FRInterpreter

Seminarska naloga 2 pri predmetu Funkcijsko programiranje

December, 2019

V tem seminarju boste implementirali interpreter za programski jezik *FR* in vse konstrukte, ki so potrebni, da lahko v njem pišemo programe.

Klic interpreterja naj ima sintakso `(fri expression environment)`, kjer `expression` predstavlja izraz v jeziku *FR*, `environment` pa spremenljivko, ki hrani začetno okolje.

Konstrukti jezika *FR* naj bodo definirani z Racketovim konstruktom `struct`. Uporabite jih za definicije konstruktov opisanih v sledečih odstavkih.

Podatkovni tipi

- Logični vrednosti `(true)` in `(false)`: `(true)` predstavlja resnično vrednost, `(false)` pa neresnično.
- Cela števila `(zz n)`: `n` je celo število v Racketu.
- Racionalna števila `(qq e1 e2)`: `e1` predstavlja števec, `e2` pa imenovalec. Števec in imenovalec sta tuji (ali pa je števec enak 0 in imenovalec enak 1) celi števili v *FR*.
- Kompleksna števila[1] `(cc e1 e2)`: `e1` predstavlja realno komponento, `e2` pa kompleksno. Obe komponenti sta racionalni števili v *FR*.
- Zaporedja `(..  e1 e2)`, `(empty)`: `(empty)` predstavlja konec zaporedja, `(..  e1 e2)` pa zaporedje, ki ga dobimo, če rezultat evalvacije izraza `e1` dodamo na začetek zaporedja, ki ga dobimo kot rezultat evalvacije izraza `e2`.

Nadzor toka

- Vejitev `(if-then-else condition e1 e2)`: Če se izraz `condition` evalvira v `(false)`, potem je rezultat evalviran izraz `e2`, v vseh drugih primerih je rezultat evalviran izraz `e1`.
- Preverjanje tipov `(is-zz?  e1)`, `(is-qq?  e1)`, `(is-cc?  e1)`, `(is-bool?  e1)`, `(is-seq?  e1)`, `(is-proper-seq?  e1)`, `(is-empty?  e1)`: Funkcije vračajo `(true)`, če je rezultat izraza `e1` ustreznega tipa. V primeru `(is-proper-seq?  e1)` je rezultat `(true)` le, če je podano zaporedje "pravo" zaporedje (se konča z `(empty)`).

  Opomba: Obnašamo se, kot da cela števila niso racionalna števila, pa tudi ne kompleksna. Podobno naj velja tudi za racionalna števila.
- Seštevanje `(add e1 e2)`: Glede na tipe argumentov ločimo več primerov:
  - Če sta izraza logični vrednosti je rezultat disjunkcija[2] (`e1` $\vee$ `e2`).
  - Če sta izraza `e1` in `e2` celi števili, potem je rezultat njuna vsota, ki je spet celo število.
  - Če je eden izmed izrazov `e1` in `e2` racionalno število (in je drugi izraz celo ali racionalno število), potem je rezultat racionalno število, ki predstavlja njuno vsoto.

[1] Tu gre le za podmnožico kompleksnih števil z racionalno realno in imaginarno komponento.
[2] Na mesto disjunkcije bi tu lahko uporabili ekskluzivni ali.

