UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Matematika – 1. stopnja

Ana Borovac

# Funkcijsko reagentno programiranje

Delo diplomskega seminarja

Mentor: prof. dr. Andrej Bauer

Ljubljana, 2017

Kazalo

## Funkcijsko reagentno programiranje

Povzetek

Iz lambda računa se je razvilo funkcijsko programiranje, katerega glavni element je funkcija. Funkcije ob enakih argumentih vedno vrnejo enak rezultat, kar močno vpliva na nadzor programa, s tem olajša spreminjanje programa in paralelno programiranje. Funkcije lahko tudi podamo kot argument drugi funkciji, jih hranimo v podatkovnih strukturah in jih uporabljamo, kot bi sicer uporabljali druge vrednosti, npr. sezname ali števila. Funkcijsko reagentno programiranje je način programiranja uporabniških vmesnikov in drugih sistemov, ki se odzivajo (reagirajo) na dogodke iz okolja. Vrednosti, ki se spreminjajo s časom, funkcijskemu programiranju dodajo reagentnost. V programskem jeziku Elm so te vrednosti signali. Iz že vgrajenih signalov, ki prihajajo iz zunanjega okolja, lahko ustvarimo nove, ki opisujejo našo aplikacijo. Aplikacija, napisana v Elmu, je sestavljena iz treh delov: modela, posodobitvenega dela in dela, kjer je implementiran prikaz modela.

## Functional reactive programming

Abstract

Functional programming was developed from lambda calculus, where function is the main part. Functions with the same arguments always return the same result which means that the program is easier to control and changing of the program and parallel programming is easier as well. We can add functions as an argument to other function, save them in data structures and we can use them as we would other values, e. g. lists or numbers. Functional reactive programming is the way of programming user interfaces and other systems which react to events from outer world. Values that are changing over time give reactivity to the functional programming. In programming language Elm are those values signals. From already integrated signals, which come from external environment, we can create new signals which describe our application. Application written in Elm consists of three parts: model, update part and the part where view of the model is implemented.

**Math. Subj. Class. (2010):** 68N18

**Ključne besede:** funkcijsko programiranje, funkcijsko reagentno programiranje, lambda račun, signal

**Keywords:** functional programming, functional reactive programming, lambda calculus, signal

# 1. Uvod

Računalništvo se je začelo že pred izumom računalnika z računskimi postopki, kjer je bil človek izvajalec postopka, ki mu danes pravimo algoritem. Formalno je bil algoritem definiran v letih 1935 ter 1936 in to kar trikrat [14]:

- Alonzo Church: Lambda račun (*Nerešeni problem elementarne teorije števil*, maj 1935)
- Kurt Gödel: Rekurzivne funkcije (Stephen Kleene: *Splošne rekurzivne funkcije naravnih števil*, julij 1935)
- Alan Mathison Turing: Turingov stroj (*O izračunljivih številih, z uporabo na Entscheidungsproblem*, november 1936)

Na vse tri definicije je vplival Hilbertov odločitveni problem, *Entscheidungsproblem*. Potem ni tako zelo nenavadno, da so v dveh letih na treh različnih krajih dognali ekvivalenten rezultat. David Hilbert je želel narediti postopek, ki bi za vsako trditev ugotovil, ali je veljavna. Predpostavljal je, da je vsaka veljavna trditev dokazljiva in obratno, da je vsaka dokazljiva trditev veljavna. Nekaj let kasneje je Kurt Gödel našel protiprimer. Če želimo dokazati, da noben postopek ne bo rešil Hilbertovega problema, potrebujemo formalno definicijo postopka.

Prvi, ki je ponudil rešitev, je bil Alonzo Church z lambda računom. Dokazal je, da če je algoritem vse, kar lahko zapišemo z lambda računom, potem ne obstaja splošen algoritem Hilbertovega problema.

Kmalu za Churchem je Kurt Gödel postavil svojo definicijo z rekurzivnimi funkcijami. Church je kasneje dokazal, da sta definiciji ekvivalentni.

Leto kasneje je Turning s svojim Turnigovim strojem prišel še do tretjega ekvivalentnega rezultata. Turingov stroj velja za predhodnika današnjega računalnika, čeprav gre za abstrakten stroj. Stroj ima za razliko od računalnikov neomejen pomnilnik, neskončen trak, ki je razdeljen na celice. Stroj izvaja ukaze, ki so zapisani na traku, drugega za drugim, pri tem lahko tudi zapiše na trak, kot narekujejo ukazi [13].

V razdelku 2 si bomo najprej pogledali, kako je lambda račun vplival na funkcijsko programiranje, kaj to sploh je, kako zelo se funkcijski programski jeziki razlikujejo od ukaznih (npr. Python). V razdelku 3 bomo funkcijskemu programiranju dodali reagentnost in pogledali, kakšno rešitev ponuja programski jezik Elm s signali (razdelka 4 in 6); kakšne operacije lahko delamo s signali, ali so te bistveno drugačne od navadnih funkcij. Osnove Elmove sintakse bomo opisali v razdelku 5. V razdelku 7 si bomo ogledali, kakšno obliko programa zahteva Elm na praktičnem primeru rekurzivnega risanja in na koncu še, kakšna je nereagentna različica Elma (razdelek 8).

Vsi praktični primeri bodo napisani v programskem jeziku Elm 0.16.

# 2. Funkcijsko programiranje

Razdelek je napisan na podlagi [2, 11, 12, 18, 19, 25, 26, 27, 28] in [9, pogl.6].

*Funkcijsko programiranje* (FP) je programska paradigma, ki temelji na lambda računu. Kot že samo ime pove, pri tovrstnem programiranju pišemo funkcije in jih združujemo v aplikacijo.

## 2.1. **Lambda račun.**

*Lambda račun* je veja v logiki, ki definira matematično natančen zapis funkcij. Velja tudi za najmanjši univerzalni programski jezik. Lambda

izraze gradimo iz $\lambda$-abstrakcij, aplikacij, spremenljivk in konstant. Abstrakcije simbolno pišemo kot:

$$\lambda\,\text{argument}\,.\,\text{izraz}$$

Abstrakcija $\lambda\,x\,.\,x$ predstavlja identiteto, analogen, v matematiki uporabnejši zapis bi bil: $x \mapsto x$. Identiteta preslika vrednost samo vase, tako tudi vse abstrakcije eno spremenljivko preslikajo v rezultat, ki je zopet lambda izraz, kar pomeni, da je sestavljen iz $\lambda$-abstrakcij, aplikacij, spremenljivk in konstant.

Aplikacija predstavlja izraz, uporabljen na izrazu. Če se vrnemo na identiteto, bi aplikacijo predstavljala uporabna abstrakcije na določeni vrednosti, vzemimo število 1:

$$\underbrace{(\lambda\,x\,.\,x)}_{\text{izraz}}\ \underbrace{1}_{\text{izraz}}$$

Če imamo več izrazov zapisanih drugega za drugim, upoštevamo dogovor leve asociativnosti, kar bi pomenilo, če so $E_1, E_2, \ldots, E_n$, $n \in \mathbb{N}$, lambda izrazi, potem izraz:

$$E_1 E_2 \cdots E_n$$

izračunamo iz leve proti desni [1]:

$$(\cdots((E_1 E_2)E_3)\cdots E_{n-1})E_n$$

Kako pa zapišemo lambda izraz, ki sešteje dve števili, če $\lambda$-abstrakcije sprejmejo le en argument? Zapisali bi na način:

$$\lambda\,x\,.\,(\lambda\,y\,.\,x + y)$$

Če izraz uporabimo še na številih 1 in 2 ter upoštevamo dogovor leve asociativnosti, dobimo:

$$((\lambda\,x\,.\,(\lambda\,y\,.\,x + y))\,1)\,2 = (\lambda\,y\,.\,1 + y)\,2 = 1 + 2 = 3$$

Izhodiščni izraz lahko zapišemo tudi v poenostavljeni obliki:

$$\lambda\,x\,y\,.\,x + y$$

Upoštevali smo dogovor o desni asociativnosti, kar bi v splošnem pomenilo, da je zapis:

$$\lambda\,x_1\,.\,(\lambda\,x_2\,.\,(\cdots(\lambda\,x_n\,.\,f(x_1, \ldots, x_n))\cdots))$$

ekvivalenten zapisu [1]:

$$\lambda\,x_1 \cdots x_n\,.\,f(x_1, \ldots, x_n)$$

2.2. **Funkcije kot prvovrstne vrednosti.** V Pythonu so cela števila prvovrstne vrednosti, kar pomeni, da jih lahko hranimo v podatkovnih strukturah in spremenljivkah, so argument funkcijam ali vrnjeni rezultat. Primer vrednosti, ki v Pythonu ni prvovrstna, je slovar, ki ga ne moremo podati kot ključ drugemu slovarju. V funkcijskih jezikih so funkcije obravnavane kot prvovrstne vrednosti, zato jih lahko podamo kot argument funkciji, so rezultat druge funkcije, jih hranimo v podatkovnih strukturah in podobno.

2.2.1. *Anonimne funkcije.* Osnovni gradniki funkcijskega programskega jezika so funkcije. Če funkcijo uporabimo enkrat oz. malokrat, jo velikokrat napišemo kot anonimno funkcijo (lambda izraz), ki je brez imena, kar pride prav pri velikem številu funkcij, ki bi potrebovale različna imena. Anonimna funkcija je oblika ugnezdene funkcije, ki pa mora biti implementirana znotraj zaprtja. Zaradi slednje lastnosti lahko anonimna funkcija dostopa do spremenljivk zunanje funkcije, v kateri je ugnezdena. Zaprtje si lahko predstavljamo kot okolje, ki ima svoje lokalne spremenljivke, do katerih lahko ugnezdene funkcije dostopajo. Ugnezdene funkcije si zapomnijo, iz katerega okolja prihajajo. Spomnimo se primera poenostavljenega lambda izraza, ki predstavlja vsoto dveh števil, $\lambda\, x\, y\, .\, x + y$. Enako bi v Elmu zapisali kot:

```
\x y -> x + y
```

Zgornji izraz je tudi okrajšava, v ozadju se zgodi enako kot v lambda izrazu:

```
\x -> (\y -> x + y)
```

Običajno pišemo poenostavljene anonimne funkcije:

```
\argumenti -> rezultat
```

2.2.2. *Funkcije višjega reda.* Funkcije višjega reda so tiste, ki sprejmejo funkcijo (lahko tudi več) kot argument ali pa vrnejo funkcijo kot rezultat.

**Primer 2.1.** Primer funkcije višjega reda je funkcija *map*, ki sprejme dva argumenta: funkcijo in seznam. Funkcijo uporabi na vsakem elementu seznama in iz novih elementov sestavi nov seznam, ki ga vrne kot rezultat. ♢

2.3. **Programski jezik s statičnimi tipi.** Razlika med programskimi jeziki s statičnimi tipi in z dinamičnimi tipi je čas preverjanja tipov.

Jeziki z dinamičnimi tipi, npr. Python ali JavaScript, tipe preverijo med izvajanjem programa; kar pomeni, da bi do napake o neskladnosti tipov prišlo šele, ko bi program med izvajanjem prišel do tovrstnega problema. Primer takšne napake, na katero bi med pisanjem lahko pozabili, bi bil primerjanje vrednosti, ki je niz, in številske vrednosti. Primer zapišimo v programskem jeziku Python:

```
a = "5"
b = 3
if a > b:
   ...
```

Programski jeziki s statičnimi tipi bi napako prepoznali že med prevajanjem programa, ker niza in celega števila ne moremo primerjati, tipi se ne ujemajo. Preverijo se tudi tipi argumentov in rezultatov funkcij, možna sta dva načina. Prvi je, da nas programski jezik prisili v pisanje tipov za dele programa (npr. funkcije ali spremenljivke) in med prevajanjem preveri, če se tipi ujemajo z napisanim. Drug način je, da se tipi izpeljejo in če jih prevajalnik ne uspe izpeljati, javi napako. Haskell in Elm sta jezika, ki spadata v slednjo skupino, omogočata pa tudi pisanje tipov delov programa, kar se izkaže za dobro prakso.

