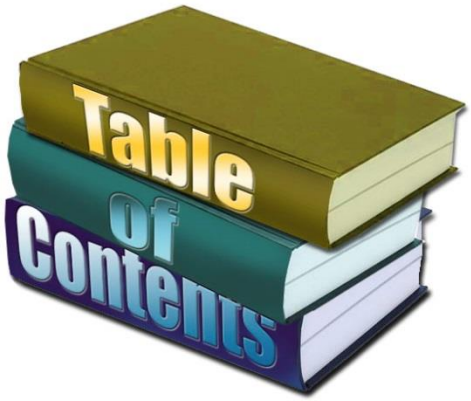
PROGRAMIRANJE
2014/15

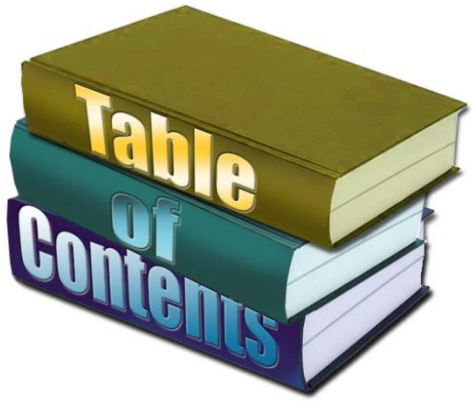
memoizacija
makro sistem
interpreter

Pregled



- memoizacija
- makro sistem
- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk
  - definicija funkcij

Kje smo?

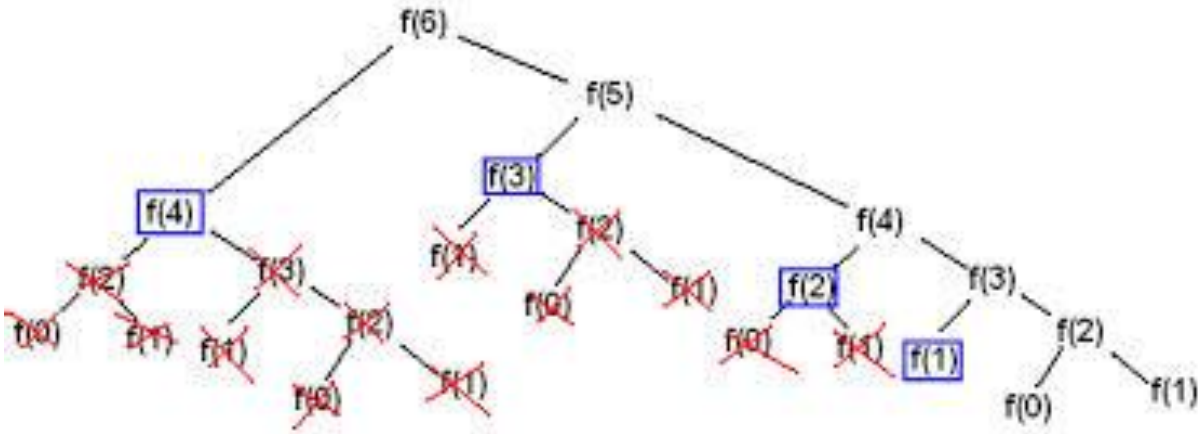


- dinamično tipiziranje
- lokalno okolje
- takojšnja in zakasnjena evalvacija
- zakasnitev in sprožitev
- tokovi
- memoizacija

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?

