FUNKCIJSKO PROGRAMIRANJE

2023/24

magistrski študijski program

Uvod in funkcijsko programiranje

Programiranje

Izvajalci

- **nosilec:**
  Zoran Bosnić
  (kabinet: 2. nadstropje, R2.17)

- **asistent:**
  Klemen Klanjšček
  (laboratorij: 3. nadstropje, LKRV, 3.28)

Cilji predmeta

- postati boljši programer
- razumeti delovanje programskega jezika
- izstopiti iz okvirja objektno-usmerjenega programiranja
- naučiti se drugačnih (trendovskih?) pristopov k programiranju
- znati kritično razumeti članke, kot je ta ☺
  http://fsharpforfunandprofit.com/posts/ten-reasons-not-to-use-a-functional-programming-language/

- prehod v čisto funkcijsko okolje, ki omogoča učenje pristopov k funkcijskemu programiranju
  - zakaj?

Literatura

SML:

- Riccardo Pucella: Notes on Programming SML/NJ, Cornell, 2001 (povezava na učilnici!),
- Michael R. Hansen, Hans Rischel: Introduction to Programming using SML. Addison-Wesley, 1999.
- CSE341, University of Washington.

Racket:

- Matthew Flatt, Robert Bruce Findler et al.: The Racket Guide, http://docs.racket-lang.org/guide/.

Python:

- Python za programerje. Ljubljana: Fakulteta za računalništvo in informatiko, 2009.

Splošna:

- Ravi Sethi: Programming Languages: concepts & constructs. Addison-Wesley, 1996.
- A. Tucker, R. Noonan: Programming Languages: Principles and Paradigms. McGraw-Hill, 2007.

literatura... ?

Obveznosti predmeta

1. **laboratorijske vaje**
   - predvidoma 9 nalog (#0 - #8)
   - oddaja programske datoteke preko učilnice
   - obvezna pozitivna ocena (>= 50%) vsaj 5 nalog: #0, #1, #2 in še 2 po lastni izbiri
   - za vseh 9 nalog: bonus +10% na izpitu (prišteje se k pozitivni oceni)

2. **2 seminarski nalogi**
   - samostojno delo doma, zagovor na vajah
   - 2 nalogi (SML, Racket)
   - pri vsaki potrebno doseči vsaj 50%
   - nalogi tvorita oceno sprotnega dela

3. **končni izpit**
   - pisni (in/ali ustni) izpit

ČE so laboratorijske vaje pozitivne,
POTEM
KONČNA OCENA =
50% * (pisni izpit + bonus)
+ 50% * seminarski nalogi

Vsebina predmeta

- **funkcijsko programiranje** — SML
  - uvod v Standardni ML
  - sestavljeni podatkovni tipi
  - konteksti spremenljivk
  - ujemanje vzorcev, gnezdenje vzorcev
  - funkcije višjega reda, map/reduce/filter
  - ovojnice, delne aplikacije, currying
  - skrivanje kode v programskih modulih
  - zakasnjena evalvacija — Racket
  - memoizacija
  - uporaba makro sistema
  - močna/šibka, dinamična/statična tipizacija
  - implementacija interpreterja
  - funkcijsko programiranje v mešano-paradigemskem okolju? — Python
  - primerjava z objektno-usmerjenim progr.

Vaje, nadomeščanja

- **laboratorijske vaje**
  - asistent: Klemen Klanjšček
  - pričetek vaj že ta teden

- **prazniki** (predavanja odpadejo):
  - 31. 10. 2023
  - 26. 12. 2023
  - 2. 1. 2024

Organizacija?

- gradnja znanja z individualnim delom
- učimo se z raziskovanjem, testiranjem, preizkušanjem variacij programov
- doma preverimo programsko kodo s predavanj in utrdimo razumevanje predelane snovi do naslednjih predavanj
- cenimo iznajdljivost (zunanji viri, programerski triki) in jo radi delimo
- radi si pomagamo in ustvarjamo pozitivno vzdušje na forumu, kjer delimo znanje s kolegi in se vzpodbujamo
- upoštevamo etične vrednote, kolegialnost in kulturo izražanja
- Discord vs. predmetni forum?