2.3.1. *Tipi.* Poznamo tipe:

- `Int` – cela števila
- `String` – niz
- `List a` – seznam elementov tipa `a`

- `Bool` – vrednosti tega tipa zavzamejo le dve alternativi: `True` ali `False`
- funkcija – funkcije so tudi same po sebi svoj tip, kako jih zapisujemo, bomo spoznali v nadaljevanju
- in drugi

Funkcija, ki sprejme argument tipa `tip1` in vrne rezultat tipa `tip2`, se s tipi piše na način:

```
funkcija1 : tip1 -> tip2
```

**Primer 2.2.** Identiteto se v Elmu piše kot:

```
identiteta : Int -> Int
identiteta x = x
```

Prva vrstica opiše tip argumenta, ki ga funkcija `identiteta` sprejme, v našem primeru so to cela števila, in tip vrnjenega rezultata, ki je prav tako celo število. V drugi vrstici pa je zapisan predpis funkcije. ♢

Princip se nadaljuje tudi pri funkcijah z več argumenti.

```
funkcija2 : tip1 -> tip2 -> tip3
```

Kar pomeni, da funkcija `funkcija2` sprejme dva argumenta (en je tipa `tip1` in drugi tipa `tip2`) in rezultat tipa `tip3`. Se pravi, najprej naštejemo tipe argumentov in potem še tip rezultata.

**Primer 2.3.** Zapišimo funkcijo, ki sešteje dve števili v Elmu:

```
vsota: Int -> Int -> Int
vsota x y = x + y
```

Enako, kot v primeru 2.2, prva vrstica opiše tip funkcije. Rezultat in argumenta, ki ju funkcija sprejme, so cela števila. ♢

Zakaj je takšno pisanje smiselno? Funkciji ni nujno podati takšnega števila argumentov, kot jih predvideva, lahko jih podamo manj. V tem primeru funkcija vrne funkcijo. Zato lahko postavimo oklepaje na način:

```
funkcija2 : tip1 -> (tip2 -> tip3)
```

Če bi podali samo prvi argument (tipa `tip1`), bi nam funkcija `funkcija2` vrnila funkcijo `tip2 -> tip3`. Pravimo, da je operator `->` desno asociativen. To pa se tudi zgodi v ozadju, funkcija vedno sprejme le en argument, če je teh navidezno več, najprej vzame prvega in vrne funkcijo, ki sprejme ostale in vrne rezultat, postopek se rekurzivno nadaljuje.

Za aplikacijo pa velja dogovor o levi asociativnosti [10]:

```
funkcija2 a b = (funkcija2 a) b
```

V razdelku 2.2 smo spoznali funkcije kot prvovrstne vrednosti in povedali, da jih lahko podamo kot argument funkciji. Spodnji primer (`funkcija3`) prikazuje, kako s tipi zapišemo argument, ki je funkcija:

```
funkcija3 : (tip1 -> tip2) -> tip3 -> tip4
```

Funkcija `funkcija3` sprejme za prvi argument funkcijo, ta sprejme argument tipa `tip1` in vrne rezultat tipa `tip2`. Drugi argument funkcije `funkcija3` je tipa `tip3`,

končni rezultat funkcije pa je iz množice `tip4`. Zaradi dogovora o desni asociativnosti so v tem primeru oklepaji obvezni.

**Primer 2.4.** Najprej definirajmo pomožno funkcijo `pomoznaFunkcija`, ki jo bomo kasneje podali kot argument funkciji `mnozenje`:

```
pomoznaFunkcija : Int -> Int
pomoznaFunkcija x = x + 3

mnozenje: (Int -> Int) -> Int -> Int
mnozenje funkcija x = (funkcija x) * 5
```

Sedaj pa izvedimo klic funkcije `mnozenje`.

```
mnozenje pomoznaFunkcija 2 = 25
```

Funkcija `mnozenje` najprej število 2 poveča za 3 (`pomoznaFunkcija`), rezultat pa pomnoži s 5. ♢

2.4. **Čiste funkcije.** Vrednosti v FP se ne spreminjajo, nimajo stanja, ampak se pri transformaciji naredijo čisto nove vrednosti. Recimo, da želimo dodati element v seznam. Pri ukaznih programskih jezikih bi se star seznam podaljšal za element, ki ga želimo dodati. Pri funkcijskem programiranju stari seznam ostane nedotaknjen, naredi se povsem nov seznam, ki vsebuje vse stare elemente in tega, ki smo ga dodali. Če dodamo še lastnost, da so funkcije brez stranskih učinkov (npr. izpisi), ki bi vplivali na končni rezultat funkcij, dobimo čiste funkcije. Kar privede do tega, da funkcije v FP vedno vrnejo isto vrednost pri istih argumentih, v nasprotju z ukaznimi jeziki, kjer lahko vedno dobimo drugačen rezultat. Tako so te funkcije bolj matematične, $\sin \pi$ bo vedno enak 0, ne glede na okoliščine.

2.5. **Rekurzija.** Ker vrednosti v FP nimajo stanja, zank ne moremo izvajati (npr. števca ne moremo povečevati za 1), zato si pomagamo z rekurzijo. Rekurzija je klic funkcije same sebe. Rekurzivni klici lahko hitro porabijo ves pomnilnik, saj si mora računalnik pri vsakem rekurzivnem klicu zapomniti, od kje je bila funkcija klicana, pomnilnik pa zapolnjujejo tudi nove vrednosti. Pri optimizaciji je učinkovit t. i. *klic z repa*. Cilj optimizacije je, da se ob klicu funkcije ne porablja dodaten pomnilnik. Da se stare vrednosti, ki jih ne uporabljamo več, brišejo, skrbi *avtomatsko čiščenje spomina*.

2.5.1. *Klic z repa.* V primeru 2.5 je prikazano, kako potekajo rekurzivni klici.

**Primer 2.5.** Recimo, da želimo ustvariti seznam dolžine $n$, elementi pa so števila 2. V ta namen definiramo funkcijo `funkcijaA`:

```
funkcijaA : Int -> List Int
funkcijaA n =
  case n of
    0 -> []
    n -> 2 :: funkcijaA (n-1)
```

V funkciji `funkcijaA` je uporabljen stavek `case`, ki omogoča obravnavanje primerov števila $n$ in s tem določitev zaustavitvenega pogoja. Stavek `case` bomo podrobneje spoznali v razdelku 5.2. Izraz `2 :: seznam` doda število 2 na začetek seznama `seznam`, seveda, se pri tem ustvari nova vrednost.

Uporabimo našo funkcijo na primeru, ko je $n = 3$:

```
funkcijaA 3 =
  = 2 :: funkcija 2
  = 2 :: 2 :: funkcija 1
  = 2 :: 2 :: 2 :: funkcija 0
  = 2 :: 2 :: 2 :: []
  = 2 :: 2 :: [2]
  = 2 :: [2, 2]
  = [2, 2, 2]
```

V tem primeru vidimo, da za takšno vrsto rekurzije potrebujemo kar nekaj pomnilnika, saj si mora računalnik zapomniti, od kje je bila funkcija klicana in kaj mora z rezultatom narediti, v našem primeru je to dodajanje števila 2.

Enak rezultat lahko dosežemo še na drugačen način:

```
funkcijaB : List Int -> Int -> List Int
funkcijaB sez n =
  case n of
  0 -> sez
  n -> funkcijaB (2 :: sez) (n-1)
```

Poglejmo še to funkcijo na primeru, ko je $n = 3$:

```
funkcijaB [] 3 =
  = funkcijaB [2] 2
  = funkcijaB [2, 2] 1
  = funkcijaB [2, 2, 2] 0
  = [2, 2, 2]
```

Kar je bistvena razlika med funkcijo `funkcijaA` in funkcijo `funkcijaB` je, da si računalniku pri drugi funkciji ni potrebno zapomniti, od kje je bila klicana, samo vrne končni rezultat. ♢

Optimizaciji, da je programski jezik sposoben prepoznati tip rekurzije, ki kliče zgolj samo sebe in pri tem ne izvaja drugih operacij, ki bi povzročile, da bi si računalnik moral zapomniti, od kje je bila funkcija klicana, pravimo rekurzivni klic z repa. Klic z repa je v splošnem optimizacija, ki povzroči, da računalnik ne potrebuje dodatnega pomnilnika ob klicu druge funkcije ali metode, ker vrne njen rezultat.

2.5.2. *Čiščenje spomina.* Poznamo jezike (npr. `C++`), ki od programerja zahtevajo, da upravlja s pomnilnikom, se pravi pove, koliko prostora bodo zasedli določeni deli programa. Izkaže se, da je to precej zahtevno delo, ki ga človek največkrat ne opravi najbolje, zato je boljše, če se pomnilnik upravlja avtomatsko. To pomeni, da se med izvajanjem programa brišejo vrednosti iz spomina, ki jih program ne potrebuje več. Implementacija čiščenja spomina je precej zahtevna ter odvisna od programskega jezika in ni univerzalnega algoritma, ki bi deloval pri vseh jezikih. Ker vrednosti v FP nimajo stanja in se ustvarijo nove vrednosti, je čiščenje spomina nekoliko olajšano.

## 3. Funkcijsko reagentno programiranje

Razdelek je napisan na podlagi [3, 6, 7, 15, 16].

Če funkcijskemu programiranju dodamo še vrednosti, ki se spreminjajo s časom, dobimo *funkcijsko reagentno programiranje* (FRP), ki je predvsem uporabljeno pri

programiranju uporabniških vmesnikov, igric, v robotiki, glasbi, skratka, kjer je veliko komuniciranja z okolico. Elm je primer programskega jezika, ki podpira FRP.

Začetnika funkcijskega reagentnega programiranja sta Paul Hudak in Conal Elliott. Leta 1997 sta izdala članek z naslovom "Funkcijsko reagentna animacija". Že naslov pove, da je šlo idejno za animacije, te pa so še posebno zanimive, če se ustvarjajo s spreminjajočimi se dogodki v odvisnosti od časa. Paul Hudak je bil ameriški profesor računalništva na prestižni Yale University, najbolj pa je poznan kot soustvarjalec programskega jezika Haskell. Conal Elliott se večino svojega življenja ukvarja z računalniško grafiko. Njegova ciljna publika so grafični oblikovalci, glasbeniki in otroci. Kar nekaj let je bil razvijalec v Microsoftu. V svojem delu "Funkcijsko reagentna animacija" sta vrednosti v jeziku razdelila na dva dela: vrednosti, ki se stalno spreminjajo s časom, in vrednosti, ki se dogajajo le ob diskretnih časih (dogodki). Paradigmo sta vgradila v programski jezik Haskell, ki pa je zaradi svoje lenobe povzročal časovne in prostorske probleme. Da je programski jezik len, pomeni, da svoje naloge prelaga, dokler jih ni nujno potrebno izvesti. Python je primer neučakanega programskega jezika.

Cilj FRP v realnem času je bil odpraviti časovni in prostorski problem. Nihče si ne želi uporabniškega vmesnika, ki bi v dnevu nekaj izrisal, ali pa robota, ki bi se prepočasi odzival na zunanje dejavnike. Vse, kar bi program izvedel, bi se zgodilo v omejenem času in porabilo omejen prostor. V ta namen je bil razvit programski jezik iz dveh delov; bazni jezik je ostal neomejen, omejeni del jezika je bil namenjen prejemanju signalov iz okolice. Signal je tip vrednosti, s katerim lahko opišemo dogodke in vrednosti, ki se stalno spreminjajo s časom.

Dogodkovno FRP se od FRP v realnem času razlikuje le po tem, da vsebuje diskretne signale, katerih vrednosti se spremenijo le ob dogodkih. Paradigma je bila primarno razvita za namene robotike.

Kmalu za dogodkovnim FRP se je razvilo FRP s puščicami, ki namesto dogodkov uporablja signalne funkcije. Signalne funkcije si lahko predstavljamo kot funkcije, ki signal preslikajo v signal. Tako programer preko signalnih funkcij dostopa do signalov in ne direktno, kar je rešilo časovni in prostorski problem. Največje pomanjkljivosti te paradigme so globalne zamude in nepotrebne posodobitve. Globalne zamude so posledica dolgega računanja ob dogodkih. Nepotrebne posodobitve so tiste, ki se izvedejo, čeprav se vhodni podatki niso spremenili.

Cilj FRP s pristopom potisni in povleci je bil izpopolniti jezik, ki bo kar se da učinkovit. Nekateri dogodki, čeprav so deklarirani kot vrednosti, ki se stalno spreminjajo, se ne spreminjajo stalno (npr. miške ne premikamo), zato ni potrebno, da se stalno posodabljajo, ampak le, ko je to potrebno.

## 4. Elm

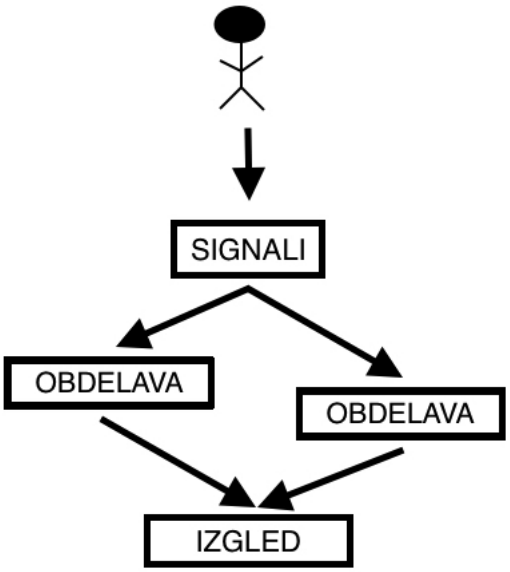
V razdelku sta uporabljena vira [3, 24].

