FUNKCIJSKO PROGRAMIRANJE

2023/24

lastni podatkovni tipi

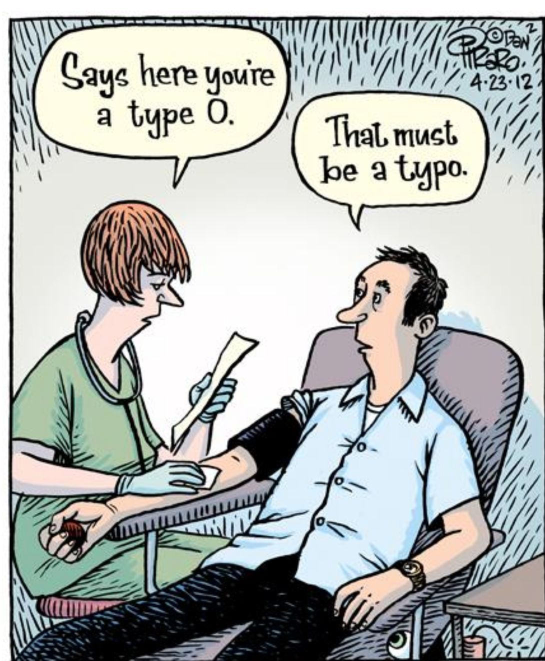
interpreter

argumenti funkcij

primerjava FP in OUP

© Zoran Bosnić

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

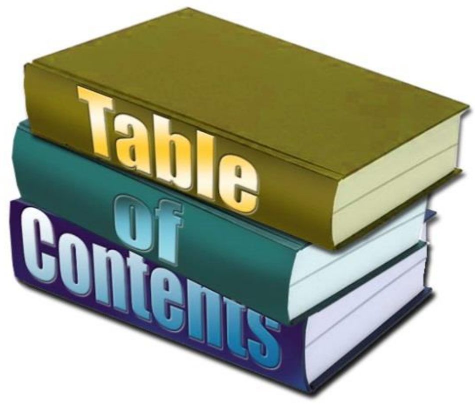
```
(struct ime (komp1 komp2 ... kompn) #:transparent)
```

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)

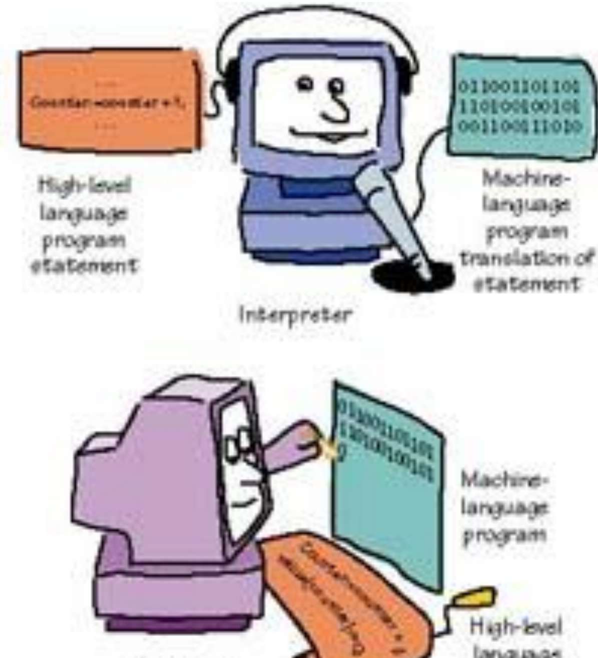
- prednosti
  - implementacija tipa je popolnoma skrita
  - razširitev programa z novim podatkovnim tipom
  - samodejno preverjanje napak
  - podatka ne moremo izdelati drugače kot s konstruktorjem
  - do podatka ne moremo dostopati drugače kot s funkcijami za dostop
  - primeri

Pregled



- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk in lokalnega okolja
  - definicija funkcij
  - definicija makrov
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja

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
  - program je v sintaksi jezika X
  - rezultat je program v jeziku P
  - program v jeziku X 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

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

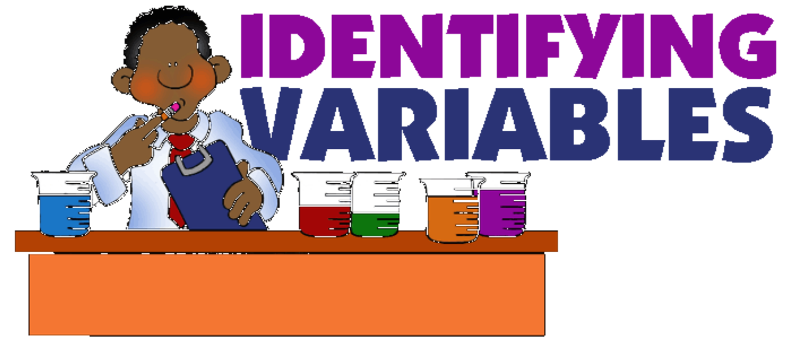
- preverjanje ustreznosti podatkovnih tipov
- preverjanje pravilne sintakse?
  - delno preverja že Racket `(jais (negiraj 1 2 3))`
  - napaka: `(jais (negiraj (konst "lalala")))`
  - potrebno dopolniti kodo, da preverja tudi pravilno sintakso konstant!

Razširitve

- <u>**razširitve preprostega jezika**</u>

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

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

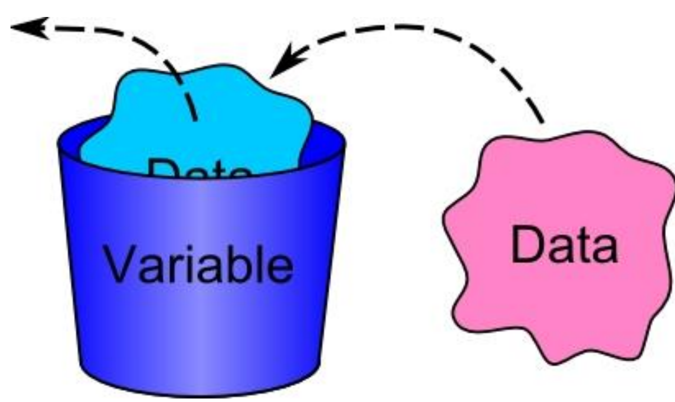


Definiranje spremenljivk in lokalnega okolja

- 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 `funkcija`, ki se je evalvirala v tip `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)

- možne razširitve: rekurzivne funkcije, več formalnih argumentov, anonimne funkcije, optimizacija ovojnic

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)
```

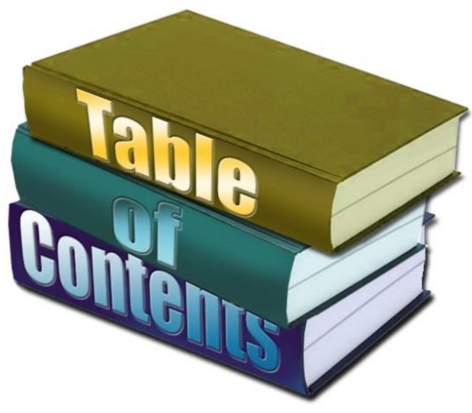
```
(define (vsota-sez sez)
  (if (null? sez)
      (konst 0)
      (sestej (car sez)
              (vsota-sez (cdr sez)))))
```

```
> (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



- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk in lokalnega okolja
  - definicija funkcij
  - definicija makrov
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja

Funkcije z različnim številom argumentov

- **poljubno število argumentov** podamo z imenom spremenljivke brez oklepaja. V funkciji so vsi ti argumenti podani v seznamu, ki sledi ključni besedi `lambda`.

