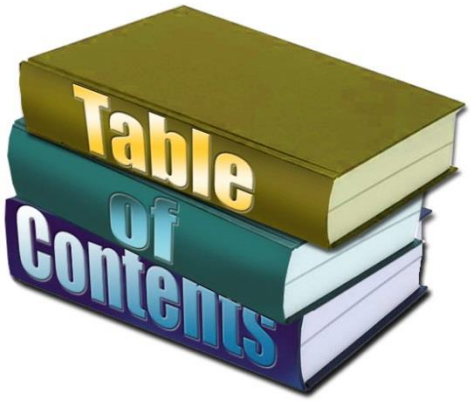
```
> sez
'("a" 2 #f 3.14)
> (car sez)
"a"
> (cdr sez)
'(2 #f 3.14)
> (car (cdr (cdr (cdr sez))))
3.14
```

# 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
- seznami in pari
- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija

# 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 oglati)
- smiselno je, da je `pogojN = #t` ("globalni" else)

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


Lokalno okolje,
tokovi, makri