Elm je precej nov programski jezik, leta 2012 ga je v Haskellu implementiral Evan Czaplicki, namenjen je funkcijsko (reagentnemu) programiranju in s tem ustvarjanju uporabniških vmesnikov. Končen, preveden program je v označevalnem jeziku HTML, slogovnem jeziku CSS in programskem jeziku JavaScript. HTML se uporablja za izdelovanje spletnih strani, stilsko jih oblikujemo s CSS, logika pa se doda s programskim jezikom JavaScript, ki ni namenjen funkcijskemu programiranju. Težave nastanejo, ko pri večjih projektih želimo kaj spremeniti, dodati.

Tovrstnih težav je pri funkcijskem programiranju bistveno manj oz. jih lažje odpravimo, po drugi strani se veliko hitreje naučimo programskih jezikov prvega tipa. Cilj Evana Czaplickija je bil ustvariti jezik, ki bi se ga hitro naučili, ob dodajanju oz. spreminjanju programa pa bi imeli malo težav, ki bi bile hitro odpravljive. Ciljna tarča Elma so bili programerji programskega jezika JavaScript. Czaplicki si je zastavil vprašanje, kako programerje, ki ne uporabljajo funkcijskega programiranja, prepričati, da bodo uporabljali Elm. V ta namen ima Elm tudi zelo natančno strukturo in nam ne pusti, da bi se program prevedel, če ne napišemo, kaj naj se zgodi ob vseh možnih vhodnih podatkih, se tipi ne ujemajo in v podobnih primerih. Ko napake polovi, nam poda kar nekaj predlogov, kaj smo lahko storili narobe.

4.1. **FRP prvega reda.** Czaplicki v enem izmed svojih predavanj [20] uvrsti Elm v t. i. *FRP prvega reda*. Značilnosti tovrstnega programiranja so:

- signali so povezani z okoljem; Primer signala, ki je povezan z okoljem, je dotik na zaslon na dotik (signal dotika je neposredno povezan s programskim jezikom).
- signali so neskončni; Končni signal bi bil, če bi lahko miško med izvajanjem izklopili iz računalnika. Vsi vgrajeni signali, ki se uporabijo v aplikaciji, se opazujejo med celotnim izvajanjem programa.
- signalni grafi so statični; Signalni graf (slika 1) nam prikazuje obdelavo signalov. Imamo nekaj vhodnih signalov, iz katerih pridobimo novo vrednost aplikacije, ta pa se na koncu obdela na način, ki ga je mogoče prikazati na zaslonu.
- program je privzeto sinhroniziran; Med obdelavo signalov moramo biti pozorni, da bo končni rezultat pravilen, se dogodki ne bodo pomešali med samo (npr. če tipkamo besedilo, se črke na zaslon izpisujejo v enakem vrstnem redu, kot so bile pritisnjene). V Elmu je slednja lastnost privzeta.




Slika 1. Signalni graf opiše delovanje programske kode. Uporabnik z dogodki, kot je klik na miško, vpliva na vrednosti signalov, ki se potem obdelajo, na koncu se vrednost aplikacije obdela na način, ki ga je mogoče prikazati na zaslonu.

Iz omenjenih lastnosti FRP prvega reda je Elm učinkovit jezik, programi so v naprej določeni obliki, mogoče je tudi potovanje v času.

4.2. **Optimizacija.** Če bi želeli Elm uvrstivi v eno izmed vrst FRP, ki so navedene v razdelku 3, bi ga uvrstili v dogodkovno FRP, saj dogodki iz okolja povzročijo spreminjanje programa. Pri tem je potrebno omeniti, da se prikaz na zaslonu spremeni le, če se je res kaj spremenilo. Slednje pomeni, da tudi, če iz okolja prihajajo dogodki, ki sicer vplivajo na program, ampak ne na izgled, se aplikacija ne riše znova in znova.

Kot omenjeno na začetku razdelka 4, je prevedeni program pravzaprav JavaScript. Od tod sledi, da je čiščenje spomina avtomatsko [30], implementiran je tudi klic z repa [11].

4.3. **Razhroščevalnik.** Elm zaradi čistih funkcij (funkcije ob enakih argumentih vedno vrnejo enak rezultat) omogoča potovanje v času; kar pomeni, da čas lahko zavrtimo nazaj, v preteklosti nekaj spremenimo, čas zavrtimo naprej, kjer so takoj vidne spremembe iz spremenjene preteklosti, ne da bi program še enkrat prevedli in vnesli točno enake vhodne podatke kot prvič. *Time traveling debugger* je orodje, ki nam omogoča potovanje v času in pomaga popraviti semantične napake [23].

## 5. Sintaksa Elma

V tem razdelku bomo spoznali osnovne sintaktične lastnosti programskega jezika Elm [21].

5.1. **Pogojni izraz.** Pogojni izraz se sintaktično ne razlikuje prav veliko od tistega v Pythonu, zapisuje se na sledeči način:

```
if <pogoj1> then
  ...
else if <pogoj2> then
  ...
else
  ...
```

5.2. **Obravnavanje primerov.** Recimo, da bi neka funkcija sprejela argument, ki bi bil tipa `Bool`. Radi bi ločili primera, ko je argument enak `True` oziroma `False`. V Elmu se različni primeri obravnavajo s pomočjo izraza `case`:

```
case <vrednost> of
  <primer1> ->
    ...
  <primer2> ->
    ...
```

5.3. **Lokalne definicije.** Zapis težko berljive funkcije si lahko olajšamo z definicijami lokalnih vrednosti, ki jih smemo definirati samo enkrat znotraj istega `let` izraza. Se pravi, če vrednosti `a` dodelimo število 5, ji v nadaljevanju ne smemo dodeliti katerekoli druge vrednosti. Sintaktično lokalne vrednosti zapišemo kot:

```
let
  <vrednost1> = ...
  <vrednost2> = ...
  ...
in
  <izraz>
```

Dodati je potrebno, da so lahko lokalne vrednosti tudi funkcije, ki jih zapišemo enako, kot bi jih sicer, dodamo lahko tudi tipe.

5.4. **Nov tip vrednosti.** Nov tip vrednosti lahko definiramo kot unijo disjunktnih alternativ in tako opredelimo vse mogoče alternative, ki jih lahko zavzame nov tip:

```
type NovTip =
  Alternativa1
  | Alternativa2
  ...
```

V Elmu se vsi tipi pišejo z veliko začetnico.

5.5. **Tip Maybe a.** Če ne vemo, ali vrednost obstaja, uporabimo tip `Maybe a`, ki ima alternativi `Just a` in `Nothing` [29]:

```
type Maybe a =
  Just a
  | Nothing
```

Recimo, da v celoštevilskem seznamu iščemo prvo liho število; če funkcija najde liho število, ga vrne kot rezultat oblike `Just Int`, sicer vrne `Nothing`. Ker ne vemo, ali poljuben seznam vsebuje liho število, je rezultat tipa `Maybe a`, bolj natančno `Maybe Int`.

```
prvoLiho : List Int -> Maybe Int
prvoLiho seznam =
  case seznam of
    [] ->
      Nothing
    prvoStevilo :: ostanekSeznama ->
      if prvoStevilo % 2 == 1 then
        Just prvoStevilo
      else
        prvoLiho ostanekSeznama
```

Funkcija `prvoLiho` sprejme za argument seznam celih števil in najprej loči primera. Če je seznam prazen, takrat vrne `Nothing`, sicer ima seznam vsaj en element, `prvoStevilo`, in `ostanekSeznama`. Vsak seznam lahko namreč zapišemo kot:

```
element1 :: ··· :: elementN :: []
```

Kar je ekvivalentno zapisu:

```
[element1,..., elementN]
```

Potem preveri, če je `prvoStevilo` liho (ostanek pri deljenju z dva je enak ena) in če je, vrne to število, sicer nadaljuje na preostanku seznama.

5.6. **Zapis.** Opišimo, kaj je zapis v Elmu. Zapis je tip vrednosti, ki se zapiše znotraj zavitih oklepajev, sestavljajo pa ga polja:

```
{ polje1 : tipPolja1, polje2 : tipPolja2,... }
```

Se pravi, polja podamo drugega za drugim, vrednosti polj pa so nekega tipa. Recimo, da bi radi z zapisom opisali hišo, koliko ima oken, nadstropij in ali ima klet (da ali ne):

```
hisa : { okna : Int, nadstropja : Int, klet : Bool }
hisa =
  { okna = 10, nadstropja = 3, klet = True }
```

Konstantna funkcija `hisa` je tipa zapis, ki vsebuje polja: `okna, nadstropja, klet`. Tip vrednosti je podan v prvi vrstici, v drugi oz. tretji pa smo poljem dodali eksaktne vrednosti. Do polj dostopamo s pomočjo pike, npr. klic `hisa.okna` vrne vrednost `10`.

Hiša je postala premajhna, zato so se lastniki odločili za povečanje le-te, dodali so ji eno nadstropje in tri okna:

```
novaHisa : { okna : Int, nadstropja : Int, klet : Bool }
novaHisa =
  { hisa | okna = hisa.okna + 3
         , nadstropja = hisa.nadstropja + 1 }
```

Če gledamo funkcijo `novaHisa` s stališča tipov, se ni spremenila od funkcije `hisa`, saj še vedno gledamo iste podatke o hiši. Lahko bi podatke o novi hiši vnesli na enak način, kot smo to storili prvič, ampak smo se odločili, da to storimo drugače. Druga in tretja vrstica pravita, da je `novaHisa` pravzaprav `hisa`, kateri smo dodali (prišteli) tri okna in eno nadstropje. Podatek o kleti ni naveden, zato je vrednost polja `klet` enaka `hisa.klet`. Pri tem je potrebno opozoriti, da se funkcija `hisa` ni spremenila, ustvarila se je nova vrednost.

Opazili smo že, da sta funkciji `hisa` in `novaHisa` enakega tipa. Tip zapisa lahko tudi shranimo pod novo ime, naj bo to `OpisHise`:

```
type alias OpisHise =
  { okna : Int, nadstropja : Int, klet : Bool }
```

S `type alias` poimenujemo nekaj že znanega, ne definiramo novega tipa vrednosti, tudi ni nujno, da zapisu dodelimo novo ime, lahko je to tudi funkcija ali kateri drug tip.

5.7. **Poenostavitev programske kode.** Elm omogoča kar nekaj možnosti za poenostavitev programske kode, ki potem postane lažje berljiva. Spoznali bomo operator `|>`. Vrednost, ki stoji na levi strani operatorja, postane zadnji argument funkcije, ki stoji na desni strani operatorja:

```
argument |> funkcija → funkcija argument
```

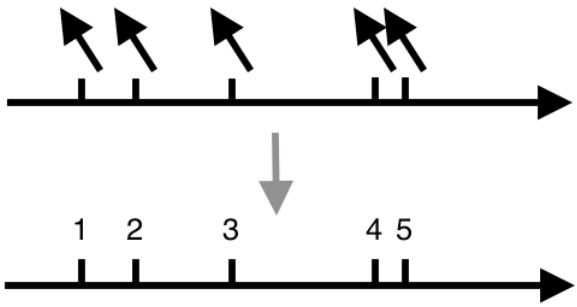
## 6. Signali

Glavni vir razdelka je Elmova knjižnica [33], dopolnjena z [20].

Signali so velik del FRP, so vrednosti, ki se diskretno spreminjajo s časom in naredijo funkcijsko programiranje reagentno, kar pomeni, da aplikacija reagira na zunanje dejavnike. Najlažje si jih predstavljamo kot neskončne časovne vrste, ki

predstavljajo zaporedja dogodkov v času. Vrednost signala se spremeni ob dogodku (npr. klik na miško). Ne moremo pa vedeti, kaj se bo zgodilo s signalom v prihodnosti oz. ne moremo predvideti, kaj bo uporabnik storil.

**Primer 6.1.** Želimo beležiti kolikokrat smo kliknili na miško. Ustvarimo signal, ki se poveča za ena ob vsakem kliku, kot prikazuje slika 2.




Slika 2. Vrednost novega signala dobimo s povečanjem pretekle vrednosti za ena ob vsakem kliku na miško.