  - Če je eden izmed izrazov `e1` in `e2` kompleksno število (in je drugi izraz celo, racionalno ali kompleksno število), potem je rezultat kompleksno število, ki predstavlja njuno vsoto.
  - Če sta izraza `e1` in `e2` zaporedji, je rezultat seštevanja njuna združitev, tako da zaporedje `e1` nadaljujemo z `e2`.
- Množenje `(mul e1 e2)`: Glede na tipe argumentov ločimo več primerov:
  - Če sta izraza logični vrednosti je rezultat konjunkcija (`e1` $\wedge$ `e2`).
  - Če sta izraza `e1` in `e2` celi števili, potem je rezultat njun produkt, ki je spet celo število.
  - Če je eden izmed izrazov `e1` in `e2` racionalno število (in je drugi izraz celo ali racionalno število), potem je rezultat racionalno število, ki predstavlja njun produkt.
  - Če je eden izmed izrazov `e1` in `e2` kompleksno število (in je drugi izraz celo, racionalno ali kompleksno število), potem je rezultat kompleksno število, ki predstavlja njun produkt.
- Primerjanje `(leq? e1 e2)`: Rezultat je logična vrednost v *FR*. Glede na tipe argumentov ločimo več primerov:
  - Če sta izraza logični vrednosti je rezultat implikacija `e1` $\implies$ `e2`.
  - Če sta izraza `e1` in `e2` celi ali racionalni števili, potem je rezultat `e1` $\leq$ `e2`.
  - Če sta izraza zaporedji je rezultat `(true)`, če ima zaporedje `e1` enako ali manjše število elementov kot zaporedje `e2`.
- Zaokroževanje `(rounding e1)`: Zaokroži celo ali racionalno število (podano kot rezultat izraza `e1`) k najbližjemu celemu številu. Če je število ravno na sredini, zaokrožimo na najbližje sodo celo število.
- Ujemanje `(=? e1 e2)`: Vrne rezultat `(true)`, če se evalvirana izraza `e1` in `e2` ujemata oz. sta enaka.
- Ekstrakcija `(left e1)`, `(right e1)`: V primeru, da je izraz `e1` racionalno število `(left e1)` vrne števec, `(right e1)` pa imenovalec.

  Če je izraz `e1` kompleksno število, `(left e1)` vrne realno komponento, `(right e1)` pa imaginarno.

  Za zaporedje `e1` izraz `(left e1)` vrne prvi element zaporedja, `(right e1)` pa preostali del zaporedja.
- Nasprotna vrednost `(~ e1)`: Vrne nasprotno vrednost za izraze `e1`, ki se evalvirajo v bodisi logično vrednost, celo ali racionalno število. Za kompleksna števila vrne konjugirano vrednost podanega števila.
- Operatorja `(all? e1)`, `(any? e1)`: Če zaporedje `e1` ne vsebuje logične vrednosti `(false)`, potem je rezultat izraza `(all? e1)` enak `(true)`, drugače `(false)`.

  Če zaporedje `e1` ne vsebuje **le** logičnih vrednosti `(false)`, potem je rezultat izraza `(any? e1)` enak `(true)`, drugače `(false)`.

Spremenljivke

Implementirajte spremenljivke in okolje za njihovo shranjevanje. Ko poženete interpreter naj bo okolje prazno. Med njegovim delovanjem so vrednosti shranjene in prebrane. Okolje naj bo predstavljeno s seznamom parov v jeziku Raket, ki naj bo naslednje oblike `(list (var_name_1 . value_1) (var_name_2 . value_2) ... (var_name_n . value_n))`. Definirate sledeča konstrukta za definicijo in uporabo spremenljivk.

- Lokalno okolje `(vars s e1 e2)`: Razširi trenutno okolje z imenom spremenljivke `s`, ki ima vrednost `e1`, ter v razširjenem okolju evalvira izraz `e2`. Če sta `s` in `e1` Racketova seznama, potem razširi trenutno okolje z vsemi imeni in vrednostmi v seznamih in evalvira izraz `e2`.
- Vrednost spremenljivke `(valof s)`: Ob evalvaciji vrne vrednost spremenljivke. V okolju se lahko nahaja več spremenljivk z istim imenom, ki se med seboj senčijo. Izraz naj vrne pravilno vrednost spremenljivke, ki ni zasenčena.

Funckije

Implementirajte funkcije, skripte (procedure) in klice. Funkcije uporabljajo leksikalni doseg, skripte pa dinamičnega. Definirajte sledeče konstrukte za delo s funkcijami, skriptami in klici:

- Funkcije in procedure `(fun name farg body)`, `(proc name body)`: `name` predstavlja ime funkcije oz. procedure, podano v obliki Racketovega niza. `fargs` je Racketov seznam argumentov in `body` predstavlja telo funkcije podano kot izraz v jeziku *FR*.

  Če je ime funkcije prazen niz, gre za anonimno funkcijo. Vsi argumenti funkcije morajo imeti različna imena (Racketove nize).