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** naborom osnovnih argumentov in
  - poljubnim številom **dodatnih neobveznih** argumentov

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

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

tretja oblika sintakse

```
(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 **naslavljajo s 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 "!")))
```

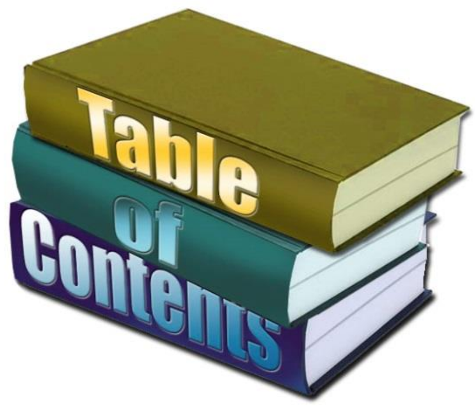
```
> (pozdrav #:ime "Helga" #:voscilo "Auf Wiedersehen")
Auf Wiedersehen, Helga!
> (pozdrav #:voscilo "Auf Wiedersehen" #:ime "Helga")
Auf Wiedersehen, Helga!
```

- 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



- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk in lokalnega okolja
  - definicija funkcij
  - definicija makrov
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja

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

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

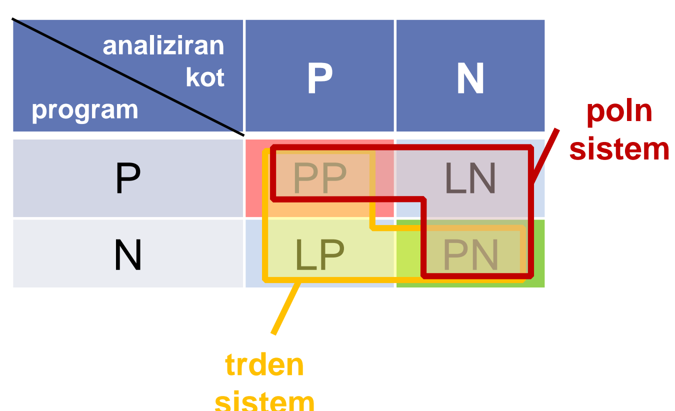
```
(define (fun2 x) (if (> x 10)
                     #t
                     (list 1 "zivjo")))
```

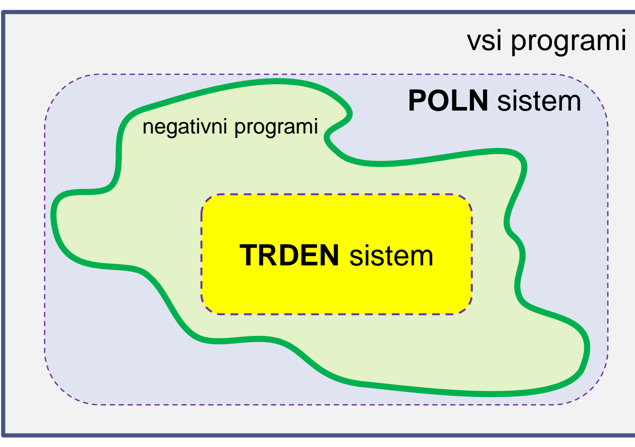
← v SML neveljaven

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

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 (drugače povedano: vedno pravilno razpozna, da je pozitivni program res pozitiven)
  - vendar pa: ima lažno pozitivne primere (= pravilni/negativni programi, ki jih zaznamo kot nepravilne/pozitivne)
- **sistem je POLN** (angl. *complete*), če nikoli ne zavrne negativnega programa (drugače povedano: vedno pravilno razpozna, da je negativni program res negativen)
  - vendar pa: ima lažno negativne primere (= nepravilni/pozitivni programi, zaznani kot pravilni/negativni)





Zakaj nepolnost?