♢

Formalno je signal tip vrednosti, vrednosti signala pa morajo biti določenega tipa:

```
type Signal a
```

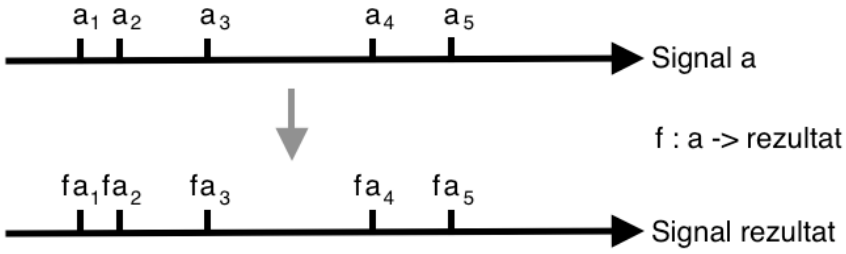
Primeri signalov:

- `Window.dimensions : Signal (Int, Int)` – velikost zaslona (širina, višina)
- `Mouse.position : Signal (Int, Int)` – koordinati miške
- `Mouse.isDown : Signal Bool` – zavzame vrednost `True`, če uporabnik pritisne na miško, in `False`, če miška ni pritisnjena

**Opomba 6.2.** `Window` in `Mouse` sta paketa, ki vsebujeta funkcije `dimensions` oz. `position`, `isDown`.

Opisali smo, kaj so signali, kako si jih lahko predstavljamo s časovnimi vrstami, sedaj pa bi radi z njimi manipulirali.

6.1. **Funkcija map.** Funkcija `map` člene časovne vrste s pomočjo funkcije preslika v novo časovno vrsto, kot prikazuje slika 3.




Slika 3. Funkcija `map` vrednosti starega signala (`a`$_i$) priredi novo vrednost tako, da izračuna vrednost funkcije `f` z argumentom `a`$_i$.

**Primer 6.3.** Predstavljajmo si neskončno zaporedje celih števil. Na vsakem številu bi radi uporabili funkcijo, ki bi vsakemu številu prištela število 3. Tako bi dobili novo zaporedje, kjer bi bila vsa števila za 3 večja od istoležnih števil prvotnega zaporedja. V Elmu bi to izgledalo takole:

```
map (\a ->  a + 3) nasSignal
```

Kjer `nasSignal` predstavlja prvotno zaporedje, anonimna funkcija pa vsako celo število poveča za 3.

Prvotno in novo zaporedje sta v jeziku signalov signala s celoštevilskimi vrednostmi (`Signal Int`). ♢

Vrednosti vsakega signala so enega tipa, v primeru 6.3 je signal tipa `Signal Int`, kar pomeni, da ima signal celoštevilske vrednosti. S tem, ko smo mu prišteli neko število, se tip signala ni spremenil, lahko pa bi se (npr. vsaki številki bi priredili barvo in bi tako dobili signal barv), kar prikazuje tudi spodnja shema:

```
map : (a -> rezultat) -> Signal a -> Signal rezultat
```

Funkcija `map2` je tako po imenu kot tudi uporabi podobna funkciji `map`. `Map2` sprejme tri (ne dva) argumenta: funkcijo, ki poveže vrednosti dveh signalov v novo vrednost in dva signala. Oglejmo si shemo:

```
map2 :
  (a -> b -> rezultat)
  -> Signal a
  -> Signal b
  -> Signal rezultat
```

Opazimo, da sta signala lahko tudi različnega tipa, le da ju znamo s funkcijo dveh spremenljivk preslikati v rezultat.

Funkcije `map3, map4` in `map5` posplošijo funkcijo `map`; namesto enega sprejmejo tri, štiri oz. pet signalov in funkcijo, ki vrednosti vseh podanih signalov preslika v vrednost novega signala.

6.2. **Funkcija merge.** Če želimo signala združiti po principu (pokvarjene) zadrge, uporabimo funkcijo `merge`. Za združitev več signalov uporabimo funkcijo `mergeMany`.

**Primer 6.4.** Če bi združili signal levega klika miške in signal desnega klika, lahko dobimo npr. zaporedje: levo, levo, desno, levo, desno, desno,... Se pravi, ni nujno, da se spremeni vrednost enega signala in potem nujno vrednost drugega. ♢

Za razliko od funkcije `map2`, `merge` sprejme dva argumenta, ki sta signala istega tipa, da lahko iz njiju, brez dodatne funkcije (transformacije), ustvari nov signal, ki je istega tipa kot prvotna signala:

```
merge : Signal a -> Signal a -> Signal a
```

Funkcija `mergeMany` posploši osnovno funkcijo, le da za argument prejme seznam signalov, ki so enakega tipa:

```
mergeMany : List (Signal a) -> Signal a
```

6.3. **Funkcija foldp.** Obstaja tudi možnost, da beležimo zgodovino. V tem primeru bi uporabili funkcijo `foldp`:

```
foldp :
  (a -> vrednost -> vrednost)
  -> vrednost
  -> Signal a
  -> Signal vrednost
```

Iz sheme lahko ugotovimo, da funkcija `foldp` sprejme tri argumente:

- funkcijo, ki iz vrednosti starega signala in trenutne vrednosti novega signala vrne novo vrednost novega signala,
- začetno vrednost novega signala in
- signal.

S funkcijo `foldp` bi lahko implementirali motivacijski primer iz začetka razdelka, kjer želimo ustvariti signal, ki bi beležil število klikov na miško:

```
belezenjeKlikov : Signal Int
belezenjeKlikov =
  foldp (\_ stevilo -> stevilo + 1) 0 Mouse.clicks
```

Na začetku je vrednost novega signala enaka nič, ob vsakem kliku pa se stara vrednost signala poveča za ena. Ker nova vrednost ni odvisna od klika, je klik v anonimni funkciji zapisan z anonimnim argumentom; v zgornji anonimni funkciji ga vidimo kot podčrtaj in si nam ni potrebno izmišljevati imena za argument, ki ga v izrazu ne potrebujemo.

6.4. **Funkcija filter.** Funkcija `filter` omogoča, da iz signala preberemo le določene vrednosti, filtriramo signal:

```
filter : (a -> Bool) -> a -> Signal a -> Signal a
```

Funkcija `filter` sprejme tri argumente: signal, začetno vrednost novega (filtriranega) signala in funkcijo, ki iz vrednosti signala vrne `True` ali `False`, se pravi, se odloči, katere vrednosti naj obdrži.

Poznamo še tri posebna filtriranja; `filterMap`, `dropRepeats` in `sampleOn`.

```
filterMap : (a -> Maybe b) -> b -> Signal a -> Signal b
```

Argumenti funkcije `filterMap` so: funkcija ki vrne `Maybe b`, začetna vrednost novega signala in signal. Argumenti so podobni kot pri funkciji `filter`, le da se obdržijo vrednosti, pri katerih funkcija, ki jo podamo kot prvi argument, vrne `Just b` in ne `True`, kot pri funkciji `filter`.

```
dropRepeats : Signal a -> Signal a
```

Z `dropRepeats` iz prvotnega signala izpustimo ponavljajoče se vrednosti. Recimo, da imamo signal, ki beleži zadnji klik na miško; ali je bil klik levi ali desni. Če uporabnik dvakrat zaporedoma klikne levi gumb miške, drugi klik ne povzroči spremembe vrednosti novega signala.

Funkcija `sampleOn` pa ustvari nov signal iz vrednosti starega signala, ko se zgodi neki dogodek neodvisno od izvornega signala:

```
sampleOn : Signal a -> Signal b -> Signal b
```

**Primer 6.5.** Želimo ustvariti signal, ki bo beležil, kje je bila miška kliknjena:

```
sampleOn Mouse.clicks Mouse.position
```

Vrednost vrnjenega signala se spremeni ob vsakem kliku na miško, priredi se mu vrednost signala `Mouse.position`. ♢

6.5. **Funkcija constant.** Vse do sedaj omenjene funkcije na signalih kot argument sprejmejo že obstoječ signal. Funkcija `constant` omogoča, da ustvarimo signal, ki ima konstantno vrednost:

```
constant : a -> Signal a
```

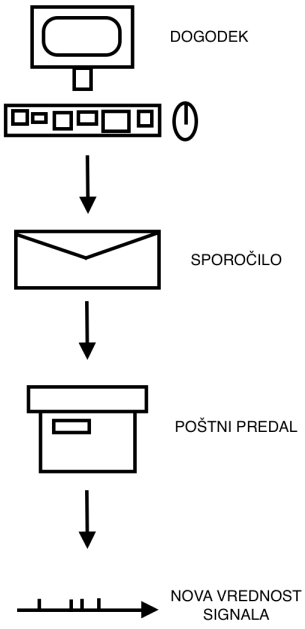
6.6. **Poštni predal.** Recimo, da želimo spremeniti vrednost signala ob določenem dogodku, npr. ob pritisku na gumb ali vnosu besedila v polje. Da bomo lahko dogodek povezali s signalom, moramo signalu dodeliti naslov, preko katerega dostopamo do njega. Slednje omogoča tip vrednosti `Mailbox a` [8]:

```
type alias Mailbox a :
  { address : Address a
  , signal : Signal a
  }
```

Vrednost tipa `Mailbox` ustvarimo s funkcijo `mailbox`, katere argument je začetna vrednost signala, ki pripada ustvarjenemu poštnemu predalu:

```
mailbox : a -> Mailbox a
```

Slika 4 prikazuje idejo poštnih predalov; ob dogodku se ustvari sporočilo, ki se preko naslova pošlje v poštni predal, kjer je signal, katerega vrednost se ustrezno spremeni.




Slika 4. Poštni predal omogoča spreminjanje pripadajočega signala, do katerega dostopamo preko naslova.

Videli smo, da se da s signali marsikaj početi, izločiti pa želimo primere signale signalov, saj bi s tem izgubili lastnost statičnega signalnega grafa. Signal signalov si lahko predstavljamo, kot da bi preklapljali med signali. Implementacija tega

ni nemogoča, je pa potrebno postaviti nekaj pravil. Podrobnosti so dostopne v predavanju Evana Czaplickija [20].

## 7. Oblika programa

Podlaga za razdelek sta [17, 22].

Sedaj si bomo ogledali praktični primer, ki bo napisan v programskem jeziku Elm. Naš cilj bo ustvariti spletno aplikacijo, ki bo sposobna rekurzivnega risanja, kar pomeni, da bomo lahko svoje slike uporabljali kot osnovna lika (kvadrat in krog). Program bomo poenostavili samo na risanje krogov, kvadrate bomo izpustili. Celotna programska koda je napisana v razdelku 9.

Do funkcij in tipov znotraj paketa dostopamo s pomočjo pike:

```
imePaketa.funkcija/tip
```

Dogovorili se bomo, da bomo ime paketa izpuščali, kjer je jasno, iz katerega paketa prihaja funkcija ali tip.

### 7.1. **Statična vsebina.**

Najprej bomo izrisali krog na zaslon. HTML elementi, kot so `div, h1, svg,...`, so sintaktično zelo podobni v Elmu.

**Primer 7.1.** Za primer vzemimo značko `<div>`.

V znakovnem jeziku HTML bi jo zapisali kot:

```
<div atributi> vsebina </div>
```

Elm pa bi enako zapisal kot:

```
div [atributi] [vsebina]
```

Znački lahko med atribute podamo razred, preko katerega dostopamo do elementov istega razreda, jih na ta način stilsko oblikujemo, lahko podamo tudi unikatno ime, dogodke (npr. klik na element, vnos besedila v polje in podobno), velikost elementa in drugo. Kako se atributi zapisujejo v Elmovi sintaksi, bomo videli v nadaljevanju razdelka. ♢

Vrnimo se na naš praktični primer. Napisali bomo funkcijo `main`, ki vrne končni rezultat aplikacije, zato mora biti tipa, ki ga lahko prikažemo na zaslonu, v našem primeru bo tipa `Html`.

```
main : Html
main =
  div
    []
    [ fromElement (collage 500 500
                    [ circle 100 |> filled yellow
                                 |> move (10, 0) ])
    ]
```

Opišimo zgornjo kodo: Ustvarjen je krog (`circle`) s polmerom velikosti `100`, rumene barve in premaknjen za vektor `(10, 0)`. Krog je postavljen v območje (`collage`) velikosti `500×500`. Ker je `collage` tipa `Element`, ga s pomočjo funkcije `fromElement` spremenimo v HTML, kar so zahteve funkcije `div`. Sedaj smo ustvarili statično spletno stran, ki ni odzivna na zunanje dejavnike.

