Tečajevi naprednog računarstva u Križevcima

HACK2020 | Hub for Advanced Computing Križevci

Web.Igra.Haskell

Bilješke s obrazovnog programa održanog u sklopu projekta HACK 2020.

Marko Dimjašević

Organizator:

udruga point križevci

Pokrovitelj:

Embassy of the United States
Zagreb - Croatia

Križevci, 2021.

Izvorni tekst dokumenta: a605bd3, 2021-08-20 16:32:26 +0200

Contents

1 Sažetak

Ove bilješke nastale su u sklopu obrazovnog programa *Web.Igra.Haskell* koji je dio programa Hub for Advanced Computing Križevci 2020. Bilješke daju uvid u funkcijski programski jezik Haskell i pobliže upoznaju čitatelja s jezikom kroz izradu web igre. Obrazovni program pokrio je široko područje u relativno malom broju radionica, od koncepta programiranja, preko izraza, pridruživanja varijabli i funkcija do web poslužitelja i komunikacije putem protokola HTTP. Bilješke sadrže poveznice na opširnije materijale za pojedine teme. Konačna implementacije igre ostvarene kroz obrazovni program je javno i slobodno dostupna na webu.

2 | Uvod u Haskell

Računalo je naprava koja izvodi točno određeni skup radnji. Koja je sljedeća radnja koju u nekom trenu treba izvesti računalo određena je naredbom koja je pohranjena u memoriji računala. Te radnje čovjek može programirati, odnosno odrediti koje radnje računalo treba obaviti. Tako možemo računalu zadati da obavi neku nama korisnu radnju poput provedbe kompleksnog izračuna, ali također može poslužiti i za zabavu u obliku računalnih igara. Drugim riječima, moguće je napisati program koji računalu daje instrukcije što valja napraviti. Svako računalo razumije vrlo elementaran jezik kojim mu se daju naredbe za radnje, odnosno u kojem je moguće napisati program. Taj se jezik još naziva strojni jezik. Dakle, s računalom čovjek može jednosmjerno komunicirati putem tzv. programskog jezika. Da je računalo shvatilo naredbu vidimo manifestacijom promjene stanja računala, recimo prikazom poruke na ekranu računala. Dok je takva komunikacija u teoriji moguća i u početku računarstva se tako obavljala, odnosno tako su se programirale radnje za računalo, s vremenom su se razvili programski jezici koji su imali sve manje elemente specifične za samo računalo, odnosno naredbe su postale sve apstraktnije, a time i ljudima pogodnije za programiranje računala.

Haskell je tzv. viši programski jezik, odnosno dovoljno apstraktan jezik da programer, odnosno čovjek koji računalu zadaje radnje u obliku programa, ne mora brinuti o specifičnostima računala i njegove takozvane arhitekture. U Haskellu je programer oslobođen i drugih briga poput zauzimanja i oslobađanja računalne memorije potrebne za izvođenje, odnosno izvršavanje programa. Haskell je nastao krajem 1980-ih godina u vrijeme kada se intenzivno istraživalo i razvijalo funkcijske programske jezike. Funkcijski programski jezici su alternativa tada, no i danas dominantnoj paradigmi imperativnog programiranja i pripadnih imperativnih programskih jezika. Primjeri danas široko zastupljenih imperativnih jezika su C, C++, Java, Python i JavaScript. Dok spomenuti imperativni jezici imaju određene elemente funkcijskih programskih jezika, tek jezike kao što su Scheme, Racket, Erlang, OCaml i Haskell smatramo funkcijskim.

Ono što je karakteristično za funkcijske programske jezike jest da se programi sastoje od primjene i kompozicije funkcija. Može se reći da je sve u funkcijskim programskim jezicima funkcija! Tako funckije preslikavaju vrijednosti u druge vrijednosti i cijeli program je velika kompleksna funkcija koja ulaznu vrijednost pretvara u konačnu vrijednost. Ta jednostavnost ideje funkcijskih jezika u izravnoj je poveznici s algebrom, granom matematike koja se uči u redovnoj školskoj nastavi i nudi prednost u razumijevanju programa u odnosu na one pisane

u imperativnim jezicima. S druge strane imperativni programski jezici program vide kao niz naredbi koje mijenjaju stanje računala dok se program izvršava. Utoliko su imperativni jezici bliži prirodi računala kakva danas poznajemo: imperativno izvršavanje jedne po jedne naredbe i izmjena trenutnog stanja. Unatoč tome, programi pisani u modernim funkcijskim programskim jezicima efikasno se prevode i izvode u strojnom jeziku na računalima.

Programski jezik Haskell naziv je dobio po matematičaru Haskellu Curryju, dok je temelje funkcijske paradigme dao matematičar Alonzo Church tridesetih godina 20. stoljeća definiranjem lambda računa. Lambda račun je minimalni formalni sustav za izražavanje izračuna putem funkcija i pomoću njega može se opisati svaki program, odnosno kažemo da može simulirati svaki Turingov stroj. Račun se sastoji od samo tri konstrukta: uvođenje funkcije (eng. abstraction), primjena funkcije (eng. application) i uvođenje varijabli (eng. variable binding). Izračun funkcija se odvija putem redukcije u kojoj ključnu ulogu ima supstitucija, odnosno zamjena izraza njima jednakim izrazima.

Haskell je poznat po svom tipskom sustavu u smislu teorije tipova, gdje o tipovima možemo razmišljati kao o kategorijama vrijednosti. Jezik se ističe od drugih poznatih funkcijskih programskih jezika po tzv. čistoći, odnosno inzistiranju da je sve funkcija, slično funkciji u matematici. U usporedbi s drugim programskim jezicima, Haskell je relativno slabo zastupljen u primjeni. Unatoč tome postoji krug ljudi koji ga razvijaju i svakodnevno koriste za razne svrhe, uključujući za izradu hardvera. Centralni alat prevođenih programskih jezika je kompajler, odnosno prevodilac, i to je slučaj i kod Haskella. Najkorišteniji prevodilac Haskella u druge jezike, uključujući u strojne jezike, je Glasgow Haskell Compiler (GHC).

3 Instalacija programskih alata

Haskell i popratne programske alate moguće je instalirati na nekoliko načina. Alati za Haskell još su u intenzivnom razvoju pa njihova instalacija nije uvijek najjednostavnija, odnosno nisu uvijek dostupni u obliku sistemskih paketa za distribucije GNU/Linuxa. Slijedi detaljan opis za instalaciju svakog od potrebnih alata.