- problem izvajanja statične analize, ki ugotovi vse troje:
  - ali je sistem trden,
  - ali je sistem poln in
  - ali je sistem ustavljiv

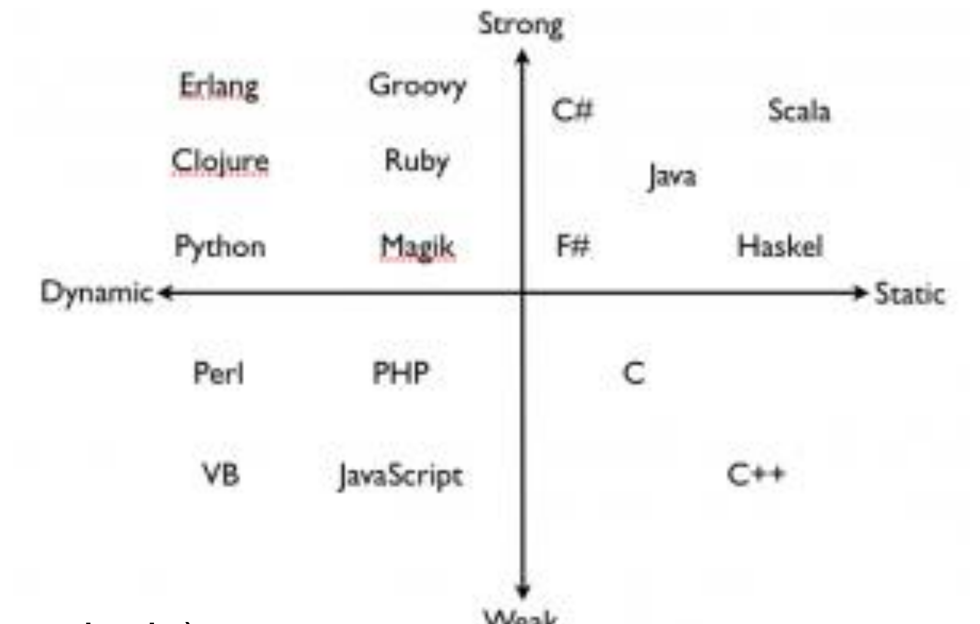
  je **NEODLOČLJIV** (obstaja matematični teorem)

- 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)**

- tudi sistem tipov v SML je trden, vendar ni poln
  - trdni/nepolni sistemi tipov so praksa
  - primer lažno pozitivnega primera (zavrnjeni primeri, ki ne povzročajo težav)?

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

Šibko tipiziranje



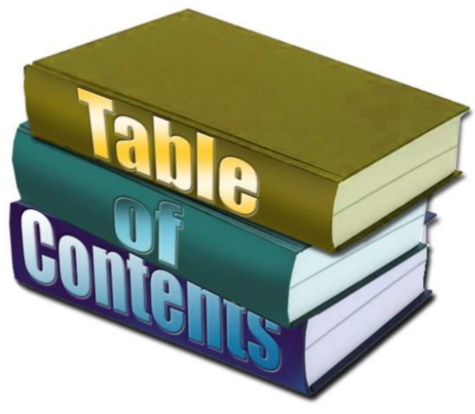
- šibko tipiziranje (angl. *weak typing*)
- sistem, ki:
  - je POLN (dopušča lažne negativne primere)
  - med izvajanjem se lahko zgodi napaka (potrebno preverjanje)
  - sistem izvaja malo statičnih ali dinamičnih preverjanj
  - rezultat: neznan?
  - C / C++
- prednosti šibko tipiziranih sistemov
  - "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)

Prednosti obeh sistemov

	statično preverjanje	dinamično preverjanje
kombiniranje podatkovnih tipov v seznamih in vejah programa	ni možno, vendar pa zato v programu vemo, katere tipe seznamov in funkcij lahko pričakujemo	je možno, moramo pa v programu uporabiti vgrajene predikate za preverjanje tipov
sprejemanje množice programov	zavrača pravilne programe	sprejema več pravilnih programov
čas ugotavljanja napak	napake v programu ugotovimo zgodaj	napake ugotovi šele med izvajanjem
hitrost izvajanja	prihrani na prostoru in času, ker ne označuje spremenljivk z značkami posameznih podatkovnih tipov	prevajalnik potrebuje več prostora in časa za označevanje spremenljivk z značkami, programer ima več dela
večkratna uporabnost programske kode	manjša, vendar s tem večje nadzorovanje napak	večja, vendar odpira možnosti za več napak v programu
prototipiranje novih programov	težje, potrebno določiti podatkovne tipe vnaprej; vendar pa lažje nadzorovanje vpliva sprememb na delovanje obstoječe kode	preprosteje, vendar slabše nadzorovanje vpliva sprememb na obstoječo kodo