Funkcijsko programiranje

Funkcijsko programiranje

- algoritem - opis postopka za reševanje problema
- načini opisovanja problemov v sodobnih programskih jezikih:
  - **proceduralno**: program je zaporedje navodil (C, Pascal, skriptni jeziki, Basic)
  - **deklarativno**: program je specifikacija, ki opiše problem; jezik izvede reševanje (SQL, Prolog)
  - **objektno-usmerjeno**: programi manipulirajo zbirke objektov s stanji in metodami (C++, Java, Python)
  - **funkcijsko**: program je zapisan kot zaporedje funkcij. Te imajo vhode in izhode, ne hranijo in spreminjajo pa notranjega ali globalnega stanja (spremenljivk) (Standardni ML, OCaml, Haskell, Lisp, Racket, Scheme, Clean, Mercury, Erlang).
- mešano-paradigemski jeziki (npr. Python, R, ...)

Funkcijsko programiranje

- večji programi -> večja kompleksnost kode
- potreba po večji abstrakciji kode, izogibanje pretiranim podrobnostim
- analogija - babičin recept:

Yes, dear, to make pea soup you will need split peas, the dry kind. And you have to soak them at least for a night, or you will have to cook them for hours and hours. I remember one time, when my dull son tried to make pea soup. Would you believe he hadn't soaked the peas? We almost broke our teeth, all of us. Anyway, when you have soaked the peas, and you'll want about a cup of them per person, and pay attention because they will expand a bit while they are soaking, so if you aren't careful they will spill out of whatever you use to hold them, so also use plenty water to soak in, but as I said, about a cup of them, when they are dry, and after they are soaked you cook them in four cups of water per cup of dry peas. Let it simmer for two hours, which means you cover it and keep it barely cooking, and then add some diced onions, sliced celery stalk, and maybe a carrot or two and some ham. Let it all cook for a few minutes more, and it is ready to eat.

Per person: one cup dried split peas, half a chopped onion, half a carrot, a celery stalk, and optionally ham.

Soak peas overnight, simmer them for two hours in four cups of water (per person), add vegetables and ham, and cook for ten more minutes.

Marijn Haverbeke: Eloquent JavaScript. A Modern Introduction to Programming

Prvi koraki po abstrakciji

- ostanimo pri proceduralnem (imperativnem) programiranju
  - abstraktnejši pristop k programiranju,
  - zmanjšanje količine nepotrebnih podrobnosti z namenom povečanja preglednosti programa
- primer:

```
def sestej1(stevila):
    i = 0
    sum = 0
    meja = len(stevila)
    while i < meja:
        sum += stevila[i]
        i += 1
    return sum

def sestej2(stevila):
    sum = 0
    for x in stevila:
        sum += x
    return sum
```

"Imamo neko spremenljivko *i*, ki začne šteti pri 0 in šteje do velikosti polja, ki mu rečemo *meja*, ter za vsako vrednost *i* poiščemo ustrezen element v seznamu stevila in ta element prištejemo skupni vsoti..."

"Za vsak element seznama števila prištej ta element k skupni vsoti"

želimo iti še dlje? DA! Funkcijsko programiranje:

- zapis programa kot evalvacija matematičnih funkcij,
- fokus: KAJ izračunati namesto KAKO izračunati

Funkcijsko programiranje

- lastnosti funkcijskih jezikov
  - funkcije so objekti
  - podpora funkcijam višjega reda (funkcije, ki se izvajajo na funkcijah, ki se izvajajo na funkcijah, ki se izvajajo na ...)
  - uporaba rekurzije namesto zank
  - poudarek na delu s seznami
  - izogibanje "stranskim učinkom" programa (spreminjanje globalnega stanja, lokalne spremenljivke)
    - do največjega deleža programskih napak pridemo, kadar spremenljivke med izvajanjem programa dobijo nepričakovane vrednosti. Rešitev: Izognimo se prirejanjem!