3.1 Stack

Ovdje ćemo pokazati kako instalirati Stack unutar dvije distribucije GNU/Linuxa. Stack je alat za upravljanje paketima i projektima u Haskellu. U Fedori potrebno je u naredbenom retku izvršiti ove dvije naredbe:

```
sudo dnf update
sudo dnf install stack
```

dok je u Debianu (i svim derivativnim distribucijama, uključujući Ubuntu) potrebno izvršiti:

```
sudo apt update
sudo apt install haskell-stack
```

To će ažurirati popis sistemskih paketa i potom instalirati Stack. Distribucije nemaju uvijek najnoviju inačicu paketa pa je sljedećom naredbom moguće instalirati najnoviji Stack, no to nije neophodan korak:

```
stack upgrade
```

Instaliranu inačicu moguće je doznati naredbom:

```
stack --version
```

Iako to nije potrebno raditi, moguće je zasebno instalirati prevodilac GHC na ovaj način:

```
stack --resolver lts-16.25 setup
```

3.2 VSCodium

Uobičajeno je u razvoju softvera koristiti razvojno okruženje. Čini ga skup alata koji se koriste za uobičajene radnje poput pisanja programskog koda, prevođenja, organizacije koda, izvršavanja i testiranja. Tendecija je razvoja generičkih razvojnih okruženja koja se mogu prilagoditi za pojedini programski jezik. Jedno takvo često korišteno okruženje je VSCode. Iako je izvorni kod VSCodea slobodan softver, njegova izvršna inačica nije pa ćemo mi koristiti varijantu VSCodea poznatu pod nazivom VSCodium. Upute za instalaciju preuzete su s https://vscodium.com/.

Da bi se instalirao VSCodium u Debianu, potrebno je odobriti kriptografski ključ, dodati repozitorij i potom instalirati paket `vscodium`:

```
wget -qO - \
  https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/-/raw/master/pub.gpg \
  | gpg --dearmor | sudo dd of=/etc/apt/trusted.gpg.d/vscodium.gpg
echo 'deb https://paulcarroty.gitlab.io/vscodium-deb-rpm-repo/debs/ vscodium main' \
  | sudo tee --append /etc/apt/sources.list.d/vscodium.list
sudo apt update && sudo apt install codium
```

U Fedori je za isti ishod potrebno izvršiti:

```
sudo rpm --import \
  https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/-/raw/master/pub.gpg
printf "[gitlab.com_paulcarroty_vscodium_repo]
  \nname=gitlab.com_paulcarroty_vscodium_repo
  \nbaseurl=https://paulcarroty.gitlab.io/vscodium-deb-rpm-repo/rpms/
  \nenabled=1
  \ngpgcheck=1
  \nrepo_gpgcheck=1
 \ngpgkey=https://gitlab.com/paulcarroty/vscodium-deb-rpm-repo/-/raw/master/pub.gpg" \
  |sudo tee -a /etc/yum.repos.d/vscodium.repo
sudo dnf install codium
```

Preostalo je još instalirati proširenje (ekstenziju) koje omogućava razvoj u Haskellu unutar VSCodiuma:

```
codium --install-extension haskell.haskell
```

3.3 Haskell Language Server

Iako nije neophodan, u razvoju itekako može biti od koristi Haskell Language Server (skraćeno HLS). Radi se o poslužitelju koji se cijelo vrijeme izvršava u pozadini i služi za komunikaciju

između razvojnog okruženja — u našem slučaju VSCodiuma — i prevodilaca što je u našem slučaju GHC. HLS tako kontinuirano prevodi program koji mijenjamo i javlja razvojnom okruženju ima li grešaka. Također javlja druge korisne informacije poput dokumentacije funkcije pozicioniranjem miša unutar razvojnog okruženja na naziv funkcije.

U razvoju igre Connect4 koristit ćemo inačicu prevodilaca GHC 8.8.4. S web stranice Haskell Language Servera potrebno je preuzeti inačicu HLS-a prilagođenu dotičnom GHC-u. Ako s `${HOME}` označimo korisničku mapu, preuzetu datoteku HLS-a potrebno je smjestiti u mapu:

```
${HOME}/.local/bin/haskell-language-server-Linux-8.8.4
```

U ekstenziji VSCodiuma potrebno je podesiti putanju Server Executable Path na istu tu putanju:

```
${HOME}/.local/bin/haskell-language-server-Linux-8.8.4
```

4 Interaktivna razvojna linija

Funkcijski jezici od svojih početaka imaju interaktivnu razvojnu liniju (eng. read-eval-print loop, REPL), a u novije vrijeme to su počeli uvoditi i imperativni programski jezici. Radi se o programu u komandnoj liniji pomoću kojeg je moguće vršiti interakciju s programskim jezikom, ovdje konkretno s Haskellom. Liniju je moguće pokrenuti davanjem sljedeće naredbe u naredbenom retku:

```
stack repl
```

Jednom kad je REPL pokrenut, moguće je zadavati naredbe i unositi iznose iza znaka, odnosno upita `>`. Svaka linija koja ne počinje tim znakom je rezultat evaluacije naredbe ili izraza. Recimo, unosom izraza `5 + 6` iza upita `>`:

```
> 5 + 6
```

dobivamo ispis broja 11 u sljedećoj liniji što predstavlja rezultat evaluacije unesenog izraza. Moguće je raditi i sa znakovnim nizovima, bilo dobivanjem istog niza za rezultat bilo spajanjem dva niza:

```
> "Dobar dan!"
"Dobar dan!"
> "Danas je dobar dan " ++ "za programiranje u Haskellu!"
"Danas je dobar dan za programiranje u Haskellu!"
```

Unosom naredbe `:info` i navođenjem simbola ili naziva neke funkcije možemo dobiti sažete informacije o unesenom:

```
> :info (++)
(++) :: [a] -> [a] -> [a]  -- Defined in ‘GHC.Base’
infixr 5 ++
```

Simbol `++` naveden je unutar zagrada kako bi se naznačilo da se radi o infiksnom operatoru, odnosno funkciji dva argumenta čiji se naziv može navoditi između operanada i u tom slučaju se navodi bez zagrada.

5 | Prvi pohranjeni programi

Osim pisanja u pravilu vrlo kratkih programa u interaktivnoj razvojnoj liniji, programe je moguće pohraniti u datoteku na disk. Time se otvara mogućnost višekratnog rada na programskom kôdu jer je datoteku nakon pohrane moguće opet učitati i nastaviti s radom.