7.2. **Dinamična vsebina.** Dinamiko bomo najprej dodali risalni površini. Cilj je ustvariti risalno površino tako veliko, kot je zaslon, ki se bo ob spremembi velikosti zaslona tudi pomanjšala oz. povečala. Na tem mestu se bomo spomnili signalov, vrednosti, ki se spreminjajo s časom. Dodali bomo signal `Window.dimensions`, ki je tipa `Signal (Int, Int)`. S funkcijo `map` si bomo pomagali, da bomo prišli do cilja. Spomnimo se, da funkcija `map` ustvari nov signal, ki vrednosti starega signala priredi novo vrednost.

```
main : Signal Html
main =
  map
    (\(w, h) ->
      div
        []
        [ fromElement (collage w h
                        [ circle 100 |> filled yellow
                                     |> move (10, 0) ])
        ]
    )
    Window.dimensions
```

Opazimo, da velikost risalnega območja ni več točno `500×500`, ampak sta širina in višina vrednosti signala `Window.dimensions`. Spremenil se je tudi tip funkcije `main`, iz tipa `Html` v signal z vrednostmi tipa `Html`.

Dodajmo še signal `Mouse.position`, katerega vrednosti sta koordinati miške. Ker bomo tokrat povezali dva signala v novega, bomo uporabili funkcijo `map2`.

```
main : Signal Html
main =
  map2
    (\(w, h) (x, y) ->
      div
        []
        [ fromElement (collage w h
                        [ circle 100 |> filled yellow
                                     |> move (toFloat x
                                              , toFloat y)
                        ])
        ]
    )
    Window.dimensions Mouse.position
```

Sedaj smo dobili krog, ki se premika, kot se miška oz. skoraj tako, v `x` smeri se premika pravilno (kot si želimo), v `y` smeri pa ravno obratno (ko gremo z miško navzgor po ekranu se krog pomika navzdol in obratno). Razlog tiči v koordinatnem sistemu: koordinatno izhodišče pozicije miške je v levem zgornjem kotu zaslona. Koordinata `y` narašča, ko se z miško pomikamo navzdol. Naredimo nov signal, ki bo koordinatno izhodišče postavil na sredino zaslona. V istem koraku bomo koordinati miške spremenili v decimalni števili, kar smo zgoraj storili s pomočjo funkcije `toFloat`.

```
signalMiska : Signal (Float, Float)
signalMiska =
  map2
    (\(w, h) (x, y) ->
      (toFloat x - toFloat w/2 , toFloat h/2 - toFloat y)
    )
    Window.dimensions Mouse.position
```

Če sedaj zamenjamo signal `Mouse.position` s signalom `signalMiska`:

```
main : Signal Html
main =
  map2
    (\(w, h) (x , y) ->
      div
        []
        [ fromElement
          (collage w h [ circle 100 |> filled yellow
                                    |> move (x, y) ])
        ]
    )
    Window.dimensions signalMiska
```

Takšna spletna aplikacija premika krog skupaj z uporabnikovo miško.

Dodajmo, če je miška pritisnjena, se krog premika, sicer je na mestu.

Funkcija `div` kot prvi argument sprejme seznam atributov. Sem sodijo tudi funkcije, ki se izvedejo ob določenem dogodku in vrnejo atribut. Primeri:

- `onClick : Address a -> a -> Attribute` – funkcija se izvede ob kliku na miško
- `onMouseDown : Address a -> a -> Attribute` – funkcija se izvede, ko uporabnik pritiska na miško
- `onMouseUp : Address a -> a -> Attribute` – funkcija se izvede, ko miška ni pritisnjena

Opazimo, da vse omenjene funkcije sprejmejo prvi argument tipa `Address a`. V podrazdelku 6.6 smo opisali tip vrednosti `Mailbox a`, ki ga sestavljata polji `address` in `signal`. Tukaj praktično vidimo, zakaj potrebujemo poštne predale (`Mailbox a`) in njihove naslove (`Address a`). Zaenkrat v našem programu še nimamo signala, do katerega bi lahko dostopali preko naslova, zato ga moramo ustvariti s pomočjo funkcije `mailbox`:

```
postniPredal : Mailbox Miska
postniPredal =
  mailbox MiskaGor
```

Ustvarili smo poštni predal, vrednosti pripadajočega signala so tipa `Miska`. Na začetku ima signal vrednost `MiskaGor`. `Miska` ni eden od znanih tipov, zato ga moramo na novo definirati:

```
type Miska =
  MiskaGor
  | MiskaDol
```

Tako definiran tip ima dve alternativi: `MiskaGor` in `MiskaDol`.

Imamo poštni predal in nov tip. Sedaj nastavimo vrednost signala iz poštnega predala: ko je miška pritisnjena navzdol, je vrednost signala enaka `MiskaDol`, ko pa ni pritisnjena, naj ima signal vrednost `MiskaGor`.

**Opomba 7.2.** Enak učinek bi pridobili z že vgrajenim signalom `Mouse.isDown`, ki ima vrednost `True`, če je miška pritisnjena, sicer pa `False`. Prednost našega pristopa je, da če želimo dodajati nove elemente spletni aplikaciji (npr. `div`), na ta način lahko z dodajanjem novih vrednosti tipu `Miska` ločimo, kje je bila miška pritisnjena, brez preračunavanja iz koordinat miške.

```
main : Signal Html
main =
  map2
    (\(w, h) (x , y) ->
      div
        [ onMouseDown postniPredal.address MiskaDol
        , onMouseUp postniPredal.address MiskaGor
        ]
        [ fromElement (collage w h
                        [ circle 100 |> filled yellow
                                     |> move (x, y) ])
        ]
    )
    Window.dimensions signalMiska
```

Signal poštnega predala se sicer spreminja, naš krog pa se še vedno premika tako, kot se miška. Dodajmo še, da se bo krog premikal le, ko bo miška pritisnjena nanj. Drugače povedano: krog se premika ali pa je na mestu in ima dve mogoči alternativi. Alternative nas spomnijo na definiranje tipov, kot smo to naredili pri alternativah miške: `MiskaGor` in `MiskaDol`.

```
type StanjeKroga =
  KrogPremikaj
  | KrogMiruj
```

Premislimo, kakšen mora biti signal, ki bo beležil, ali se krog premika ali ne, se pravi, bodo njegovi elementi tipa `StanjeKroga`. Imamo 6 možnosti:

| `Miska` \ `StanjeKroga` | `KrogPremikaj` | `KrogMiruj` |
|---|---|---|
| `MiskaGor` | `KrogMiruj` (konec) | `KrogMiruj` |
| `MiskaDol`, miška znotraj kroga | `KrogPremikaj` | `KrogPremikaj` (začetek) |
| `MiskaDol`, miška zunaj kroga | `KrogMiruj` (konec) | `KrogMiruj` |

Tabela 1. Tabela alternativ

V tabeli 1 je prikazano, kako pridemo do nove vrednosti signala, ki beleži, ali se krog premika ali ne. Na primer, če krog premikamo in miška ni pritisnjena, potem je nova vrednost signala `KrogMiruj`. Tri celice v tabeli imajo v oklepaju dodano

"začetek" ali "konec", ki naznanjata začetek oz. konec premikanja; zakaj moramo biti pozorni na ta dva trenutka, bomo videli v nadaljevanju. Od tod je vidno, da je nova vrednost signala odvisna tudi od prejšnje, zato si bomo za beleženje vrednosti kroga pomagali s funkcijo `foldp`.

```
belezenjeStanjaKroga : Signal StanjeKroga
belezenjeStanjaKroga =
  foldp posodobiStanjeKroga KrogMiruj postniPredal.signal
```

Funkcija `posodobiStanjeKroga` vrne novo vrednost kroga, izračunano iz stare vrednosti (začetna vrednost je `KrogMiruj`) in vrednosti miške, ki jo beleži signal poštnega predala, `postniPredal.signal`. Ravno to pa smo predstavili v tabeli 1. Funkcija `posodobiStanjeKroga` je zgolj shematskega pomena, za pravilno delovanje bomo alternativam in funkciji `znotrajKroga` podali še argumente:

```
posodobiStanjeKroga : Miska -> StanjeKroga -> StanjeKroga
posodobiStanjeKroga miska stanjeKroga =
  case stanjeKroga of
    KrogPremikaj ->
      case miska of
        MiskaGor ->
          KrogMiruj
        MiskaDol ->
          KrogPremikaj
        KrogMiruj ->
          case miska of
            MiskaGor ->
              KrogMiruj
            MiskaDol ->
              if znotrajKroga then
                KrogPremikaj
              else
                KrogMiruj
```

Signal `belezenjeStanjaKorga` opiše, kdaj se krog premika, ne pa, za koliko se premakne. Ker pa je cilj rekurzivno risanje, bo v celotni sliki več različnih krogov, bi krog radi nekako posplošili. Razmislimo, kaj vse potrebujemo za izris kroga, zato se vrnimo na začetni program, ko smo izrisali statični krog. Tam so uporabljeni trije podatki: središče kroga (koordinati $x$ in $y$) ter polmer. Definirajmo splošen zapis kroga:

```
type alias Krog =
  { x : Float, y : Float, r : Float }
```

Ker smo tipu zapisa `{ x : Float, y : Float, r : Float }` samo dodelili novo ime `Krog`, smo uporabili `type alias`.

Še enkrat premislimo, kakšen mora biti signal `belezenjeStanjaKroga`. Ugotovili smo, da bomo morali pri alternativi kroga dodati tudi dotičen krog, ki se premika oz. miruje. Ker pa je naš cilj dodajati več krogov, bi bilo to smiselno dodati v tem trenutku, k vrednosti kroga. Tako imamo en krog, s katerim nekaj delamo, in ostale kroge, ki jih bomo shranjevali v seznamu.

Posodobimo tip `StanjeKroga`, `belezenjeStanjaKroga` in `posodobiStanjeKroga`.

```
type StanjeKroga =
  KrogPremikaj Krog (List Krog)
  | KrogMiruj (List Krog)
```

Prva alternativa `KrogPremikaj Krog (List Krog)` opredeli, kateri `Krog` se premika in kateri ostanejo nespremenjeni, ti so shranjeni v `List Krog`, druga alternativa `KorgMiruj` pa opredeli vse kroge, ki smo jih dodali v sliko in z nobenim nič ne delamo.

Če si še enkrat ogledamo funkcijo `posodobiStanjeKroga`, zasledimo pogoj, če je miška znotraj kroga oz. ni. Da pa bomo lahko izračunali ta podatek, moramo v funkcijo nekako vpeljati koordinati miške. Spomnimo se signala `signalMiska`, ta vsebuje prestavljeni koordinati miške, lahko dodamo še signal `postniPredal.signal`:

```
signalMiska : Signal (Float, Float, Miska)
signalMiska =
  map3
    (\(w, h) (x, y) miska ->
      (toFloat x - toFloat w/2, toFloat h/2 - toFloat y
      , miska)
    )
    Window.dimensions Mouse.position postniPredal.signal
```

Posodobimo še signal `belezenjeStanjaKroga`.

```
belezenjeStanjaKroga : Signal StanjeKroga
belezenjeStanjaKroga =
  foldp posodobiStanjeKroga (KrogMiruj []) signalMiska
```

Preden posodobimo funkcijo `posodobiStanjeKroga`, bomo definirali funkcijo, ki bo poiskala krog, na katerega je uporabnik pritisnil. Ker ne vemo, ali bo funkcija kaj našla, bo rezultat tipa `Maybe a`. Premislimo, kaj potrebujemo za izračun: potrebujemo seznam krogov, po katerem bomo iskali, koordinati miške, kjer smo kliknili in še en seznam krogov, kjer se bodo nabirali tisti krogi, ki niso kliknjeni. Zakaj potrebujemo slednji seznam? Ker ni mogoče uporabiti zank `while` ali `for`, si moramo pomagati z rekurzijo.

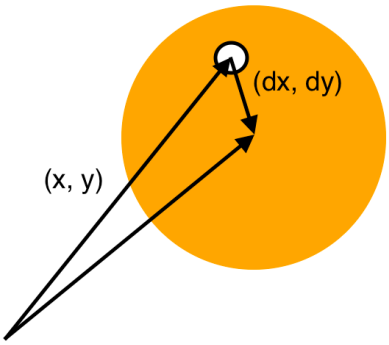
```
najdiKrog :
  List Krog -> (Float, Float) -> List Krog
  -> Maybe (Krog, List Krog)
najdiKrog seznamKrogov (x, y) zePregledaniKrogi =
  case seznamKrogov of
    krog :: ostaliKrogi ->
      if dolzinaVektorja (x - krog.x, y - krog.y) < krog.r
      then
        Just (krog, append ostaliKrogi zePregledaniKrogi)
      else
        najdiKrog ostaliKrogi (x, y)
          (krog :: zePregledaniKrogi)
    [] ->
      Nothing
```

Funkcija `dolzinaVektorja` naredi točno, kar pove njeno ime, izračuna dolžino vektorja po formuli:

$$|\vec{v}| = |(a, b)| = \sqrt{a^2 + b^2}$$

Posodobimo še funkcijo `posodobiStanjeKroga`. Funkcija `najdiKrog` najde krog, na katerega je uporabnik kliknil, nismo pa še premislili, kako se bo krog premikal. Ko se krog premika, želimo, da sta koordinati miške in koordinati središča kroga vedno oddaljeni za isti vektor. Če bomo imeli ta vektor nekje shranjen, bomo do nove pozicije kroga prišli s seštevanjem tega vektorja ter vektorja, ki opisuje položaj miške (slika 5). Vektor si bomo shranili ob trenutku, ko krog preide iz alternative `KrogMiruj` v alternativo `KrogPremikaj`. Spomnimo se, da je to en izmed posebnih trenutkov iz tabele 1. Torej bo alternativa `KrogPremikaj` potrebovala še en podatek, razdaljo med koordinatami miške in središčem kroga:

```
KrogPremikaj Krog (Float, Float) (List Krog)
```




Slika 5. Novo središče kroga se izračuna z vsoto vektorjev; vektorja `(x, y)`, ki opiše pozicijo miške, ter vektorja `(dx, dy)`, ki smo si ga shranili ob prehodu iz alternative `KrogMiruj (List Krog)` v alternativo `KrogPremikaj Krog (Float, Float) (List Krog)` in omogoča, da sta središče kroga in miška v vsakem trenutku oddaljena za isti vektor.