- prednosti funkcijskih programov
  - lažji formalni dokaz pravilnosti programa
  - idempotentne funkcije (ne spreminjajo stanja po večkratni zaporedni uporabi) -> lažje testiranje (unit test) in razhroščevanje
  - ni definiran vrsti red evalvacije funkcij, možna lena (pozna) evalvacija (lazy evaluation)
  - sočasno procesiranje -> boljša paralelizacija
  - pomagajo boljše razumeti programerske stile

Standardni ML

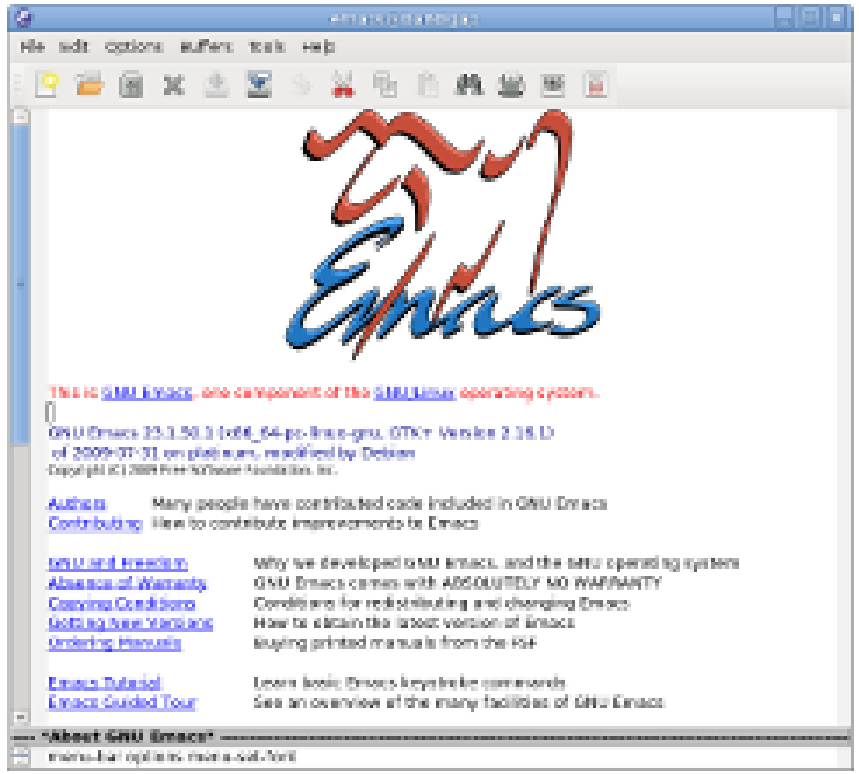
Nenavadno delovno okolje ☺

- namestitev delovnega okolja na vajah (na vaje prinesite lastne prenosnike)

- **jezik ML**
  - uporabljali bomo SML/NJ (Standard ML of New Jersey), ver 110.99.3 (julij 2021)
  - http://www.smlnj.org/

- **urejevalnik za ML**
  - Emacs (z razširitvijo sml-mode)
  - ML-Edit
  - Sublime
  - Eclipse
  - Notepad, Notepad++
  - ...

Emacs - vmesnik



Številne bližnjice tipkovnice!