Obzirom da se netrivijalni programi sastoje od više datoteka, u Stacku, o kojem je bilo riječi u poglavlju 3.1, moguće je koristiti predloške iz kojih će se za korisnika napraviti više potrebnih datoteka. Time se korisnika, odnosno programera oslobađa dosadnih zadataka i omogućava koncentriranje na bitnije stvari. Slijedi primjer izrade jednostavnog projekta u Haskellu kroz predložak `new-template`:

```
stack new helloworld new-template
```

Ovime će se napraviti mapa `helloworld` s više mapa i datoteka:

```
helloworld/
|-- app
|   |-- Main.hs
|-- ChangeLog.md
|-- helloworld.cabal
|-- LICENSE
|-- package.yaml
|-- README.md
|-- Setup.hs
|-- src
    |-- Lib.hs
|-- stack.yaml
|-- test
    |-- Spec.hs
```

Pozicioniranjem u mapu `helloworld` moguće je prevesti dani projekt u strojni kôd i pokrenuti ga:

```
stack run
```

Nakon poruka o prevođenju i pripremi programa, u zadnjem retku ispiše se poruka `someFunc`.

Dotični program izmijenit ćemo tako da funkcija `main` ne poziva funkciju `someFunc` iz modula `Lib` koja ispisuje poruku, već da izravno ispiše poruku "Dobar dan!". Potrebno je u datoteci `app/Main.hs` izmijeniti liniju `main = someFunc` u:

```
main = putStrLn "Dobar dan!"
```

Snimanjem izmijenjene datoteke `app/Main.hs` i ponovnim pokretanjem u komandnoj liniji:

```
stack run
```

dobivamo poruku "Dobar dan!".

5.1 Uvlačenje kôda

U Haskellu je bitno koliko je kôd uvučen, odnosno to određuje ispravnost i značenje kôd a. Sljedeći program je valjan:

```
main :: IO ()
main = do
  putStrLn "Dobar dan!"
  putStrLn "I dovidjenja!"
```

Međutim, sljdeći program gdje je jedna naredba za ispis uvučena više od druge nije valjana:

```
main :: IO ()
main = do
  putStrLn "Dobar dan!"
    putStrLn "I dovidjenja!"
```

GHC će za program s nejednakim brojem uvlačenja prijaviti grešku pri prevođenju.

5.2 Dokumentiranje kôda

U programiranju postoji potreba za dokumentiranjem izvornog kôda iz više razloga. Jedan razlog je da dobijemo sažet uvid u svrhu nekog dijela programa, recimo jedne funkcije, bez da moramo pročitati njezinu cijelu definiciju. U Haskellu je moguće dokumentirati kôd umetanjem posebnih komentara. Obični komentari počinju dvjema povlakama `--` i ostatak linije je komentar, odnosno nije dio kôda i GHC će zanemariti taj ostatak linije tokom prevođenja. Komentari za svrhu dokumentacije počinju linijom koja sadrži niz `-- |`. Na primjer, ovo je funkcija koja vraća dvostruku vrijednost cijelog broja i ispred njenog potpisa i definicije naveden je njen opis:

```
-- | Funkcija izracunava dvostruku vrijednost argumenta. Recimo,
-- duplo 5 ce izracunati 10.
duplo :: Int -> Int
duplo x = 2 * x
```

6 Logika igre

Logiku igre, lišene bilo kakvih popratnih detalja poput HTTP protokola ili ičega vezanog za web, definirat ćemo u ovom poglavlju. Logika će biti dana građenjem tipova specifičnih za igru Connect4, počevši od najjednostavnijih pa sve prema složenijim tipovima i pripadnim vrijednostima i funkcijama.

Za početak ćemo definirati podatkovni tip koji predstavlja žeton igrača. Igra je u dvoje pa postoje dvije vrste žetona. Radi jednostavnosti implementacije, uvodimo i treći žeton `Prazno` koji predstavlja prazno polje na ploči.

```
data Zeton = Crveni | Plavi | Prazno deriving (Eq, Ord, Show)
```

U radu s žetonima potrebno je znati je li žeton određene boje pa za to uvodimo tri funkcije:

```
-- | Funkcija provjerava je li dani žeton crvene boje.
jeLiZetonCrveni :: Zeton -> Bool
jeLiZetonCrveni Crveni = True
jeLiZetonCrveni Plavi  = False
jeLiZetonCrveni Prazno = False

-- | Funkcija provjerava je li dani žeton plave boje.
jeLiZetonPlavi :: Zeton -> Bool
jeLiZetonPlavi Crveni = False
jeLiZetonPlavi Plavi  = True
jeLiZetonPlavi Prazno = False

-- | Funkcija provjerava je li dani žeton prazan.
jeLiPrazno :: Zeton -> Bool
jeLiPrazno Crveni = False
jeLiPrazno Plavi  = False
jeLiPrazno Prazno = True
```

Stupac na ploči predstavljamo tipom `Stupac` koji je isto što i lista žetona. Dogovorno ćemo u stupcu držati najviše 6 žetona (najmanje je 0 i tad je stupac prazan). Stupac je popunjen ako na vrhu nije `Prazno`.

```
type Stupac = [Zeton]

-- | Vrijednost koja predstavlja prazan stupac.
prazanStupac :: Stupac
prazanStupac = [Prazno, Prazno, Prazno, Prazno, Prazno, Prazno]

-- | Funkcija provjerava je li dani stupac popunjen, odnosno ima li 6
-- žetona.
popunjenStupac :: Stupac -> Bool
popunjenStupac s = if (s !! 0) == Prazno then False else True
```

Potrebna nam je funkcija koja ubacuje žeton u stupac. Ako je stupac pun, ne događa se ništa, odnosno funkcija vraća nepromijenjen stupac. Ako u stupcu još ima mjesta, dodaje žeton na vrh stupca:

```
ubaciUStupac :: Zeton -> Stupac -> Stupac
ubaciUStupac z s =
  if popunjenStupac s
    then s
    else
      let (prazniDio, puniDio) = span jeLiPrazno s
       in (drop 1 prazniDio) ++ [z] ++ puniDio
```

Igraća ploča sastoji se od liste `Stupaca`. Dogovorno podrazumijevamo da ploča ima točno 7 stupaca.

```
type Ploca = [Stupac]

-- | Vrijednost koju predstavlja praznu ploču, a čini ju 7 praznih
-- stupaca.
praznaPloca :: Ploca
praznaPloca =
  [ prazanStupac
  , prazanStupac
  , prazanStupac
  , prazanStupac
  , prazanStupac
  , prazanStupac
  , prazanStupac
  ]
```