  Obravnavanje funkcij implementirajte tako, da ob interpretiranju programa se konstrukt `fun` evalvira v funkcijsko ovojnico `(closure env f)`, kjer je `f` originalni konstrukt funkcije, `env` pa okolje na mestu, kjer je bila funkcija evalvirana.

  Opomba: Funkcijske ovojnice niso veljaven del programa v jeziku *FR*, temveč so le interni mehanizem interpreterja. Funkcija je torej izraz, ki ga interpreter evalvira v ovojnico.

- Funkcijski klici `(call e args)`: Klic je definiran za vse izraze `e`, ki se evalvirajo bodisi v ovojnico (`(closure ...)`) ali pa v proceduro (`(proc ...)`. `args` je Racetov seznam izrazov, ki se evalvirajo v vrednosti argumentov. Pravila za evalvacijo so sledeča:

  – Pri klicih funkcijskih ovojnic naj se okolje vsebovano v ovojnici razširi z imeni in vrednostmi argumentov (imena najdemo v opisu funckije – `fargs`, vernosti pa so priložene klicu – `args`) in imenom funkcije, ki ga povežemo z ovojnico (da omogočimo rekurzivne klice).

    Opomba: V primeru, da je ime funkcije enako imenu argumenta funkcije, argument zasenči funkcijo.

    V tako dobljenem okolju se izvede telo funkcije.

  – Pri klicih procedur se telo funkcije izvede v lokalnem okolju (tistem, v katerem se izvede klic funkcij), ki mu dodamo le ime procedur. Ker procedure nimajo argumentov, kot seznam argumentov prejme prazen Racketov seznam.

  Funkcijsko ovojnico optimizirajte tako, da iz nje izločite senčenja zunanjih spremenljivk, senčenja spremenljivk z lokalnimi argumenti in spremenljivke, ki v funkciji niso potrebne.

Makro sistem

Implementirajte funkcije v Racetu, ki bodo delovale kot makri v jeziku *FR*, olepšale sintakso in razširile množico konstruktov. Definirajte sledeče makre:

- Števec in imenovalec ulomka `(numerator e1)`, `(denominator e1)`: Iz rezultata izraza vrne `e1` števec in imenovalec ulomka pripadajočega racionalnega števila.

- Realna in kompleksna komponenta `(re e1)`, `(im e1)`: Iz rezultata izraza `e1` vrne realno in kompleksno komponento kompleksnega števila.

- Večji `(gt? e1 e2)`: Preveri urejenost med elementoma. Definiran je za vse vrednosti, za katere je izraz `(leq? e1 e2)` legalen.

- Obrat `(inv e1)`: Če je rezultat izraza `e1` število, potem vrne njegovo obratno vrednost (celo število postane racionalno). Če je rezultat izraza `e1` zaporedje, potem vrne zaporedje iz obrnjenim vrstnim redom.

- Mapping `(mapping f seq)`: Vrne izraz (v jeziku *FR*), ki ima funkcionalnost funkcije `List.map` v Standard ML, a brez curryinga. `f` predstavlja izraz, ki se bo evalviral v funkcijsko ovojnico, `seq` pa izraz, ki se bo evalviral v zaporedje elementov. Predpostavite lahko, da je funkcija kompatibilna z elementi zaporedja.

- Filtering (`filtering f seq`): Vrne izraz (v jeziku *FR*), ki ima funkcionalnost funkcije `List.filter` v Standard ML, a brez curryinga. `f` predstavlja izraz, ki se bo evalviral v funkcijsko ovojnico, `seq` pa izraz, ki se bo evalviral v zaporedje elementov. Predpostavite lahko, da je funkcija kompatibilna z elementi zaporedja.
- Folding (`folding f init seq`): Vrne izraz (v jeziku *FR*), ki ima funkcionalnost funkcije `List.foldl` v Standard ML, a brez curryinga. `f` predstavlja izraz, ki se bo evalviral v funkcijsko ovojnico, `init` izraz, ki se bo evalviral v začetno vrednost, `seq` pa izraz, ki se bo evalviral v zaporedje elementov. Predpostavite lahko, da je funkcija kompatibilna z elementi zaporedja.