Sedaj smo si vse pripravili za funkcijo `posodobiStanjeKroga`, kjer moramo biti pozorni na trenutek, ko uporabnik konča s premikanjem kroga, saj moramo premikajoči se krog dodati ostalim krogom, ki se ne premikajo:

```
posodobiStanjeKroga :
  (Float, Float, Miska) -> StanjeKroga -> StanjeKroga
posodobiStanjeKroga (x, y, miska) stanjeKroga =
  case stanjeKroga of
    KrogPremikaj krog (dx, dy) seznamKrogov ->
      case miska of
        MiskaGor ->
          KrogMiruj (krog :: seznamKrogov)
        MiskaDol ->
          KrogPremikaj { krog | x = x + dx, y = y + dy }
            (dx, dy) seznamKrogov
```

```
    KrogMiruj seznamKrogov ->
      case miska of
        MiskaGor ->
          KrogMiruj seznamKrogov
        MiskaDol ->
          case najdiKrog seznamKrogov (x, y) [] of
            Just (najdenKrog, ostaliKrogi) ->
              KrogPremikaj najdenKrog
                (najdenKrog.x - x, najdenKrog.y - y)
                ostaliKrogi
            Nothing ->
              KrogMiruj seznamKrogov
```

Čeprav smo napisali kar nekaj vrstic kode, se z našim programom nič vidnega ni spremenilo, ker nismo posodobili glavne funkcije, funkcije `main`. Iz signala, ki beleži vrednost aplikacije, `belezenjeStanjaKroga`, bi radi narisali celotno sliko. Spomnimo se, da našo risalno površino določa funkcija `collage`, ki sprejme tri argumente: širino, višino in seznam z elementi tipa `Form`, zato mora tudi funkcija, ki bo narisala celotno sliko, vrniti tip `List Form`:

```
narisiSliko : StanjeKroga -> List Form
narisiSliko stanjeKroga =
  case stanjeKroga of
    KrogPremikaj krog _ seznamKrogov ->
      (premikajocKrog krog)
      :: (map (\krog -> mirujocKrog krog) seznamKrogov)
    KrogMiruj seznamKrogov ->
      map (\krog -> mirujocKrog krog) seznamKrogov
```

Funkcija `narisiSliko` iz vseh mogočih alternativ tipa `StanjeKroga` nariše celotno sliko. Različne primere smo ločili s pomočjo izraza `case`. Razlika med alternativama, ki jo vidimo na zaslonu, je, da je premikajoč krog oranžne barve, kar naredi funkcija `premikajocKrog`, funkcija `mirujocKrog` pa krog obarva rumeno. Pri alternativi `KrogPremikaj Krog (Float, Float) (List Krog)` moramo premikajoč krog dodati v seznam ostalih krogov, to smo naredili s pomočjo operatorja `::`. Mirujoče kroge smo v tip `Form` pretvorili s pomočjo funkcije `map`; na vsakem elementu seznama `seznamKrogov` smo uporabili funkcijo `morujocKrog`.

**Opomba 7.3.** V funkciji `narisiSliko` je uporabljena funkcija `map`. Slednja ni enaka kot tista funkcija `map`, ki je uporabljena na signalih, ta je vezana na seznam.

Funkciji `premikajocKrog` in `mirujocKrog` se razlikujeta le po barvi kroga, da je na sliki vidno, kateri izmed krogov je aktiven in kateri ne:

```
premikajocKrog : Krog -> Form
premikajocKrog krog =
  circle krog.r |> filled orange |> move (krog.x, krog.y)

mirujocKrog : Krog -> Form
mirujocKrog krog =
  circle krog.r |> filled yellow |> move (krog.x, krog.y)
```

Nazadnje posodobimo še glavno funkcijo, ki združuje vse, kar smo do sedaj napisali:

```
main : Signal Html
main =
  map2
    (\(w, h) stanjeKroga ->
      div
        [ onMouseDown postniPredal.address MiskaDol
        , onMouseUp postniPredal.address MiskaGor
        ]
        [ fromElement
          (collage w h (narisiSliko stanjeKroga)) ]
    )
    Window.dimensions belezenjeStanjaKroga
```

Naslednji korak proti rekurzivnemu risanju bi bil dodajanje novih krogov. V ta namen bomo prvotno sliko zmanjšali na 90 % širine ekrana, ostalih 10 % pa bo namenjenih za gumbe in podobno. Za dodajanje kroga bi lahko dodali gumb, da pa bo lepše, bomo narisali statični krog, kot smo ga na začetku. Ko bo uporabnik pritisnil na element (`div`), se bo moral zgoditi nek dogodek, ki ga poimenujmo `NovKrog`. Kot smo to naredili ob spustu oz. dvigu miške na risalno površino, ustvarimo nov poštni predal, ki bo poimenovan `postniPredalNov`. Kakšnega tipa pa bodo elementi signala novega poštnega predala? Eno alternativo smo že opisali, `NovKrog`, sicer pa naj bo `NicKliknjeno`. Ustvarimo nov tip in poštni predal:

```
type Klik =
  NicKliknjeno
  | NovKrog

postniPredalNov : Mailbox Klik
postniPredalNov =
  mailbox NicKliknjeno
```

Povežimo signal `postniPredalNov.signal` v signal `signalMiska`:

```
signalMiska : Signal (Float, Float, Miska, Klik)
signalMiska =
  map4
    (\(w, h) (x, y) miska nov ->
      (toFloat x - toFloat w/2, toFloat h/2 - toFloat y
      , miska, nov)
    )
    Window.dimensions Mouse.position
    postniPredal.signal postniPredalNov.signal
```

Popraviti moramo še funkcijo `posodobiStanjeKroga`:

```
posodobiStanjeKroga :
  (Float, Float, Miska, Klik) -> StanjeKroga -> StanjeKroga
posodobiStanjeKroga (x, y, miska, klik) stanjeKroga =
  case stanjeKroga of
```

```
    ...
    KrogMiruj seznamKrogov ->
      case klik of
        NovKrog ->
          KrogMiruj
            ({ x = 10, y = 20, r = 100 } :: seznamKrogov)
        NicKliknjeno ->
          case miska of
            ...
```

Nastavili smo, da se vedno doda krog s središčem (10, 20) in polmerom 100. Na koncu moramo posodobiti še funkcijo `main`:

```
main =
  let
    stil1 = [ ("width", "90%"), ("overflow", "hidden") ]
    stil2 = [ ("top", "0px"), ("right", "0px")
            , ("width", "10%"), ("position", "absolute")
            ]
  in
    map2
      (\(w, h) stanjeKroga ->
        div
          []
          [ div
              [ onMouseDown postniPredal.address MiskaDol
              , onMouseUp postniPredal.address MiskaGor
              , style stil1
              ]
              [ fromElement (collage w h
                                (narisiSliko stanjeKroga)) ]
          , div
              [ onMouseDown postniPredalNov.address NovKrog
              , onMouseUp postniPredalNov.address
                  NicKliknjeno
              , style stil2
              ]
              [ fromElement (collage
                                (floor (toFloat w *0.1))
                                (floor (toFloat h *0.1))
                                [ circle 20 |> filled
                                                green ]) ]
          ]
      )
      Window.dimensions belezenjeStanjaKroga
```

Kot smo rekli, smo dodali nov HTML element (`div`), kjer je narisan statični zelen krog. Če uporabnik klikne na območje, kjer je omenjeni zeleni krog, se doda na sliko nov krog. Opazimo nov atribut v prvem argumentu funkcije `div`, stil. Če bi pisali HTML kodo, bi stile predstavljal oblikovni jezik CSS.

Prišli smo do točke, ko lahko dodamo rekurzivnost v našo spletno aplikacijo. Kaj želimo doseči? Da bo uporabnik lahko iz osnovnih elementov, v našem primeru so to krogi, sestavil sliko, ki jo bo lahko spet uporabil kot osnovni element.

Aplikacija ima trenutno dve mogoči alternativi, ki opredelita, ali se krog premika ali ne. Poglejmo na celotno sliko malo drugače. Slike ne sestavljajo več samo samostojni krogi, ampak je na sliki lahko tudi slika. Od tod rekurzivnost. Recimo, da imamo nek gumb, ki bo trenutno sliko podvojil, bo sliki dodal sliko. Postavi se vprašanje, kam bi lahko to sliko dodali. V ta namen tipu zapisa dodelimo novo ime, poimenujmo ga `Slika`:

```
type alias Slika =
  { seznamKrogov : List Krog, seznamSlik : List Slika }
```

Zapis je ravno tak, kot smo ga opisali: sestavljen je iz osnovnih krogov in iz slik. Tip `Krog` smo definirali in zato spada v skupino že obstoječih tipov, tip `Slika` pa kliče samega sebe, tukaj nastane problem in Elm nas opozori nanj. Zato moramo ustvariti čisto nov tip.

```
type Fotografija =
  SeznamSlik (List Slika)

type alias Slika =
  { seznamKrogov : List Krog, seznamSlik : Fotografija }
```

Sedaj, ko je polje `seznamSlik` tipa `Fotografija`, ki spada med znane, že definirane tipe, je tip vrednosti `Slika` pravilno definiran.

Kakšne bodo sedaj možne alternative aplikacije? Alternativa, ko premikamo krog, `KrogPremikaj Krog (Float, Float) (List Krog)`, se spremeni v alternativo `KrogPremikaj Krog (Float, Float) Slika`. Spremenili smo le, da vrednosti, ki se ne premikajo, niso več samo seznam krogov, ampak je kar celotna slika, iz katere smo vzeli premikajoči se krog. Rekli smo, da se bo vrednost tipa `Slika` enako obnašala kot vrednost tipa `Krog`, se pravi lahko premikamo tudi vrednosti tipa `Slika`. Dodamo novo alternativo `SlikaPremikaj Slika (Float, Float) Slika`. Aplikacija je v tej alternativi, ko uporabnik pritisne na katerega izmed krogov, ki sestavljajo sliko, premikajo pa se vsi pripadajoči krogi, ne le pritisnjeni. Alternativa `KrogMiruj (List Krog)` pa se prelevi v `SlikaMiruj Slika`. Funkcija `posodobiStanjeKroga` se ne spremeni prav bistveno, paziti moramo le na spremembe iz tipa `List Krog` na `Slika` in na iskanje premikajočega objekta, ker ni nujno, da je samo krog, lahko je tudi `Slika`. Kar pa je novo, je alternativa `SlikaPremikaj Slika (Float, Float) Slika`. Pri premikanju kroga smo razmišljali na način: miška je od središča med premikanjem vedno oddaljena za enak vektor, ki smo ga na začetku shranili. Sedaj pa imamo več takšnih vektorjev, ki bi si jih morali zapomniti, zato bomo tipu zapisa `Krog` dodali še dve polji: `xx : Float` in `yy : Float`. Novi dodani polji si bosta zapomnili, kje je bil krog ob trenutku, ko začnemo premikati sliko. Na koncu, ko se slika ne premika več, ne smemo pozabiti posodobiti polji `xx` in `yy`. Sedaj moramo posodobiti le še funkciji `narisiSliko` in `main`. V podrobnosti kode se ne bomo spuščali, saj se vse rekurzivno posploši.

Program lahko poljubno izpopolnimo, na primer:

- spreminjanje velikosti krogov,
- dodajanje kvadratov,
- izrisovanje slike na desni, kot je narisan statični krog,

- dodajanje osnovnih objektov na mesto klika, ne na fiksno mesto,...

