Sistem za perzistenciju podataka u programskom jeziku Clojure

Novak Boškov

Fakultet Tehničkih Nauka, Novi Sad

Master rad

Oktobar 2016

	УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА** 21000 НОВИ САД, Трг Доситеја Обрадовића 6	Број:
		Датум:
	ЗАДАТАК ЗА МАСТЕР РАД	

(Податке уноси предметни наставник - ментор)

СТУДИЈСКИ ПРОГРАМ:	**Рачунарство и аутоматика**
РУКОВОДИЛАЦ СТУДИЈСКОГ ПРОГРАМА:	**Проф. др Мирослав Поповић**

Студент:	Новак Бошков	Број индекса:	E2 33/15
Област:	Језици специфични за домен		
Ментор:	др Игор Дејановић		

НА ОСНОВУ ПОДНЕТЕ ПРИЈАВЕ, ПРИЛОЖЕНЕ ДОКУМЕНТАЦИЈЕ И ОДРЕДБИ СТАТУТА ФАКУЛТЕТА ИЗДАЈЕ СЕ ЗАДАТАК ЗА МАСТЕР РАД, СА СЛЕДЕЋИМ ЕЛЕМЕНТИМА:

- проблем – тема рада;
- начин решавања проблема и начин практичне провере резултата рада, ако је таква провера неопходна;

НАСЛОВ МАСТЕР РАДА:

Систем за перзистенцију података у програмском језику Clojure

ТЕКСТ ЗАДАТКА:

Дизајнирати и имплементирати библиотеку за опис мапирања и перзистенцију података на програмском језику Clojure. Систем учинити независним од парадигме циљне базе података.

Користити препоручену праксу из области функционалног програмирања и перзистенције података. Документовати решење.

Руководилац студијског програма:	Ментор рада:

Примерак за: ☐ - Студента; ☐ - Ментора

Sadržaj

Slike

Glava 1

Uvod

1.1 Lisp

Lisp (ili **LISP**) je porodica programskih jezika originalno specificirana 1958. godine od strane Džona Mekartija (*John McCarthy*) i grupe za veštačku inteligenciju sa Tehnološkog instituta Masačusets (*MIT*) [37]. Prva implementacija LISP-a izvedena je na *IBM 704* računarima u svrhu lakšeg eksperimentisanja sa sistemom *Advice Taker* koji je razvijao Mekarti sa svojim timom. Glavni zahtev projekta je bio da se napravi sistem za programiranje koji bi omogućio manipulaciju izrazima — formalizovanim deklarativnim i proceduralnim rečenicama nad kojim bi dalje *Advice Taker* bio u stanju da vrši dedukciju.

Na pravcu svog razvoja Lisp prolazi kroz različite faze pojednostavljivanja da bi na kraju njegova baza bila svedena na šemu za predstavljanje parcijalno rekurzivnih funkcija, jednu vrstu simboličkih izraza koji se od tada pa nadalje nazivaju *S-expressions*, a odgovarajuće funkcije *S-functions*. Čitav Lisp sistem biva sveden na izvestan broj matematičkih ideja i notacija nad kojima se gradi kompletan sistem za programiranje u kome univerzalna S-funkcija *apply* ima teorijsku ulogu univerzalne Tjuringove mašine dok sa praktičnog stanovišta predstavlja interpreter.
U nastavku su dati originalni koncepti koji definišu Lisp uz komentare o značaju pojedinih koncepata kao i njihovoj vezi sa konceptima koji su u širokoj upotrebi u različitim oblastima računarstva danas.

(a) *Parcijalne funkcije*
Funkciju nazivamo parcijalnom kada je definisana samo na delu svog domena. Parcijalne funkcije se prirodno nameću kada se funkcije definišu izračunavanjima jer za neke vrednosti argumenata izračunavanja koja definišu vrednost funkcije ne moraju biti konačna.

(b) *Propozicioni izrazi i predikati*

Propozicioni izraz je onaj čija je vrednost za tačnu tvrdnju T a za netačnu F, predikat tako predstavlja funkciju čiji kodomen sadrži samo istinitosne vrednosti — T i F.

(c) *Kondicijalni izrazi*

Kondicijalni izrazi su notacija za simboličko predstavljanje zavisnosti vrednosti bilo koje vrste od predikata. U ovu svrhu se tradicionalno koriste reči govornog jezika. Kondicijalni izraz ima sledeću formu

$$(p_1 \to e_1, ..., p_n \to e_n) \tag{1.1}$$

gde su p propozicioni izrazi a e izrazi bilo koje vrste. Izraz 1.1 se može čitati kao „Ako je p_1 onda e_1 inače ako je p_2 onda e_2, ..., inače ako p_n onda e_n“ ili „p_1 daje e_1, p_2 daje e_2, ... , p_n daje e_n“.