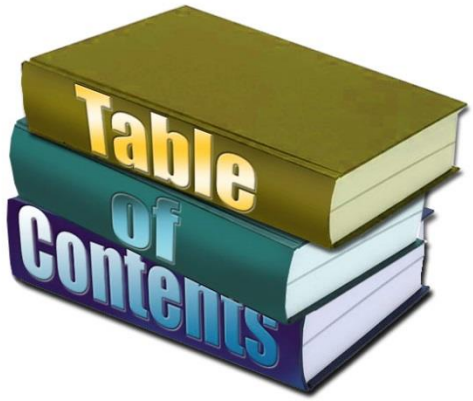


Memoizacija

- 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!`)

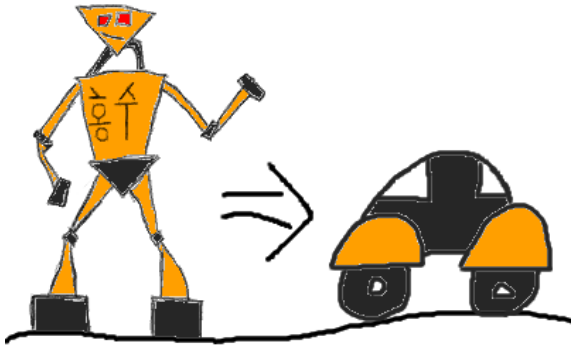
```
(define fib3
  (letrec ([resitve null]
           [pomozna (lambda (x)
                      (let ([ans (assoc x resitve)])              ; poiscemo resitev
                        (if ans
                            (cdr ans)                             ; vrnemo obstojeco resitev
                            (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



- memoizacija
- makro sistem
- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk
  - definicija funkcij

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, Java, …) ima 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 ()
    [(drugi 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 razširjene sintakse

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-delay1
  (syntax-rules ()
    [(my-delay 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 raširitev 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()[(dvakrat3 x)(* 2 x)]))
```

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

```
(define-syntax dvakrat5
  (syntax-rules()
    [(dvakrat3 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

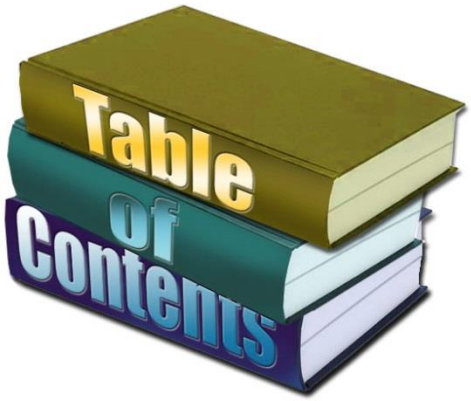
```
> (let [tmp 5]
       [other 6])
   (let ([tmp tmp])
     (set! tmp other)
     (set! other tmp))
   (list tmp other))
'(5 6)
```

naivna makro razširitev

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

Racket

Pregled



- memoizacija
- makro sistem
- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk
  - definicija funkcij

Lastni podatkovni tipi

- **v ML** smo definirali lastne podatkovne tipe
- spomnimo se, da `datatype` v ML ponuja
  - *alternative* podvrst tipa
    (`datatype x = PRVO | DRUGO | TRETJE`)
  - *rekurzivno* definicijo tipa
    (`datatype x = PRVO of x | DRUGO`)

- **Racket**:
  - dinamično tipiziran, zato eksplicitna definicija alternativ ni potrebna
  - preprosta rešitev:
    - simulacija alternativ s seznami oblike
      `(tip vrednost1 … vrednostn)`
    - izdelava funkcij za preverjanje podatkovnega tipa in funkcij za dostop do elementov
    - primer

Nerodnost... ☹

- rešitev ni praktična, dopušča veliko možnosti za napake
  - pri konstruktorju podamo napačno vrednost

```
(Segment "kuku" (Avto "fiat" "modri"))
```

  - preverjanje tipa povzroči napako, če ne upoštevamo načina implementacije

```
(define (Avto? x) (eq? (car x) "avto"))
(Avto? "zivjo")
```

  - dostop do elementov ne preveri, ali je vsebina pravega tipa

```
(Avto-barva (Avto 3.14 2.71))
```

  - sami izdelujemo lastne sezname brez uporabe konstruktorjev

```
(define x (list "avto" "porsche" "rdec"))
```

  - uporaba lastnih metod za dostop (obremenjevanje z implementacijo)