Splošno:

- **C-x C-c**: izhod
- **C-x C-f**: odpri datoteko
- **C-x C-s**: shrani datoteko
- **C-h b**: seznam razpoložljivih bližnjic

Uporaba odlagališča:

- **C-w**: izreži
- **M-w**: kopiraj
- **C-y**: prilepi

Okna:

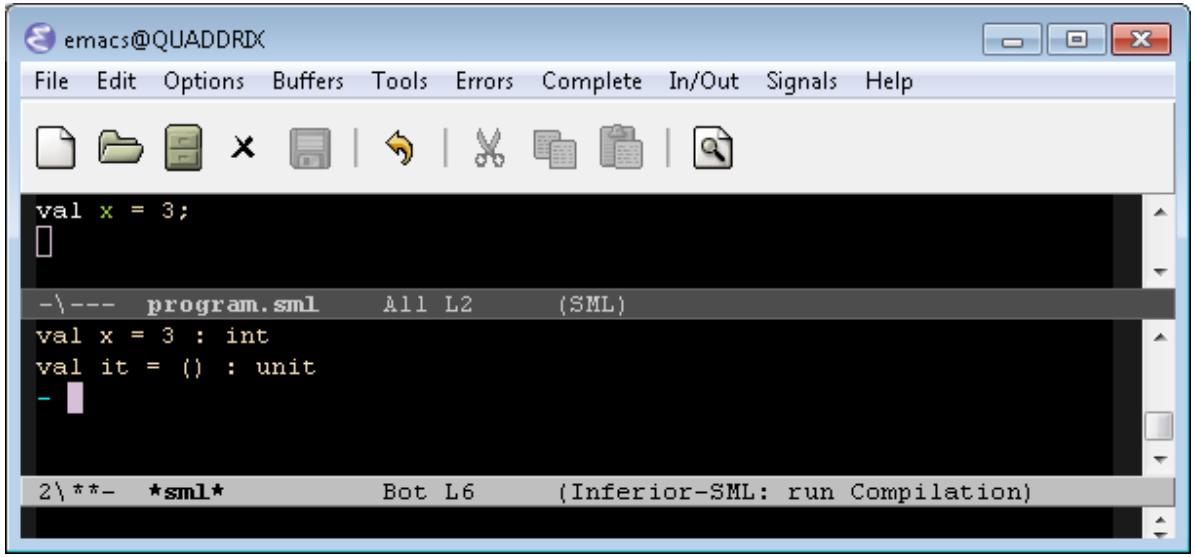
- **C-x 2**: delitev na 2 okna (zgoraj in spodaj)
- **C-x 1**: uporaba enega okna

Pomoč? Referenčni plonk listki (glej učilnico)!

Program v ML

- **REPL - Read-Eval-Print Loop:**
  branje-evalvacija-izpis rezultatov, ki se izvaja v zanki
- SML ni interpreter, temveč prevajalnik. REPL torej v ozadju prevede vsak ukaz v strojno kodo

- **Uporaba iz Emacsa:**
  - **C-c C-s**: požene SML REPL v novem oknu
  - obstoječo datoteko odpremo z ukazom **use "datoteka.sml";**
  - **C-d** prekine delo z REPL



Preprosti izrazi

```
- (* jaz sem komentar *)
- 12;                          (* celoštevilska vrednost *)
- true;                        (* logična vrednost *)
- 3.14;                        (* realna vrednost *)
```

Sintaksa in semantika

SINTAKSA = kako program pravilno zapišemo

SEMANTIKA = kaj program pomeni

- preverba pravilnosti podatkovnih tipov, angl. type-checking (pred izvajanjem)
- evalvacija - izračun vrednosti (med izvajanjem)

ML preverja semantiko z uporabo **statičnega** in **dinamičnega okolja:**

- **statično okolje** hrani podatke o programu, potrebne za preverjanje njegove pravilnosti pred izvajanjem. Hrani npr. podatkovne tipe obstoječih spremenljivk.
- **dinamično okolje** hrani podatke o programu, ki so potrebne za preverjanje pravilnosti med izvajanjem. To so npr. vrednosti obstoječih spremenljivk.

Poglejmo si primere...