Sad kad imamo funkciju za ubacivanje žetona u stupac, potrebna nam je funkcija koja ubacuje žeton u ploču. Prvi argument kaže koje boje je žeton, drugi argument kaže u koji stupac po redu ubacujemo (0. stupac je skroz lijevo, a 6. stupac je skroz desno), a treći argument je ploča u koju ubacujemo. Rezultat funkcije je nova ploča s ubačenim žetonom. Kao i kod funkcije `ubaciUStupac`, ako pokušamo ubaciti žeton u već popunjen stupac, ploča ostaje nepromijenjena.

```
ubaciUPlocu :: Zeton -> Int -> Ploca -> Ploca
ubaciUPlocu z indeks p =
  if indeks < 0 || indeks >= 7
    then p
    else
      let (doStupca, odStupca) = splitAt indeks p
          stupac = p !! indeks
          noviStupac = ubaciUStupac z stupac
      in doStupca ++ [noviStupac] ++ drop 1 odStupca
```

Kako bismo znali je li došlo do kraja igre, potrebna je funkcija koja provjerava jesu li sva polja na ploči popunjena. Ako jesu, vraća `True`, inače vraća `False`.

```
jeLiPopunjenaPloca :: Ploca -> Bool
jeLiPopunjenaPloca p = if all popunjenStupac p then True else False
```

Zatim su nam potrebne funkcije, po jedna za crvenu i jedna za plavu boju žetona, koje provjeravaju ima li igrač s dotičnom bojom žetona pobjedu na vrhu stupca, odnosno jesu li najgornja četiri žetona te boje.

```
jeLiPobjedaCrvenogNaVrhuStupca :: Stupac -> Bool
jeLiPobjedaCrvenogNaVrhuStupca s =
  let (crveniNiz, _ostalo) = span jeLiZetonCrveni s
      duljina = length crveniNiz
  in if duljina >= 4 then True else False


jeLiPobjedaPlavogNaVrhuStupca :: Stupac -> Bool
jeLiPobjedaPlavogNaVrhuStupca s =
  let (plaviNiz, _ostalo) = span jeLiZetonPlavi s
      duljina = length plaviNiz
  in if duljina >= 4 then True else False
```

Osim na vrhu stupca, svaki od igrača može imati pobjedu u bilo kojem dijelu stupca. Prvi argument narednih funkcija je stupac u kojem se vrši provjera. Ako igrač ima pobjedu, funkcija vraća `True`, inače vraća `False`.

```
jeLiPobjedaCrvenogUStupcu :: Stupac -> Bool
jeLiPobjedaCrvenogUStupcu s =
  let sviSufiksi = tails s
  in any jeLiPobjedaCrvenogNaVrhuStupca sviSufiksi

jeLiPobjedaPlavogUStupcu :: Stupac -> Bool
jeLiPobjedaPlavogUStupcu s =
  let sviSufiksi = tails s
  in any jeLiPobjedaPlavogNaVrhuStupca sviSufiksi
```

Sljedeća funkcija dohvaća žeton u danom retku, gdje je 0. redak pri vrhu ploče. Ako ne postoji žeton na tom mjestu, funkcija vraća `Nothing`, a ako postoji vrati taj žeton unutar `Just`-a. Funkciju možemo iskoristiti i za izdvajanje stupca iz ploče.

```
naMjestu :: Int -> [a] -> Maybe a
naMjestu indeks _ | indeks < 0 = Nothing
naMjestu indeks s              = listToMaybe (drop indeks s)
```

Osim u stupcima, pobjedu je moguće postići slaganjem četiri žetona u redu ili po dijagonali. Za to uvodimo ukupno osam smjerova pomaka za potragu četiri istovrsna žetona u nizu. Potreban nam je indeks smjera relativnog pomaka u listi `pomak`. Valjana vrijednost je od 0 do uključivo 7.

```
type Smjer = Int

gore, goreDesno, desno, doljeDesno, dolje, doljeLijevo, lijevo, goreLijevo :: Int
gore = 0
goreDesno = 1
desno = 2
doljeDesno = 3
dolje = 4
doljeLijevo = 5
lijevo = 6
goreLijevo = 7
```

Nakon indeksa, potrebni su sami relativni pomaci. Definiramo listu relativnih pomaka u 8 mogućih smjerova. Prva koordinata odgovara stupcu, a druga retku.

```
pomak :: [(Int, Int)]
pomak =
  [ (0, -1)  -- gore
```

```
  , (1, -1)  -- gore-desno
  , (1, 0)   -- desno
  , (1, 1)   -- dolje-desno
  , (0, 1)   -- dolje
  , (-1, 1)  -- dolje-lijevo
  , (-1, 0)  -- lijevo
  , (-1, -1) -- gore-lijevo
  ]

-- | Broj koraka za koji radimo relativni pomak.
type BrojKoraka = Int
```

Za dani smjer i broj koraka, potrebno je izračunati indeks stupca i retka obzirom na početnu poziciju:

```
pomakUSmjeru :: Smjer -> BrojKoraka -> Int -> Int -> (Int, Int)
pomakUSmjeru smjer brojKoraka indeksStupca indeksRetka =
  ( indeksStupca + brojKoraka * fst (pomak !! smjer)
  , indeksRetka  + brojKoraka * snd (pomak !! smjer)
  )
```

Zatim nam je potrebna funkcija koja izdvaja niz od najviše 4 žetona u danom smjeru počevši od vrha stupca određenim indeksom stupca.

```
nizZetona :: Int -> Ploca -> Smjer -> Stupac
nizZetona indeksStupca p smjer =
  let polazniStupac = (p !! indeksStupca)
      indeksRetka = length (takeWhile jeLiPrazno polazniStupac)
   in if indeksRetka >= 6
    then []
    else
      let pomaci =
            [ pomakUSmjeru smjer 1 indeksStupca indeksRetka
            , pomakUSmjeru smjer 2 indeksStupca indeksRetka
            , pomakUSmjeru smjer 3 indeksStupca indeksRetka
            ]
          stupac1 = naMjestu (fst (pomaci !! 0)) p
          stupac2 = naMjestu (fst (pomaci !! 1)) p
          stupac3 = naMjestu (fst (pomaci !! 2)) p
          mZeton1 = stupac1 >>= naMjestu (snd (pomaci !! 0))
          mZeton2 = stupac2 >>= naMjestu (snd (pomaci !! 1))
```

```
        mZeton3 = stupac3 >>= naMjestu (snd (pomaci !! 2))
        nizMogucihZetona =
          [ Just ((p !! indeksStupca) !! indeksRetka)
          , mZeton1
          , mZeton2
          , mZeton3
          ]
        stvarniNiz = catMaybes (takeWhile isJust nizMogucihZetona)
    in stvarniNiz
```

Također nam je potrebna funkcija koja izdvaja niz od najviše 4 žetona u danom smjeru počevši od vrha stupca određenim indeksom stupca te na to dodaje niz od najviše 4 žetona u suprotnom smjeru.

```
nizZetonaObosmjerno :: Int -> Ploca -> Smjer -> Stupac
nizZetonaObosmjerno indeksStupca p smjer =
  let niz1 = nizZetona indeksStupca p smjer
      suprotniSmjer = (smjer + 4) `mod` 8
      niz2 = nizZetona indeksStupca p suprotniSmjer
   in reverse (drop 1 niz1) ++ niz2
```

Slijede dvije gotovo identične funkcije, no svaka je specijalizirana za žetone ili crvene ili plave boje. Obje funckije provjeravaju ima li igrač s danom bojom žetona pobjedu na ploči ubacivanjem žetona u dani stupac.