- kaj je boljše?
- smiselno je najti kompromis: del preverjanja se izvede statično, del dinamično
- programski jeziki uporabljajo kombinacijo obojega

Pregled



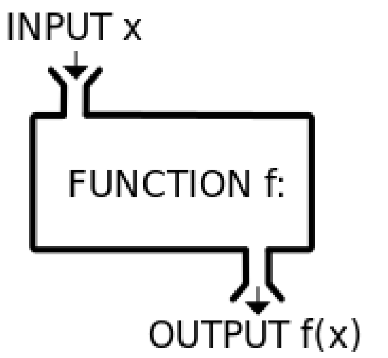
- lastni podatkovni tipi
- interpreter
  - definicija konstruktov
  - definicija spremenljivk in lokalnega okolja
  - definicija funkcij
  - definicija makrov
- funkcije z različnim številom argumentov
- podajanje argumentov po imenih
- primerjava ML in Racket
- trdnost in polnost sistema tipov
- primerjava funkcijskega in OU programiranja

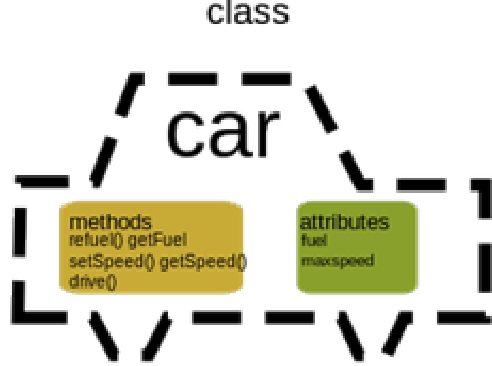
Povezava med FP in OUP

- program lahko analiziramo glede na uporabo podatkovnih tipov in funkcij:

	funkcija1	funkcija2	funkcija3	…
tip1				
tip2				
tip3				
…				

- "narediti program" pomeni "izpolniti" zgornjo tabelo s programsko kodo za vsak podatkovni tip in funkcijo, ki jo uporabljamo





Povezava med FP in OUP

- **funkcijsko programiranje**: program je množica funkcij, ki so zadolžene vsaka za svojo operacijo

	funkcija1	funkcija2	funkcija3	...
tip1				
tip2				
tip3				
...				

- **objektno-usmerjeno programiranje**: program je množica razredov, ki v sebi vsebujejo različne operacije nad primerki razreda

	funkcija1	funkcija2	funkcija3	...
tip1				
tip2				
tip3				
...				

Povezava med FP in OUP

- FP in OUP sta torej le drugačni perspektivi na izdelavo istih programov
- **katerega izbrati?**
  - osebni programerski stil
  - upoštevati je potrebno način razširjanja programa

- **razširjanje programa z novo kodo:**
  - funkcijsko programiranje
    - če dodamo novo funkcijo, moramo v njej pokriti vse konstruktorje za podatkovni tip (sicer nas prevajalnik opozori) ✓
    - če samo razširimo podatkovni tip, moramo dopolniti vse funkcije s kodo za delo s tem tipom !
  - objektno-usmerjeno programiranje
    - če dodamo novo funkcijo za delo s podatkovnimi tipi, jo moramo implementirati na novo v vseh ločenih razredih (ne upoštevajmo možnosti dedovanja) !
    - če implementiramo novi razred, v njemu implementiramo vse možne funkcije (metode) za delo s tem razredom ✓

Funkcijsko programiranje
v Pythonu