7.3. **Deli programa.** V prejšnjem podrazdelku 7.2 smo ob vsakem spreminjanju, dodajanju, posodabljanju spletne aplikacije morali biti pozorni na:

- model: sem sodita `Krog` in `Slika`;
- posodobitveni del: temu delu pripada funkcija `posodobiStanjeKroga` in kar paše k njej (novi tipi, signali);
- prikazna funkcija: to je funkcija, ki nam iz vrednosti aplikacije izriše sliko na zaslon, v našem primeru je bila to funkcija `narisiSliko`.

Našteti elementi programa so tipični za aplikacijo, napisano v programskem jeziku Elm. Tudi jezik nas prisili, da je program tako napisan, če želimo, da deluje pravilno.

## 8. Elm kot nereagenten programski jezik

Do sedaj smo uporabljali Elmovo različico 0.16, ki v tem trenutku ni najnovejša. Maja 2016 je bila objavljena verzija 0.17 [5], v kateri ni več signalov, programski jezik ni več reagenten, temveč funkcijski jezik. Razlog za izključitev signalov je težko razumevanje le-teh. Oblika programa je postala strožja, kar pomeni, da je programiranje bolj vodeno, oblika pa je vnaprej določena.

8.1. **Elm 0.17.** Oblika programa v Elmu 0.17 je bila povzeta po vzorcu testnega paketa `StartApp`, ki je bil dostopen verziji 0.16 [34]:

```
import StartApp

main =
  app.html

app =
  StartApp.start
    { init = zacetniModel, update = posodobitvenaFunkcija
    , view = prikaznaFunkcija, inputs = seznamSignalov }
```

Tipična aplikacija zapisana z Elmom 0.17:

```
import Html.App

main =
  Html.App.program
    { init = zacetniModel
    , update = posodobitvenaFunkcija
    , view = prikaznaFunkcija
    , subscriptions = narocnina }
```

Opišimo zgornje funkcije:

- `zacetniModel`: konstantna funkcija, ki opiše začetno vrednost modela
- `posodobitvenaFunkcija`: funkcija, ki modelu priredi novo vrednost
- `prikaznaFunkcija`: poskrbi za izgled aplikacije
- `narocnina`: v Elmu 0.16 bi bil na tem mestu seznam signalov, ki jih ni v različici 0.17, nam pa ta funkcija omogoča spreminjanje programa ob dogodkih iz okolja; primer 8.1 prikazuje, kako bi funkcija izgledala na primeru poenostavljenega rekurzivnega risanja

**Primer 8.1.** Na praktičnem primeru poenostavljenega rekurzivnega risanja poglejmo, kaj je zamenjalo signale:

```
narocnina : Slika -> Sub StanjeSlike
narocnina slika =
  if slika.premikanje then
    batch
      [ Mouse.downs (\{ x, y } ->
                      dolociStanje slika
                        (toFloat x) (toFloat y))
      , Mouse.moves (\{ x, y } ->
                      PremikajSliko
                        (toFloat x, toFloat y))
      , Mouse.ups (\{ x, y } ->
                    KoncajPremikanjeLik)
      ]
  else
    batch
      [ Mouse.downs (\{ x, y } ->
                      dolociStanje slika
                        (toFloat x) (toFloat y))
      ]
```

♢

V primeru 8.1 se pojavi nov tip vrednosti, `Sub msg`, ki je zamenjal signale. Vloga funkcije `narocnina` je, da iz dogodkov iz okolja (npr. pritisk na miško in premikanje miške) priredi novo vrednost aplikacije, v našem primeru je vrednost tipa `StanjeSlike`. Da lahko opazujemo več različnih dogodkov, smo za združitev teh uporabili funkcijo `batch` [32]:

```
batch : List (Sub msg) -> Sub msg
```

Poznamo še dve funkciji, ki delujeta na vrednostih tipa `Sub msg`:

```
map : (a -> msg) -> Sub a -> Sub msg
```

in

```
none : Sub msg
```

Prevedimo v jezik signalov: V naročitveni funkciji `narocnina` povemo, katere signale bomo opazovali in kako njihove spremembe vplivajo na aplikacijo. Signal `Mouse.position` se je prevedel v funkcijo `Mouse.moves`, ki za svoj argument sprejme funkcijo, ta pa poziciji miške priredi novo vrednost aplikacije, zapisano s simboli [31]:

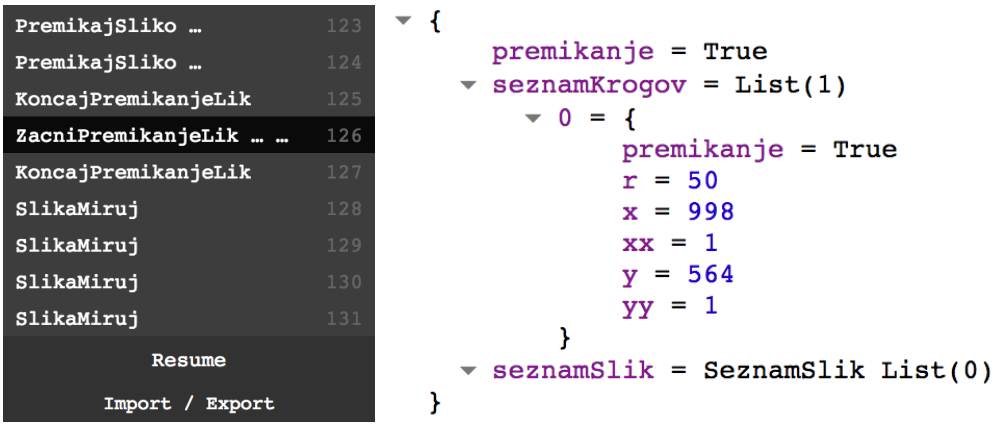
```
moves : (Position -> msg) -> Sub msg
```

8.2. **Elm 0.18.** Trenutno najnovejša različica Elma je 0.18, ki je bila predstavljena novembra 2016. Od različice 0.17 se ne razlikuje prav veliko, ima pa zopet omogočeno potovanje v času [4].

Če vklopimo razhroščevalnik, se med izvajanjem programa na zaslonu prikaže dodatno okno (slika 6), ki ima tri dodatne možnosti:


Slika 6. Razhroščevalnik v Elmovi različici 0.18 omogoča pregled preteklih dogodkov, uvoz in izvoz zgodovine dogodkov.

- "razišči zgodovino": klik na gumb nam odpre dodatno okno, v katerem je seznam vseh preteklih vrednosti aplikacije (slika 7), s klikanjem nanje se lahko premikamo v zgodovino, poleg seznama pa imamo pogled tudi na naš model, kaj se z njim dogaja oz. spreminja;
- "uvozi" / "izvozi": zgodovino iz prejšnje točke lahko izvozimo v .txt datoteko, ki jo lahko uvozimo na drugem računalniku, kar omogoča ponovitev dogodkov in s tem lažje odkrivanje napak.




Slika 7. Na desni je prikazan model v trenutku, ki smo ga izbrali med preteklimi dogodki na levi.