Ako igrač ima pobjedu, dotična funkcija vraća `True`, inače vraća `False`.

```
jeLiPobjedaCrvenog :: Ploca -> Int -> Bool
jeLiPobjedaCrvenog p indeksStupca =
  let smjerovi = [0, 1, 2, 3] -- [0, 1, 2, 3]
      nizovi = fmap (nizZetonaObosmjerno indeksStupca p) smjerovi
      imaLiPobjedu = any jeLiPobjedaCrvenogUStupcu nizovi
   in imaLiPobjedu


jeLiPobjedaPlavog :: Ploca -> Int -> Bool
jeLiPobjedaPlavog p indeksStupca =
  let smjerovi = [0, 1, 2, 3] -- [0, 1, 2, 3]
      nizovi = fmap (nizZetonaObosmjerno indeksStupca p) smjerovi
      imaLiPobjedu = any jeLiPobjedaPlavogUStupcu nizovi
   in imaLiPobjedu
```

Za funkcije `jeLiPobjedaCrvenog` i `jeLiPobjedaPlavog` možemo reći da su središnje funkcije logike igre koje provjeravaju je li igra dovršena. Čim bilo koja od funkcija vrati `True`, znamo da

je igra gotova. Igra također može završiti neriješeno i za takvu ploču funkcija `jeLiPopunjenaPloca` također vraća `True`.

Funkcije iz ovog poglavlja koristit će se kasnije u kôdu koji pripada poglavlju 8.3 o rukovateljima web sučelja.

7 | Web programiranje

World Wide Web ili skraćeno Web je sutav dokumenata. Izumio ga je engleski znanstvenik Timothy Berners-Lee 1989. godine radeći u Europskom vijeće za nuklearna istraživanja (CERN) u Ženevi. Svaki dokument određen je tzv. jedinstvenim lokatorom resursa (eng. Unique Resource Locator, URL). Na primjer, `https://hack.udruga-point.hr/` određuje web stranicu projekta Hub for Advanced Computing Križevci.

Resursi na webu se prenose putem protokola Hypertext Transfer Protocol (HTTP). Resurse poslužuju web poslužitelji, a primaju ih web klijenti. Primjeri web poslužitelja su programi nginx i Apache Web Server, dok su primjeri web klijenata web preglednici kao što su Mozilla Firefox i lynx te komandnolinijski alati kao što su curl i wget. Za moderni Web neophodni su i brojni drugi standardizirani protokoli kao što su Hypertext Transfer Protocol Secure (HTTPS), Domain Name System (DNS), Transport Layer Security (TLS), Transmission Control Protocol (TCP) i User Datagram Protocol (UDP). Također su presudni standardizirani (otvoreni) datotečni formati poput HyperText Markup Language (HTML), Cascading Style Sheets (CSS), Extensible Markup Language (XML), Portable Network Graphics (PNG) i JavaScript Object Notation (JSON).

Upravo zato što je Web određen standardiziranim protokolima i datotečnim formatima, ne ovisi o samo jednom softverskom programu ili dobavljaču. Web je podržan na brojnim operacijskim sustavima, uključujući GNU/Linux, Windows, Android, OS X, iOS, BSD i dr. Iz istog razloga za korištenje weba moguće je koristiti bilo koji web preglednik, recimo Firefox, Chrome ili Safari. Sve navedeno ujedno čini Web odličnom platformom za izvršavanje i distribuciju softvera. U sklopu HACK-a razvijena je web igra Connect4 te igri igrači pristupaju putem URL-a, a izvršavanje igre se odvija kroz interakciju poslužitelja i web klijenta, npr. Firefoxa.

7.1 Pristupanje web stranici

Web stranica je dokument na webu kojoj možemo pristupiti web preglednikom otvaranjem danog URL-a (eng. Uniform Resource Locator), odnosno poveznice. Komponente URL-a, tj. poveznice su:

1. Protokol (recimo http) iza kojeg slijedi dvotočka i dvije kose crte `://`,

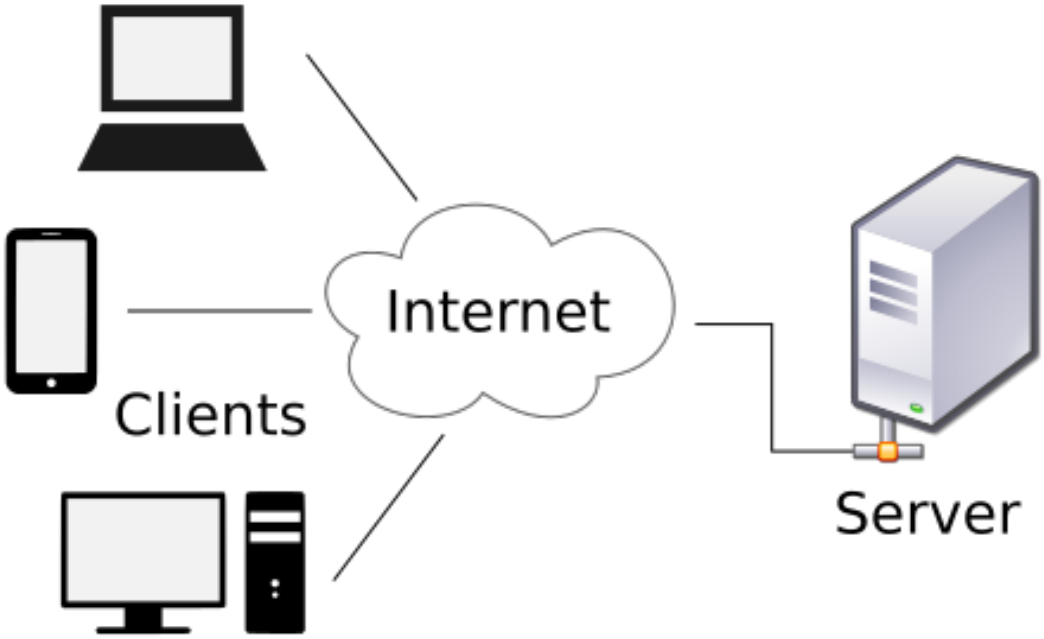

Figure 7.1: Model klijent-poslužitelj. (c) David Vignoni, 2011., licenca LGPL v2.1.