```
namesto (Avto-barva x) uporabimo (car (cdr x))
```

- breme izogibanja napakam pade na program in pomožne funkcije

Boljši način: struct

- definicija lastnega tipa s komponentami

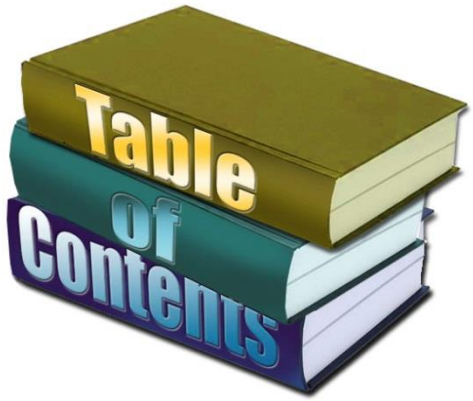
```
(struct ime (komp1 komp2 ... kompn) #:transparent)
```

atribut, ki omogoča izpis v REPL

- rezultat je avtomatska izdelava funkcij:
  - `(ime komp1 komp2 … kompn)` konstruktor novega tipa
  - `(ime? e)` preverjanje vrste tipa
  - `(ime-komp1 e),...,(ime-kompn e)` dostop do komponent (ali napaka)

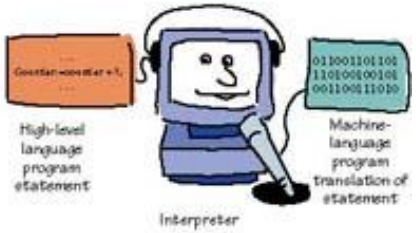
- prednosti
  - implementacija tipa je popolnoma skrita
  - razširitev programa z novim podatkovnim tipom
  - samodejno preverjanje napak
  - podatka ne moremo izdelati drugače kot s konstruktorjem
  - do podatka ne moremo dostopati drugače kot s funkcijami za dostop
  - primeri

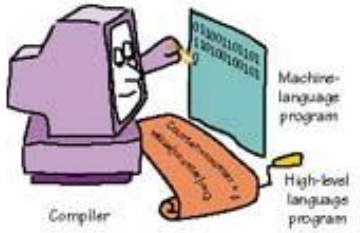
Pregled



- memoizacija
- makro sistem
- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk
  - definicija funkcij

Interpreter ali prevajalnik?



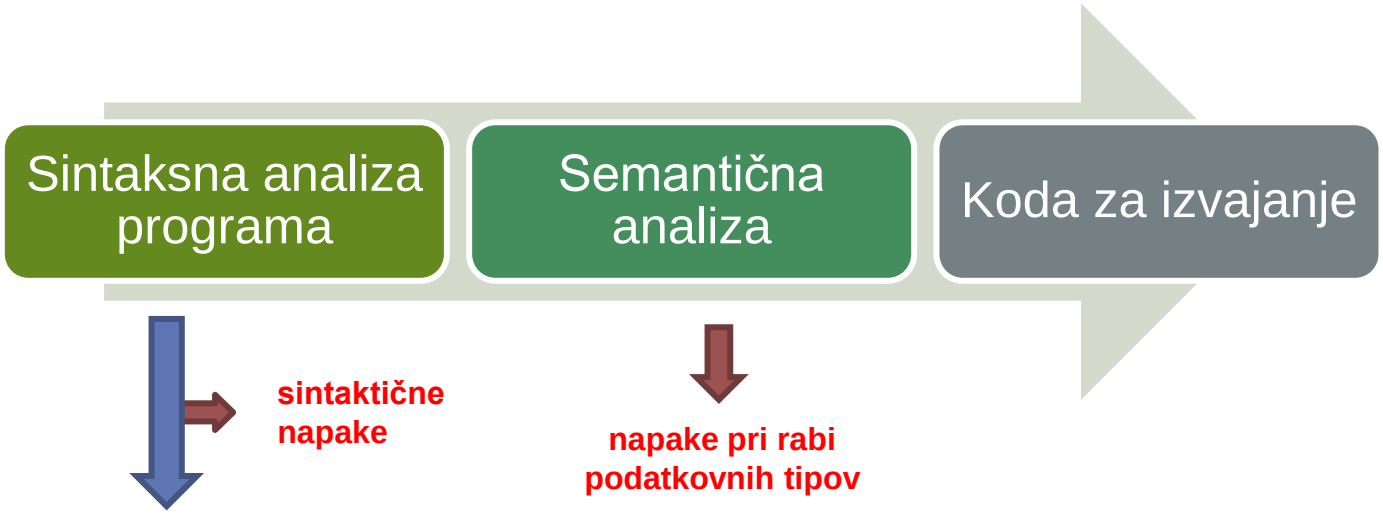


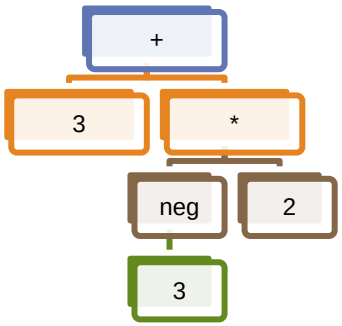
Dve alternativi za implementacijo programskega jezika:

- **INTERPRETER** za programski jezik X
  - napišemo ga v programskem jeziku 0