Vezava spremenljivk

V splošnem:

```
val x = e
```

x je spremenljivka

- sintaksa: zaporedje črk, številk in znakov _, ki se ne prične s številko
- pravilnost tipa: preveri, ali je že prisotna v statičnem okolju
- evalvacija: preberi vrednost iz dinamičnega okolja

e je izraz

- sintaksa: vrednost ali sestavljeni izraz
- semantika: vsaka vrednost ima svoj "vgrajeni tip" in se evalvira sama vase

Vezava spremenljivk ni enako **prirejanju vrednosti** spremenljivki! Spremenljivko je možno vezati večkrat. Nova vezava **zasenči** (angl. *shadow*) prejšnjo vezavo (slaba programerska praksa).

Program v SML?

Program je zaporedje vezav (angl. *binding*, spremenljivko ***povežemo*** z njeno vrednostjo), primer:

```
- val x = 3                    (* vezava spremenljivke x *)
- val y = 5                    (* podpičja lahko izpustimo *)
- val z = x + y                (* matematična operacija *)
- val uspesno = if z > 5 then true else false
                               (* pogojni stavek *)
```

Seštevanje

SINTAKSA

`e1 + e2`

`e1` in `e2` sta vgnezdena izraza (podizraza)

SEMANTIČNA PRAVILA

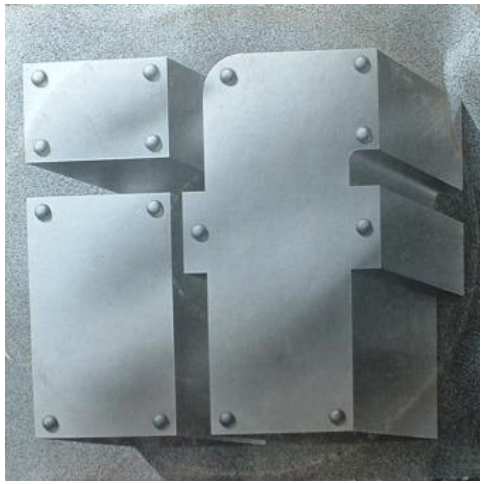
1. Pravilnost tipov

če je `e1` tipa `int` in je `e2` tipa `int`, je rezultat tipa `int`

2. Način evalvacije

če je vrednost izraza `e1` enaka `v1`
in vrednost izraza `e2` enaka `v2`,
je vrednost izraza `e1+e2` enaka `v1+v2`

Pogojni stavki



SINTAKSA

if `e1` then `e2` else `e3`

`e1`, `e2` in `e3` so vgnezdeni izrazi

SEMANTIČNA PRAVILA

1. Pravilnost tipov

`e1` mora biti tipa `bool`

`e2` in `e3` morata biti enakega tipa (!!!) - imenujmo ga `t`

rezultat celega izraza je tipa `t`

2. Način evalvacije

evalviraj `e1` v vrednost `v1`

če je `v1` enako `true`, evalviraj `e2` v `v2`; `v2` je rezultat

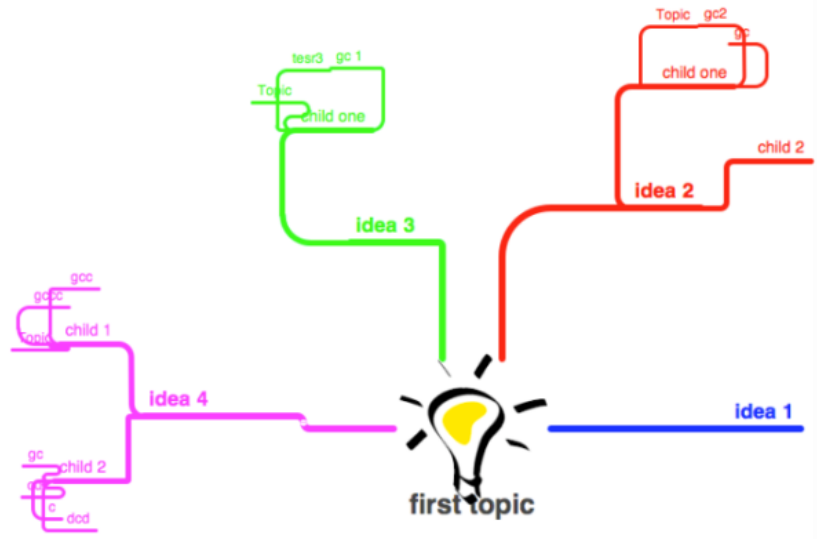
če je `v1` enako `false`, evalviraj `e3` v `v3`; `v3` je rezultat