## 9. Dodatek

V nadaljevanju je napisana celotna programska koda poenostavljenega rekurzivnega risanja, ki je opisana v razdelku 7.

```
import Color exposing (..)
import Graphics.Collage exposing (..)
import Mouse
import Window
import Html exposing (..)
import Html.Attributes exposing (..)
import Html.Events exposing (..)
```

```
-- MODEL


type alias Krog =
  { x : Float, y : Float, xx : Float, yy : Float, r : Float }


type alias Slika =
  { seznamKrogov : List Krog, seznamSlik : Fotografija }


type Fotografija
  = SeznamSlik (List Slika)



-- POSODOBITVENI DEL


type Miska
  = MiskaGor
  | MiskaDol


type Klik
  = NicKliknjeno
  | NovKrog
  | DodajSliko


type StanjeKroga
  = KrogPremikaj Krog ( Float, Float ) Slika
  | SlikaPremikaj Slika ( Float, Float ) Slika
  | SlikaMiruj Slika


belezenjeStanjaKroga : Signal StanjeKroga
belezenjeStanjaKroga =
  let
    zacetnaSlika =
      { seznamKrogov = [ { x = 10, y = 20, xx = 10, yy = 20, r = 100 } ]
      , seznamSlik = SeznamSlik [] }
  in
    Signal.foldp posodobiStanjeKroga (SlikaMiruj zacetnaSlika) signalMiska


posodobiStanjeKroga :
  ( Float, Float, Miska, Klik ) -> StanjeKroga -> StanjeKroga
posodobiStanjeKroga ( x, y, miska, klik ) stanjeKroga =
  case stanjeKroga of
    KrogPremikaj krog ( dx, dy ) slika ->
      case miska of
        MiskaGor ->
          SlikaMiruj { slika | seznamKrogov = krog :: slika.seznamKrogov }

        MiskaDol ->
          KrogPremikaj { krog | x = x + dx, y = y + dy } ( dx, dy ) slika
```

```
    SlikaMiruj slika ->
      case klik of
        NovKrog ->
          SlikaMiruj { slika | seznamKrogov = { x = 10, y = 20, xx = 10
                                              , yy = 20, r = 100 }
                                              :: slika.seznamKrogov }

        DodajSliko ->
          SlikaMiruj { slika | seznamSlik = dodajSliko slika
                                              slika.seznamSlik }

        NicKliknjeno ->
          case miska of
            MiskaGor ->
              SlikaMiruj slika

            MiskaDol ->
              case najdiKrog slika.seznamKrogov ( x, y ) [] of
                Just ( najdenKrog, ostaliKrogi ) ->
                  KrogPremikaj najdenKrog
                    ( najdenKrog.x - x, najdenKrog.y - y )
                    { slika | seznamKrogov = ostaliKrogi }

                Nothing ->
                  case najdiSliko slika.seznamSlik ( x, y ) [] of
                    Just ( najdenaSlika, ostaleSlike ) ->
                      SlikaPremikaj najdenaSlika ( x, y )
                        { slika | seznamSlik = SeznamSlik ostaleSlike }

                    Nothing ->
                      SlikaMiruj slika

    SlikaPremikaj slika ( dx, dy ) ostanekSlike ->
      case miska of
        MiskaGor ->
          SlikaMiruj { ostanekSlike | seznamSlik = dodajSliko slika
            ostanekSlike.seznamSlik }

        MiskaDol ->
          SlikaPremikaj
            { slika
              | seznamKrogov = premakniKroge slika.seznamKrogov
                                 ( dx, dy ) ( x, y ) []
              , seznamSlik = premakniFotografijo slika.seznamSlik
                               ( dx, dy ) ( x, y ) []
            }
            ( dx, dy )
            ostanekSlike


premakniKroge :
  List Krog -> ( Float, Float ) -> ( Float, Float ) -> List Krog
  -> List Krog
premakniKroge krogi ( dx, dy ) ( x, y ) zePremaknjeniKrogi =
  case krogi of
    k :: kx ->
```

```
      premakniKroge kx ( dx, dy ) ( x, y )
        ({ k | x = x + k.xx - dx, y = y + k.yy - dy } :: zePremaknjeniKrogi
        )

    [] ->
      zePremaknjeniKrogi


premakniFotografijo :
  Fotografija -> ( Float, Float ) -> ( Float, Float ) -> List Slika
  -> Fotografija
premakniFotografijo fotografija ( dx, dy ) ( x, y ) zePremaknjeneSlike =
  case fotografija of
    SeznamSlik seznam ->
      case seznam of
        s :: sx ->
          premakniFotografijo (SeznamSlik sx)
            ( dx, dy )
            ( x, y )
            ({ s
              | seznamKrogov = premakniKroge s.seznamKrogov ( dx, dy )
                                 ( x, y ) []
              , seznamSlik = premakniFotografijo s.seznamSlik ( dx, dy )
                               ( x, y ) []
             }
              :: zePremaknjeneSlike
            )

        [] ->
          SeznamSlik zePremaknjeneSlike


najdiKrog :
  List Krog -> ( Float, Float ) -> List Krog -> Maybe ( Krog, List Krog )
najdiKrog seznamKrogov ( x, y ) zePregledaniKrogi =
  case seznamKrogov of
    krog :: ostaliKrogi ->
      if dolzinaVektorja ( x - krog.x, y - krog.y ) < krog.r then
        Just ( krog, List.append ostaliKrogi zePregledaniKrogi )
      else
        najdiKrog ostaliKrogi ( x, y ) (krog :: zePregledaniKrogi)

    [] ->
      Nothing


najdiSliko :
  Fotografija -> ( Float, Float ) -> List Slika
  -> Maybe ( Slika, List Slika )
najdiSliko fotografija ( x, y ) zePregledaneSlike =
  case fotografija of
    SeznamSlik seznam ->
      case seznam of
        s :: sx ->
          case najdiKrog s.seznamKrogov ( x, y ) [] of
            Just ( najdenKrog, ostaliKrogi ) ->
              Just ( s, List.append sx zePregledaneSlike )
```

```
            Nothing ->
              case najdiSliko s.seznamSlik ( x, y ) zePregledaneSlike of
                Just ( najdenaSlika, ostaleSlike ) ->
                  Just ( s, List.append sx zePregledaneSlike )

                Nothing ->
                  najdiSliko (SeznamSlik sx) ( x, y )
                    (s :: zePregledaneSlike)


        [] ->
          Nothing


dodajSliko : Slika -> Fotografija -> Fotografija
dodajSliko slika fotografija =
  let
    posodobljenaSlika =
      spremeniXXYY slika
  in
    case fotografija of
      SeznamSlik seznam ->
        SeznamSlik (posodobljenaSlika :: seznam)


spremeniXXYY : Slika -> Slika
spremeniXXYY slika =
  { slika | seznamKrogov = spremeniXXYYkrogi slika.seznamKrogov []
          , seznamSlik = spremeniXXYYfotografija slika.seznamSlik [] }


spremeniXXYYkrogi : List Krog -> List Krog -> List Krog
spremeniXXYYkrogi krogi zeSpremenjeniKrogi =
  case krogi of
    k :: kx ->
      spremeniXXYYkrogi kx
        ({ k | xx = k.x, yy = k.y } :: zeSpremenjeniKrogi)

    [] ->
      zeSpremenjeniKrogi


spremeniXXYYfotografija : Fotografija -> List Slika -> Fotografija
spremeniXXYYfotografija fotografija zeSpremenjeneSlike =
  case fotografija of
    SeznamSlik seznam ->
      case seznam of
        s :: sx ->
          let
            posodobljenaSlika =
              { s
                | seznamKrogov = spremeniXXYYkrogi s.seznamKrogov []
                , seznamSlik = spremeniXXYYfotografija s.seznamSlik []
              }
          in
            spremeniXXYYfotografija (SeznamSlik sx)
```

```
            (posodobljenaSlika :: zeSpremenjeneSlike)

        [] ->
          SeznamSlik zeSpremenjeneSlike


dolzinaVektorja : ( Float, Float ) -> Float
dolzinaVektorja ( a, b ) =
  (a ^ 2 + b ^ 2) ^ (1 / 2)


signalMiska : Signal ( Float, Float, Miska, Klik )
signalMiska =
  Signal.map4 (\( w, h ) ( x, y ) miska nov ->
                ( toFloat x - toFloat w / 2, toFloat h / 2 - toFloat y
                , miska, nov ))
    Window.dimensions
    Mouse.position
    postniPredal.signal
    postniPredalNov.signal


postniPredal : Signal.Mailbox Miska
postniPredal =
  Signal.mailbox MiskaGor


postniPredalNov : Signal.Mailbox Klik
postniPredalNov =
  Signal.mailbox NicKliknjeno



-- PRIKAZOVALNI DEL


premikajocKrog : Krog -> Form
premikajocKrog krog =
  circle krog.r |> filled orange |> move ( krog.x, krog.y )


premikajocaFotografija : Fotografija -> List Form
premikajocaFotografija fotografija =
  case fotografija of
    SeznamSlik seznam ->
      case seznam of
        s :: sx ->
          List.append
            (List.append
              (List.map (\krog -> premikajocKrog krog) s.seznamKrogov)
              (premikajocaFotografija s.seznamSlik)
            )
            (premikajocaFotografija (SeznamSlik sx))

        [] ->
          []
```

```
premikajocaSlika : Slika -> List Form
premikajocaSlika slika =
  List.append
    (List.map (\krog -> premikajocKrog krog) slika.seznamKrogov)
    (premikajocaFotografija slika.seznamSlik)


mirujocKrog : Krog -> Form
mirujocKrog krog =
  circle krog.r |> filled yellow |> move ( krog.x, krog.y )


mirujocaFotografija : Fotografija -> List Form
mirujocaFotografija fotografija =
  case fotografija of
    SeznamSlik seznam ->
      case seznam of
        s :: sx ->
          List.append
            (List.append
              (List.map (\krog -> mirujocKrog krog) s.seznamKrogov)
              (mirujocaFotografija s.seznamSlik)
            )
            (mirujocaFotografija (SeznamSlik sx))

        [] ->
          []


mirujocaSlika : Slika -> List Form
mirujocaSlika slika =
  List.append
    (List.map (\krog -> mirujocKrog krog) slika.seznamKrogov)
    (mirujocaFotografija slika.seznamSlik)


narisiSliko : StanjeKroga -> List Form
narisiSliko stanjeKroga =
  case stanjeKroga of
    KrogPremikaj krog _ slika ->
      (premikajocKrog krog) :: (mirujocaSlika slika)

    SlikaPremikaj slika _ seznamSlik ->
      List.append (premikajocaSlika slika) (mirujocaSlika seznamSlik)

    SlikaMiruj slika ->
      List.append
        (List.map (\krog -> mirujocKrog krog) slika.seznamKrogov)
        (mirujocaFotografija slika.seznamSlik)


main : Signal Html
main =
  let
    stil1 =
      [ ( "width", "90%" ), ( "overflow", "hidden" ) ]
```

```
  stil2 =
    [ ( "top", "20px" ), ( "right", "0px" ), ( "width", "10%" )
    , ( "position", "absolute" )
    ]

  stil3 =
    [ ( "top", "1%" ), ( "right", "0px" ), ( "width", "10%" )
    , ( "position", "absolute" )
    ]
in
  Signal.map2
    (\( w, h ) stanjeKroga ->
      div
        []
        [ div
            [ onMouseDown postniPredal.address MiskaDol
            , onMouseUp postniPredal.address MiskaGor
            , style stil1
            ]
            [ fromElement (collage w h (narisiSliko stanjeKroga)) ]
        , div
            [ onMouseDown postniPredalNov.address NovKrog
            , onMouseUp postniPredalNov.address NicKliknjeno
            , style stil2
            ]
            [ fromElement (collage (floor (toFloat w * 0.1))
                                   (floor (toFloat h * 0.1))
                                   [ circle 20 |> filled green ]) ]
        , div
            [ style stil3 ]
            [ button
              [ onMouseDown postniPredalNov.address DodajSliko
              , onMouseUp postniPredalNov.address NicKliknjeno
              , style [ ( "width", "100%" ) ]
              ]
              [ Html.text "Dodaj sliko" ]
            ]
        ]
    )
    Window.dimensions
    belezenjeStanjaKroga
```

## Slovar strokovnih izrazov

**arrowized FRP** FRP s puščicami
**behavior** vrednost, ki se stalno spreminja s časom
**error message** sporočilo o napaki
**event-driven FRP** dogodkovno FRP
**first-class function** funkcija kot prvovrstna vrednost
**first-order FRP** FRP prvega reda
**Functional reactive animation** Funkcijsko reagentna animacija
**garbage collection** čiščenje spomina
**higher-order function** funkcija višjega reda
**push-pull FRP** FRP s pristopom potisni in povleci

**real-time FRP** FRP v realnem času
**tail-call** klic z repa
**two-tiered programming language** programski jezik iz dveh delov

## Literatura


[1] H. Barendregt in E. Barendsen, *Introduction to Lambda Calculus*, 1998, dostopno na `ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf`.

[2] E. Bruno, *Tail Call Optimization and Java*, verzija 15. 4. 2014, [ogled 13.3.2017], dostopno na `www.drdobbs.com/jvm/tail-call-optimization-and-java/240167044`.

[3] E. Czaplicki, *Elm: Concurrent FRP for functional GUIs*, magistrsko delo, Harvard, 2012, dostopno na `www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf`.

[4] E. Czaplicki, *The Perfect Bug Report*, verzija 14. 11. 2016, [ogled 10. 4. 2017], dostopno na `elm-lang.org/blog/the-perfect-bug-report`.

[5] E. Czaplicki, *Upgrading to 0.17*, verzija 12. 5. 2016, [ogled 10. 4. 2017], dostopno na `github.com/elm-lang/elm-platform/blob/master/upgrade-docs/0.17.md`.

[6] C. M. Elliott, *Push-pull functional reactive programming*, Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Stephanie Weirich), ACM, Edinburgh, 2009, str. 25–36.

[7] C. Elliott in P. Hudak, *Functional Reactive Animation*, v: Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (Simon L. Peyton Jones, Mads Tofte in A. Michael Berman), ACM, Amsterdam, 1997, str. 263–273.

[8] M. Goldstein, *Using Mailboxes in Elm*, verzija 30. 7. 2015, [ogled 15. 3. 2017], dostopno na `gist.github.com/mgold/461dbf37d4d34767e5da`.

[9] M. Lipovača, *Learn You a Haskell for Great Good!*, No Starch Press, 2011.

[10] M. Pretnar, *Haskell 2*, verzija 2016, [ogled 20. 1. 2017], dostopno na `vimeo.com/191460259`.

[11] A. Rauschmayer, *Tail call optimization in ECMAScript 6*, verzija 30. 6. 2015, [ogled 1. 6. 2017], dostopno na `2ality.com/2015/06/tail-call-optimization.html`.

[12] R. Rojas, *A Tutorial Introduction to the Lambda Calculus*, CoRR **abs/1503.09060** (2015), dostopno na `arxiv.org/pdf/1503.09060.pdf`.

[13] A. M. Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, Proc. London Math. Soc. **2** (1936) 230–265.

[14] P. Wadler, *Propositions as types*, Commun. ACM **58** (2015) 75–84.

[15] Z. Wan, W. Taha in P. Hudak, *Event-Driven FRP*, v: Practical Aspects of Declarative Languages, 4th International Symposium, PADL 2002 (Shriram Krishnamurthi in C. R. Ramakrishnan), Lecture Notes in Computer Science **2257**, Springer, Portland, 2002, str. 155–172.

[16] Z. Wan, W. Taha in P. Hudak, *Real-Time FRP*, v: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (Benjamin C. Pierce), ACM, Firence, 2001, str. 146–156.

[17] *An Introduction to Elm*, [ogled 12. 12. 2016], dostopno na `guide.elm-lang.org/`.

[18] *Anonymous function*, v Wikipedia: The Free Encyclopedia, [ogled 20. 1. 2017], dostopno na `en.wikipedia.org/wiki/Anonymous_function`.

[19] *Closures*, [ogled 21. 1. 2017], dostopno na `developer.mozilla.org/en-US/docs/Web/JavaScript/Closures`.

[20] *"Controlling Time and Space: understanding the many formulations of FRP" by Evan Czaplicki*, verzija 21. 9. 2014, [ogled 10. 10. 2015], dostopno na `www.youtube.com/watch?v=Agu6jipKfYw`.

[21] *Elm Syntax*, [ogled 29. 5. 2017], dostopno na `elm-lang.org/docs/syntax`.

[22] *Elm tutorial*, [ogled 20. 12. 2016], dostopno na `www.elm-tutorial.org/index.html`.

[23] *Elm's Time Travelling Debugger*, verzija 2014, [ogled 5. 4. 2016], dostopno na `debug.elm-lang.org/`.

[24] *Evan Czaplicki - Let's be mainstream! User focused design in Elm – Curry On*, verzija 14. 7. 2015, [ogled 10. 10. 2015], dostopno na `www.youtube.com/watch?v=oYk8CKH7OhE`.

[25] *First-class function*, v Wikipedia: The Free Encyclopedia, [ogled 20. 1. 2017], dostopno na `en.wikipedia.org/wiki/First-class_function`.

[26] *Functional programming*, v: Wikipedia: The Free Encyclopedia, [ogled 20. 10. 2015], dostopno na `en.wikipedia.org/wiki/Functional_programming`.

[27] *Garbage collection (computer science)*, v Wikipedia: The Free Encyclopedia, [ogled 13. 2. 2017], dostopno na `en.wikipedia.org/wiki/Garbage_collection_(computer_science)`.
[28] *Higher-order function*, v Wikipedia: The Free Encyclopedia, [ogled 20. 1. 2017], dostopno na `en.wikipedia.org/wiki/Higher-order_function`.
[29] *Maybe*, [ogled 29. 5. 2017], dostopno na `package.elm-lang.org/packages/elm-lang/core/2.1.0/Maybe`.
[30] *Memory Management*, [ogled 1. 6. 2017], dostopno na `developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management`.
[31] *Mouse*, [ogled 22. 5. 2017], dostopno na `package.elm-lang.org/packages/elm-lang/mouse/1.0.1/Mouse`.
[32] *Platform.Sub*, [ogled 21. 5. 2017], dostopno na `package.elm-lang.org/packages/elm-lang/core/5.1.1/Platform-Sub`.
[33] *Signal*, [ogled 5. 4. 2016], dostopno na `package.elm-lang.org/packages/elm-lang/core/2.1.0/Signal`.
[34] *StartApp*, [ogled 21. 5. 2017], dostopno na `package.elm-lang.org/packages/evancz/start-app/latest/StartApp`.