2. Neobvezni korisnički podaci, no ako su navedeni, iza njih slijedi znak `@`,
3. Računalo, koje može biti dano na jedan od dva načina:
   - simboličnim nazivom još poznatim kao domena, npr. wikipedia.org,
   - Internet Protocol (IP) adresom, npr. 91.198.174.192,
4. Neobvezan priključak (eng. port), npr. 80,
5. Putanja,
6. Neobvezni upit (eng. query) koji slijedi nakon znaka `?`,
7. Neobvezni fragment (eng. fragment) koji slijedi nakon znaka .

Ako za primjer uzmemo URL `https://en.wikipedia.org/wiki/World_Wide_Web#History`, protokol je `https`, nema korisničkih podataka, računalo je dano domenom `en.wikipedia.org`, podrazumijevani priključak obzirom na protokol je 443, putanja je `/wiki/World_Wide_Web`, nema upita te je fragment `History`.

7.2 Arhitektura poslužitelj-klijent

U razvoju softvera, pa tako i razvoja softvera koji funkcionira pomoću Weba, od značaja je arhitektura poslužitelj-klijent. U njoj postoje dvije odvojene uloge, odnosno strane gdje

svaka obavlja svoju zadaću i komunicira s drugom stranom. Analogija poslužitelja i klijenta u računarstvu je gostovanje mušterije u restoranu. Po dolasku mušterije u restoran, konobar prima narudžbu od mušterije, obrađuje narudžbu (priprema naručeno piće i hranu) te potom poslužuje naručeno. Mušterija tada konzumira ono što je naručila.

Slično je u računarstvu. Poslužitelj (eng. server) je računalo ili program koji poslužuje sadržaj, resurs ili aplikaciju. Primjer poslužitelja su poslužitelj elektronske pošte i poslužitelj web stranica. Klijent (eng. client) je računalo ili program koji prima sadržaj, resurs ili koristi aplikaciju. Primjer klijenta je aplikacija za elektronsku poštu ili pak web preglednik na mobitelu. Komunikacija web poslužitelja i klijenta odvija se putem standardnih protokola poput HTTP-a. Model klijent-poslužitelj dan je slikom 7.1.

U igri Connect4 web poslužitelj će obrađivati zahtjeve prema završnim točkama te posluživati odgovore, odnosno sadržaj koji će klijent koristiti za interpretaciju stanja igre i prikaz u obliku web stranice. U sklopu ovog obrazovnog programa definirat će se protokol za igranje igre. To će biti tzv. Web API (eng. Application Programming Interface), odnosno sučelje pomoću kojeg klijent, odnosno igrač koumunicira s poslužiteljem, npr. javljanjem poteza koji želi odigrati ili upitom u trenutno stanje igre. Sučelje se sastoji od više završnih točaka i svaka ima svoju namjenu. Više o tome dano je u poglavlju 8.

7.3 Pokretanje programa

Web programe koje pišemo moguće je pokrenuti na lokalnom računalu. Kada smo u komandnoj liniji pozicionirani u mapi projekta, dovoljno je pokrenuti:

```
stack run
```

To će pokrenuti naš web program. Recimo, u primjeru igre Connect4, naredba će na lokalnom računalu, označenim domenskim imenom `localhost` ili pripadnom IP adresom 127.0.0.1, na priključku 8080 pokrenuti igru. Igri je moguće pristupiti u web pregledniku, recimo Firefoxu, otvaranjem neke od završnih točaka (eng. endpoint) u korijenu aplikacije koji je u našem slučaju `http://localhost:8080/`. Na primjer, jednom kad pokrenemo dovršenu igru, bit će moguće pristupiti jednoj od završnih točaka na adresi `http://localhost:8080/sve-igre`.

8 Web sučelje

Igru Connect4, a tako i bilo koju web aplikaciju, moguće je definirati putem Web sučelja (eng. Web API). To sučelje opisuje radnje koje je moguće napraviti u igri. Sučelje će se koristiti putem protokola HTTP, no za potrebe ovog obrazovnog programa ne moramo poznavati pozadinske detalje funkcioniranja protokola. Putem sučelja razmjenjivat će se datoteke tipa JavaScript Object Notation (JSON) i svaka će datoteka imati unaprijed propisanu strukturu kako bi poslužitelj i klijent imali unaprijed usklađen oblik komunikacije. Za definiranje web sučelja koristit ćemo tzv. ugrađeni programski jezik specifičan za definiranje domene web sučelja (eng. Embedded Domain-specific Language, eDSL). Taj je jezik ugrađen u Haskell, odnosno definiran je u Haskellu, a jezik definira skup biblioteka Haskella zvan Servant.

8.1 Biblioteke Servant

Servant je skup biblioteka za pisanje web aplikacija u Haskellu. Jedna od pogodnosti Servanta je što je moguće iz definicije Web sučelja automatski generirati dokumentaciju i time doskočiti bolnom problemu u programiranju, a to je zastarjela dokumentacija koja ne odgovara aktualnoj inačici neke implementacije. Okosnica Servanta je korištenje tipova u Haskellu. Tako će, primjerice, svaka završna točka imati svoj tip. Zahvaljujući tipovima će GHC, prevodilac Haskella, moći provjeriti da opis završnih točaka Web sučelja odgovara pripadnim funkcijama za obradu zahtjeva prema tim završnim točkama. Ukoliko ne postoji podudarnost, naša igra neće se moći ni prevesti u strojni kôd i GHC će prijaviti greške, čime ćemo spriječiti korištenje neispravne igre. Prije nego ćemo ju moći pokrenuti, morat ćemo ispraviti greške.

Servant čini više biblioteka, no ovdje je bitna samo središnja biblioteka za pisanje poslužitelja — `servant-server`. U njoj su dani tipovi i funkcije za pisanje web poslužitelja. Mi ćemo koristiti tu biblioteku za pisanje poslužitelja igre Connect4.

8.2 Sučelje igre

U igri Connect4 definirat ćemo ukupno pet završnih točaka (eng. endpoint). Svaka završna točka ima svoje značenje, odnosno namjenu. U konačnici ćemo objediniti svih pet završnih točaka u Web sučelje, odnosno Web sučelje je skup završnih točaka.