  - program je v sintaksi jezika X
  - odgovor je v sintaksi jezika X
- **PREVAJALNIK** za programski jezik X
  - napišemo ga v programskem jeziku 0
  - rezultat je program v jeziku P
  - program v jeziku 0 in program v jeziku P imata ekvivalenten pomen

- narediti interpreter ali prevajalnik je le predmet implementacije, ne definicije programskega jezika
- možne so tudi kombinacije prevajanja in interpretiranja:
  - Java je prevajalnik v JVM
  - delno prevajanje (optimizacija) in delno interpretiranje programa

Izvajanje programa





ABSTRAKTNO SINTAKSNO DREVO

sintaksno analizo lahko poenostavimo, ker so izrazi v Racketu že sami po sebi drevesne strukture !!!

Naš pristop

- preskočimo fazo sintaksne analize in razčlenjevanja s podajanjem AST, ki je že v izvornem programskem jeziku 0

- sintakso ciljnega jezika X lahko definiramo z uporabo lastnih podatkovnih tipov (`struct`)

- primer: **JAIS (Jezik Aritmetičnih Izračunov v Slovenščini)**
  - rekurzivna funkcija za računanje z izrazi
  - izrazi za:
    - definicijo konstant (konst)
    - definicijo logičnih vrednosti (bool)
    - negacijo (negiraj)
    - seštevanje (sestej)
    - vejanje (ce-potem-sicer)

Preverjanje pravilnosti programa

- primer interpreterja za JAIS:

```
(define (jais e)
  (cond [(konst? e) e]    ; vrnemo izraz v ciljnem jeziku
        [(bool? e) e]
        [(negiraj? e)
         (let ([v (jais (negiraj-e e))])
           (cond [(konst? v) (konst (- (konst-int v)))]
                 [(bool? v) (bool (not (bool-b v)))]
                 [#t (error "negacija nepričakovanega izraza")]))]
        [(sestej? e)
         (let ([v1 (jais (sestej-e1 e))]
               [v2 (jais (sestej-e2 e))])
           (if (and (konst? v1) (konst? v2))
               (konst (+ (konst-int v1) (konst-int v2)))
               (error "seštevanec ni številka")))]
        [#t (error "sintaksa izraza ni pravilna")]))
```

preverjanje ustreznosti podatkovnih tipov (semantika) že izvajamo

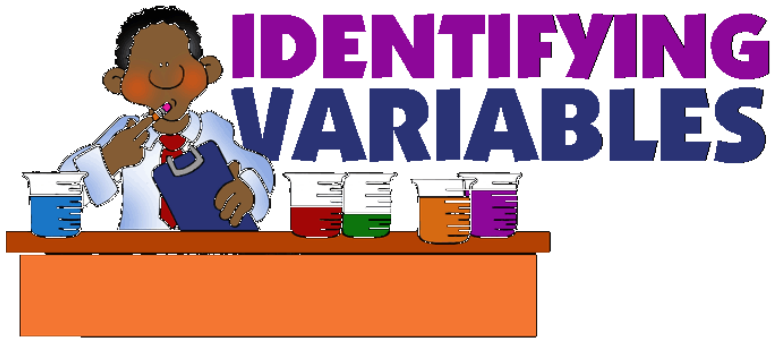
- preverjanje ustreznosti podatkovnih tipov
- preverjanje pravilne sintakse?
  - delno preverja že Racket `(jais (negiraj 1 2 3))`
  - napaka: `(jais (negiraj (konst "lalala")))`
  - potrebno dopolniti kodo, da preverja tudi pravilno sintakso konstant!

Razširitve

- **razširitve preprostega jezika**

  1. definiranje spremenljivk
  2. definiranje lokalnih okolij
  3. definiranje funkcij (funkcijskih ovojnic)
  4. definiranje makrov

potrebujemo znanje o delovanju teh elementov, ki se ga učimo od začetka predmeta

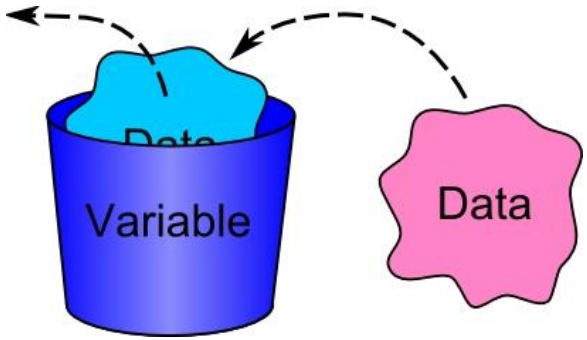


Definiranje spremenljivk in lokalnega okolja

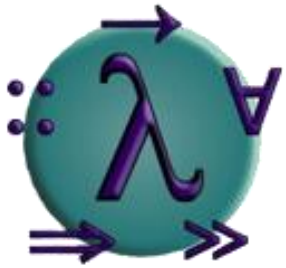
- spremenljivko beremo vedno iz trenutnega okolja (torej potrebujemo okolje)

- okolje prenašamo v spremenljivki jezika 0, ki hrani vrednosti spremenljivk X
  - okolje je na začetku prazno
  - primerna struktura je seznam parov `(ime_spremenljivke . vrednost)`
  - deklaracija nove spremenljivke doda v okolje nov par
  - shranjevanje najprej evalvira podani izraz, nato shrani vrednost

- dostop do spremenljivk
  - preverjanje, ali je spremenljivka definirana
    - če je, vrnemo vrednost
    - sicer napaka



Definiranje funkcij



- potrebujemo strukturo, ki bo hranila funkcijsko ovojnico
  (ne uporabljamo je v sintaksi programa, temveč samo pri izvajanju)

```
(struct ovojnica (okolje funkcija) #:transparent)
```

- v `okolje` shranimo okolje, kjer je funkcija definirana (leksikalni doseg!), v `funkcija` pa funkcijsko kodo

- kako izvesti funkcijski klic?

```
(klici ovojnica argument)
```

  - `ovojnica` mora biti evalvirana v primerek tipa `ovojnica`, sicer napaka
  - `argument` mora biti vrednost (konstanta, boolean), ki je argument funkcije
  - izvajanje:
    - `ovojnica-funkcija` evalviramo v okolju `ovojnica-okolje`, ki ga razširimo z:
      - imenom in vrednostjo argumenta `argument`
      - imenom funkcije, povezano z ovojnico (za rekurzijo)

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))))`

Makri v interpreterju, programski jeziki