Programske napake

Kakšni so primeri tipičnih napak?

- sintaktične napake
- napake preverjanja tipov
- napake pri evalvaciji

Poglejmo si primere...



Funkcije

- kot jih že poznamo: imajo argumente in vračajo rezultat
- deklaracija funkcije

```
fun obseg(r: real) =
    2.0 * Math.pi * r
```

- funkcija se hrani kot vrednost, ki slika vhodni argument v izhodnega

```
val obseg = fn : real -> real
```

- klic funkcije

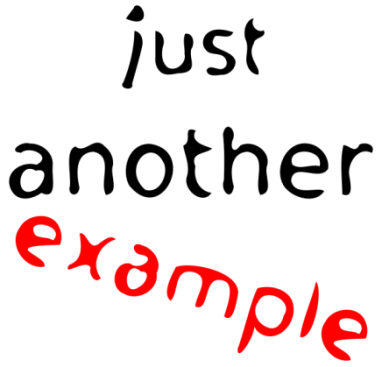
```
obseg(2.5);
obseg (2.5);
obseg 2.5;
```

Primeri funkcij

Pogojni stavek (vejanje) poznamo, kako pa doseči ponavljanje (zanko)?

- konstrukta za zankanje programa
  (for, while, repeat) nimamo

➢ odgovor: s klicem funkcije (same sebe - rekurzija)!



Napiši funkcije, ki izračunajo:

- obseg krožnice pri podanem polmeru,
- potenco x^y (x in y sta podani naravni števili),
- faktorielo podanega števila n,
- vsoto naravnih števil od 1 do n,
- vsoto naravnih števil od a do b.

Podrobno o funkcijah

- funkcije se obravnavajo kot **vrednosti**, ki se **evalvirajo kasneje** (ob klicu)
- znak "`*`" v zapisu tipov argumentov funkcije (`int * int -> int`) ne pomeni množenja
- funkcije lahko kličejo samo funkcije, ki so že definirane v kontekstu (torej definirane prej ali same sebe)
- oznake tipov lahko pogosto izpustimo in pustimo, da SML sam sklepa na njih

Formalno - deklaracija funkcije

SINTAKSA

`fun x0 (x1: t1, … , xn: tn) = e`

argument `xi` je tipa `ti`; telo funkcije je izraz `e`

SEMANTIČNA PRAVILA

1. Pravilnost tipov

uspešno, če ob klicu v statičnem okolju velja `x1: t1, …, xn: tn`
in `x0 : (t1 * … * tn) -> t` (za rekurzijo)

2. Način evalvacije

vezava doda `x0` v okolje (da lahko funkcijo kličemo)

Formalno - klic funkcije

SINTAKSA

```
e0 (e1, … , en)
```

SEMANTIČNA PRAVILA

1. Pravilnost tipov

uspešno, če ima `e0` tip `(t1 * … * tn) -> t`
in velja `e1: t1, ..., en: tn`
tedaj: `e0(e1,…,en)` ima tip `t`

2. Način evalvacije

evalviraj `e0` v `fun x0 (x1 : t1, … , xn : tn) = e`
evalviraj argumente `e1, ..., en` v vrednosti `v1, ..., vn`
evalviraj telo `e`, pri čemer preslikaj `x1` v `v1, ..., xn` v `vn`
telo `e` naj vsebuje `x0`

Osnovni in sestavljeni podatkovni tipi