Prvo ćemo definirati završnu točku `GET /sve-igre`. GET je metoda zahtjeva u protokolu HTTP prema poslužitelju koja dohvaća prikaz navedenog resursa, u našem slučaju resursa danim putanjom `/sve-igre`. Ova će završna točka klijentu dati popis svih otvorenih igara, odnosno igara u kojoj je prisutan jedan igrač i koji čeka na drugog igrača da se pridruži kako bi igra mogla započeti. Završna je točka dana sljedećim tipom u Haskellu:

```
type SveOtvoreneIgre =
     Summary "Dohvat popisa svih otvorenih igara."
  :> "sve-igre"
  :> Get '[JSON] [InfoOtvorenaIgra]
```

Ova se završna točka sastoji od tri komponente odvojene sintaksnim članom `:>` koji je dio eDSL-a Servanta. Prva komponenta pomoću `Summary` daje kratki tekstualni opis završne točke. Druga je komponenta putanja `/sve-igre` gdje se uvodni znak `/` pretpostavlja. Zadnja komponenta kaže da se radi o HTTP metodi GET, da je odgovor zapisan u datotečnom formatu JSON, te da se odgovor sastoji od liste podataka o otvorenim igrama.

Sljedeća je završna točka `POST /nova` za otvaranje nove igre navođenjem imena prvog igrača. Dana je ovim tipom u Haskellu:

```
type NovaIgra =
     Summary "Otvaranje nove igre."
  :> "nova"
  :> ReqBody '[JSON] ImeIgraca
  :> Post '[JSON] PoluInicijaliziranaIgra
```

Ova se točka sastoji od četiri komponente. Prva daje opis igre. Druga komponenta daje putanju `/nova`. Treća nam komponenta kaže da je u tijelu zahtjeva tipa POST potrebno navesti ime prvog igrača. Zadnja komponenta kaže da se radi o zahtjevu s metodom POST čiji je odgovor dan datotečnim formatom JSON, a sadržaj je informacija o poluinicijaliziranoj igri: sadrži identifikator novo otvorene igre i tajni token prvog igrača. Za dovršetak igre bit će potrebno koristiti sljedeću završnu točku.

Pomoću završne točke `POST /igra/:id-igre/dovrsetak` moguće je dovršiti otvorenu igru u kojoj je pristuan tek prvi igrač. Na primjer, ako je identifikator igre `nlrkwldbji`, zahtjev se vrši prema `POST /igra/nlrkwldbji/dovrsetak`. Dovršetak inicijalizacije se događa pridruživanjem drugog igrača igri danom identifikatorom `id-igre`, a drugi igrač zauzvrat dobiva jedinstveni tajni token kojim se autorizira poslužitelju igre. Završna je točka dana sljedećim tipom:

```
type DovrsetakNoveIgre =
     Summary "Dovršenje otvorene igre priključivanjem drugog igrača."
  :> "igra"
  :> ReqBody '[JSON] ImeIgraca
  :> Capture "id-igre" IdIgre
```

```
  :> "dovrsetak"
  :> Post '[JSON] IgracevToken
```

Završna se točka sastoji od šest komponenti. Prva komponenta je kratki opis točke. Druga daje prvi dio putanje `/igra`. Treća je komponenta tijelo zahtjeva koje treba biti u datotečnom formatu JSON i treba sadržavati ime drugog igrača koji se pridružuje poluinicijaliziranoj igri. Četvrta je komponenta identifikator igre čija se inicijalizacija dovršava, recimo `nlrkwldbji`. Peta komponenta dodaje dio putanje `/dovrsetak`. Zadnja šesta komponenta kaže da je metoda POST, da je tijelo odgovora datotečnog formata JSON i da sadrži token drugog igrača.

Jednom kad igra postoji, u bilo kojem trenutku možemo saznati njeno stanje. Tome služi završna točka `GET /igra/:id-igre`:

```
type StanjeIgre =
     Summary "Dohvaćanje stanja igre."
  :> "igra"
  :> Capture "id-igre" IdIgre
  :> Get '[JSON] StatusIgre
```

Sastoji se od četiri komponente. Kao i inače, prva komponenta dana konstruktom `Summary` sadrži kratki opis točke. Sljedeća komponenta daje prvi dio putanje, a to je `/igra`. Treća komponenta služi davanju identifikatora igre. Ako je identifikator postojeće igre `nlrkwldbji`, zahtjev bi bio `GET /igra/nlrkwldbji`. Zadnja komponenta određuje da je metoda GET, da je odgovor u datotečnom formatu JSON i da sadrži status igre. Status igre može biti informacija o igri u tijeku, o igri koja je završila neriješeno ili pak o pobjedi jednog od igrača uz završno stanje ploče.

Posljednja završna točka `POST /ubaci-zeton` služi ubacivanju žetona u ploču postojeće igre:

```
type UbaciZeton =
     Summary "Ubacivanje žetona u ploču igre koja je u tijeku."
  :> "ubaci-zeton"
  :> ReqBody '[JSON] UbacivanjeZetona
  :> Post '[JSON] StatusIgre
```

Sastoji se od četiri komponente, od čega je prva uobičajeni opis završne točke. Druga komponenta daje informaciju da je prvi dio putanje ove završne točke `/ubaci-zeton`. Treća komponenta određuje tijelo zahtjeva u datotečnom formatu JSON. Tijelo ovog zahtjeva sadrži indeks stupca u koji valja ubaciti žeton te igračev tajni token. Koji je igrač trenutno na potezu određuje valjanost tajnog tokena: igrač mora biti na potezu da bi mogao valjano koristiti ovu završnu točku, a u suprotnom će dobiti odgovor o grešci.

Sučelje igre dano je "zbrojem" navedenih pet završnih točaka:

```
type APIIgre
  =    SveOtvoreneIgre
  :<|> NovaIgra
  :<|> DovrsetakNoveIgre
  :<|> UbaciZeton
  :<|> StanjeIgre
```

Sintaksna komponenta `:<|>` upućuje na to da je moguć izbor u pozivu točne jedne od navedenih završnih točaka.