Opomba: Makri naj podane izraze evalvirajo le enkrat.

*Nadgradnja – mutacije in vzajemna rekurzija

Jeziku *FR* dodajte podporo za definicijo pravih spremenljivk in funkcij, ki so vzajemno rekurzivne. Način implementacije je prepuščen vam. To nadgradnjo boste zagovarjali ločeno.

Oddaja seminarske naloge

Nalogo oddajte v obliki ene datoteke z imenom `sem2-<vpisna_stevilka>.rkt`. V oddani datoteki naj bodo zakomentirani testi. Tako testi kot celotna koda naj bo dokumentirana. Na kratko tudi opišite katere stvari ste uspešno implementirali.

Točkovanje bo izvedeno samodejno, svoje rešitve pa boste morali tudi zagovarjati. Samodejno testiranje bo izvedeno s testnimi primeri, ki ne bodo znani vnaprej. Oddana seminarska naloga bo ocenjena glede na naslednji točkovnik. Skupno število točk ne more preseči 100.

Podatkovni tipi	10 točk
Nadzor toka	15 točke
Spremenljivke	10 točk
Procedure	10 točk
Funkcije	10 točk
Rekurzivne funkcije	10 točk
Optimizacija ovojnic	10 točk
Zahtevani makri	15 točk
*Mutacije in vzajemna rekurzija	*10 točk
slog kode	5 točk
ustni zagovor	5 točk
skupaj	100 točk

Dodatek – primeri programov v jeziku *FR*

```
> (add (mul (true) (true)) (false))
(add (mul (true) (true)) (false))

> (fri (add (mul (true) (true)) (false)) null)
(true)

> (is-proper-seq? (.. (zz 1) (.. (zz 2) (empty))))
(is-proper-seq? (.. (zz 1) (.. (zz 2) (empty))))

> (fri (.. (is-proper-seq? (.. (zz 1) (.. (zz 2) (empty))))
           (is-proper-seq? (.. (zz 1) (.. (zz 2) (zz 3))))) null)
(.. (true) (false))

> (fri (vars "a" (cc (qq (zz 1) (zz 2)) (qq (zz -3) (zz 4)))
```

```
             (mul (valof "a") (valof "a"))) null)
(cc (qq (zz -5) (zz 16)) (qq (zz -3) (zz 4)))

> (fri (vars (list "a" "b")
             (list (cc (qq (zz 1) (zz 2)) (qq (zz -3) (zz 4)))
                   (~ (cc (qq (zz 1) (zz 2)) (qq (zz -3) (zz 4)))))
             (add (valof "a") (valof "b"))) null)
(cc (qq (zz 1) (zz 1)) (qq (zz 0) (zz 1)))

> (fri (call (fun "fib" (list "n")
                  (if-then-else (leq? (valof "n") (zz 2))
                                (zz 1) (add (call (valof "fib")
                                                  (list (add (valof "n") (zz -1))))
                                            (call (valof "fib")
                                                  (list (add (valof "n") (zz -2)))))))
             (list (zz 10))) null)
(zz 55)

> (fri (all? (.. (true) (.. (leq? (false) (true))
                            (.. (=? (.. (zz -19) (zz 0))
                                    (.. (left (add (qq (zz 1) (zz 5)) (zz -4)))
                                        (zz 0)))
                                (empty)))))
       null)
(true)

> (fri (vars (list "a" "b" "c")
             (list (zz 1) (zz 2) (zz 3))
             (fun "linear" (list "x1" "x2" "x3")
                  (add (mul (valof "a") (valof "x1"))
                       (add (mul (valof "b") (valof "x2"))
                            (mul (valof "c") (valof "x3")))))) null)
(closure (list (cons "a" (zz 1))(cons "b" (zz 2)) (cons "c" (zz 3)))
         (fun "linear" '("x1" "x2" "x3")
              (add (mul (valof "a") (valof "x1"))
                   (add (mul (valof "b") (valof "x2"))
                        (mul (valof "c") (valof "x3"))))))

> (fri (add (qq (zz 9) (zz 9)) (true)) null)
. . add: wrong type argument
```