8.3 Rukovatelj

Svaka završna točka ima tzv. rukovatelja (eng. handler). Rukovatelj je funkcija koju će poslužitelj izvršiti primitkom zahtjeva. Poslužitelj naše igre može primiti bilo koji od ranije navedenih pet zahtjeva. Baš kao i web sučelje, rukovatelji su objedinjeni u vršnog rukovatelja:

```
posluzitelj
  :: ( Members
       '[ Input Konfiguracija
        , SKV IdIgre Igrac
        , SKV IdIgre StatusIgre
        , Random
        , Error SL.GreskaNovaIgra
        , Error SL.GreskaDovrsetakNoveIgre
        , Error SL.GreskaUbacivanjeZetona
        , Error SL.GreskaStanjeIgre
        ] r
     )
  => ServerT APIIgre (Sem r)
posluzitelj =
       SL.sveOtvoreneIgre
  :<|> SL.novaIgra
  :<|> SL.dovrsetakNoveIgre
  :<|> SL.ubaciZeton
  :<|> SL.stanjeIgre
```

Pojedinačni rukovatelji (recimo `SL.sveOtvoreneIgre` i `SL.novaIgra`) odvojeni su sintaksnom komponentom `:<|>` i moraju biti navedeni istim redoslijedom kao i pripadne završne točke u tipu `APIIgre`. Svaki od rukovatelja je netrivijalni dio kôda i ovdje nećemo ulaziti u njihove detalje. Tek radi primjera navodimo najjednostavnijeg rukovatelja:

```
-- | Dohvaća popis svih otvorenih igara
sveOtvoreneIgre :: Member (SKV IdIgre Igrac) r => Sem r [InfoOtvorenaIgra]
sveOtvoreneIgre = fmap (
  \(idIgre, igrac) -> MkInfoOtvorenaIgra
    { infoOtvorenaIgraIdIgre = idIgre
    , infoOtvorenaIgraPrviIgrac = igracIme igrac
    }
  ) <$> izlistajSveSkv
```

Iz tipa rukovoditelja moguće je vidjeti da kao rezultat vraća listu informacija o otvorenim igrama, a to se podudara s odgovorom ranije navedenog tipa završne točke `SveOtvoreneIgre`.

9 Testiranje softvera

Tokom razvoja igre napisani su i brojni testni primjeri. Ti testni primjeri su također programski kôd koji se sam izvršava i ne podrazumijeva intervenciju razvijatelja. Testiranje softvera sastavni je dio razvoja softvera. Testni su primjeri korisni jer utvrđuju dodatno očekivano ponašanje programa, u ovom slučaju implementacije Connect4, koje nije obuhvaćeno putem tipskog sustava Haskella. Sve testove koji su napisani za Connect4 moguće je izvršiti jednom naredbom u naredbenom retku:

```
stack test
```

Testovi su organizirani u tri cjeline koje označavaju tri zasebna aspekta implementacije: testovi logike igre, testovi sučelja logike potrebnog za interakciju s webom te testovi samog web sučelje.

U pisanju testova korišten je hspec, popularna biblioteka za testiranje. Testove pisane u hspec-u, tzv. specifikaciju, praktički je moguće čitati kao rečenice u engleskom jeziku koristeći sintaksu Haskella i konstrukte koje pruža dotična biblioteka. Na primjer, ovo je niz testnih primjera za koncept ploče dan u poglavlju 6:

```
specPloca :: Spec
specPloca = describe "Ploca" $ do
  it "Ubacivanje zetona" $ do
    ubaciUPlocu Crveni 0 nultiStupacPun `shouldBe` nultiStupacPun
    ubaciUPlocu Plavi 0 p1 `shouldBe`
      ubaciUStupac Plavi s1 : drop 1 p1
  it "Popunjenost ploce" $ do
    jeLiPopunjenaPloca p1 `shouldBe` False
    jeLiPopunjenaPloca p2 `shouldBe` False
    jeLiPopunjenaPloca punaPloca `shouldBe` True
  it "Izdvajanje niza" $ do
    nizZetona 0 p1 gore `shouldBe` [Crveni, Prazno, Prazno, Prazno]
    nizZetona 1 p1 dolje `shouldBe` [Plavi, Plavi, Plavi]
    nizZetona 2 p1 dolje `shouldBe` []
    nizZetona 3 p1 dolje `shouldBe` [Crveni, Plavi, Crveni, Plavi]
```

```
    nizZetona 6 p1 goreLijevo `shouldBe` [Plavi, Plavi, Prazno, Prazno]
    nizZetona 4 p1 lijevo `shouldBe` [Plavi, Plavi, Prazno, Plavi]
    nizZetona 4 p1 desno `shouldBe` [Plavi, Plavi, Prazno]
    nizZetona 3 p2 goreDesno `shouldBe` [Plavi, Prazno, Prazno]
  nizZetonaObosmjerno 3 p1 gore `shouldBe` [Prazno, Prazno, Crveni, Plavi, Crveni, Plavi]
  nizZetonaObosmjerno 3 p1 dolje `shouldBe` [Plavi, Crveni, Plavi, Crveni, Prazno, Prazno]
  nizZetonaObosmjerno 2 punaPloca doljeDesno `shouldBe` [Plavi, Crveni, Plavi, Plavi]
  nizZetonaObosmjerno 4 p2 lijevo `shouldBe` [Plavi, Plavi, Crveni, Plavi, Prazno, Prazno]
 it "Trazenje pobjednickog niza" $ do
    jeLiPobjedaCrvenog p1 3 `shouldBe` False
    jeLiPobjedaCrvenog p3 1 `shouldBe` True
    jeLiPobjedaCrvenog p3 2 `shouldBe` True
    jeLiPobjedaCrvenog p3 3 `shouldBe` True
    jeLiPobjedaCrvenog p3 4 `shouldBe` True
    jeLiPobjedaPlavog punaPloca 3 `shouldBe` True
    jeLiPobjedaPlavog punaPloca 0 `shouldBe` False
```

Pomoću konstrukta `shouldBe`, za danu stvarnu vrijednost koju kôd izračunava temeljem naše implementacije, opisujemo koju vrijednost očekujemo kao rezultat evaluacije izraza, odnosno stvarne vrijednosti. Čim testni primjer ne evaluira u očekivanu vrijednost, Haskell će prijaviti grešku tokom izvršavanja naredbe `stack test`. Time utvrđujemo da smo ili nešto krivo implementirali ili da test treba prilagoditi izmjenama koje su napravljene u implementaciji. Razvijatelj poznavanjem problematike, u ovom slučaju pravila igre Connect4, može utvrditi je li greška u implementaciji ili u testnom primjeru.

10 Zaključak

Obrazovni program Web.Igra.Haskell pokrio je više područja u vrlo kratkom periodu: osnove programiranja, osnove funkcijskog programiranja, programski jezik Haskell, ugrađen programski jezik Servant i osnove web programiranja s ciljem razvoja web igre Connect4. Ove bilješke, baš kao i sam obrazovni program, u sažetom obimu polaznicima obrazovnog programa daju pregled obrađenih tema kao i reference za daljnje proučavanje tema. Za razvoj pozadinskog dijela web igre Connect4 korišten je čisti funckijski jezik Haskell te njegov ugrađeni programski jezik Servant. Pomoću Servanta definirano je web sučelje (API) igre te je dana potpuna i funkcionalna implementacija API-ja. Izvorni kôd igre Connect4 dostupan je na webu na adresi: https://git.hacklab.krizevci.info/hack-2020-haskell/connect4.