Mekarti daje niz pravila koja određuju da li je vrednost izraza 1.1 definisana i ako jeste koja mu je vrednost. Idući sa leva na desno — ako se neko p čija je vrednost T pojavi pre nekog e čija je vrednost nedefinisana onda je vrednost kondicionog izraza vrednost odgovarajućeg e (ako je ono definisano). Ako se neko nedefinisano p pojavi pre nekog tačnog p, ili ako su svi p netačni, ili ako je e koje odgovara prvom tačnom p nedefinisano onda je vrednost kondicionog izraza nedefinisana. Primeri upotrebe ovih pravila dati su u nastavku:

$$\begin{gathered}(4 < 5 \to 1, 1 > 2 \to 3) = 1,\\ (2 < 1 \to 4, 2 > 1 \to 3, 2 > 1 \to 2) = 3,\\ (2 < 1 \to 3, T \to \frac{0}{0}) = nedefinisano,\\ (2 < 1 \to 3, 4 < 1 \to 4) = nedefinisano\end{gathered} \tag{1.2}$$

Kondicijalni izrazi se koriste pri definisanju matematičkih funkcija kao što su:

$$\begin{gathered}|x| = (x < 0 \to -x, T \to x),\\ sgn(x) = (x < 0 \to -1, x = 0 \to 0, T \to 1)\end{gathered} \tag{1.3}$$

(d) *Rekurzivne definicije funkcija*

Koristeći kondicijalne izraze možemo, bez cirkularnosti, definisati funkcije pomoću formula u kojima se pojavljuju funkcije koje upravo definišemo. Kao

primer Mekarti originalno navodi i formulaciju algoritma za pronalaženje najvećeg zajedničkog delioca:

$$\begin{aligned} nzd(m,n) = (m > n \rightarrow nzd(n,m), \\ rem(n,m) = 0 \rightarrow m, \\ T \rightarrow nzd(rem(n,m),m)) \end{aligned} \tag{1.4}$$

(e) *Funkcije i forme*

Uvideći da je u matematici čest slučaj nepreciznog korišćenja termina *funkcija* i da se on često uzima i za izraze oblika $y^2 + x$ Mekarti zaključuje da pošto mu je nadalje potrebno izračunavanje nad izrazima za funkcije ima potrebu za razlikovanjem formi od funkcija i za notacijom kojom bi se ta razlika iskazala. Potrebno razlikovanje i notaciju za njegovu predstavu, od koje odstupa trivijalno, nalazi u Črčovom (*Alonzo Church*) λ-kalkulusu [26].

Črč izraze oblika y^2+x zove formama. Forma može biti konvertovana u funkciju ako je moguće naći korespodenciju između promenjivih forme i prosleđene liste argumenata odgovarajuće funkcije. Ovo se postiže Črčovom λ-notacijom.

Ako je ε forma promenljivih $x_1, ..., x_n$ onda će $\lambda((x_1, ..., x_n), \varepsilon)$ biti uzeto kao funkcija od n promenljivih čija se vrednost određuje zamenom argumenata za promenljive $x_1, ..., x_n$ u redosledu iz ε i izračunavanjem rezultujućeg izraza. Na primer $\lambda((x, y), y^2 + x)$ je funkcija dve promenljive i $\lambda((x, y), y^2 + x)(3, 4) = 19$. Uvođenjem ove notacije uz još neka pravila dobija se i mogućnost korišćenja funkcija kao parametara drugih funkcija.

(f) *Izrazi za rekurzivne funkcije*

λ notacija je neadekvatna za imenovanje funkcija definisanih rekurzivno pa autor uvodi novu *label* notaciju. Korišćenjem λ-notacije može se svesti:

$$sqrt(a, x, \epsilon) = (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow sqrt(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon)) \tag{1.5}$$

na

$$sqrt = \lambda((a, x, \epsilon), (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow sqrt(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon))) \tag{1.6}$$

ali desna strana jednakosti ne može služiti kao izraz za funkciju jer nema ničega što ukazuje da referenciranje simbola *sqrt* ima značenje celog izraza. Upravo za rešavanje ovog problema uvedena je *label* notacija. $label(a, \varepsilon)$ denotira ε tako da

se pojavljivanja a unutar ε interpretiraju kao referenciranja na izraz kao celinu. Tako bi mogli napisati

$$label(sqrt, \lambda((a, x, \epsilon), (|x^2 - a| < \epsilon \rightarrow x, T \rightarrow sqrt(a, \frac{1}{2}(x + \frac{a}{x}), \epsilon)))) \qquad (1.7)$$

kao ime za našu *sqrt* funkciju.
Simbol a je takođe rezervisan pomoću $label(a, \varepsilon)$ tako da se njegova vrednost može sistematično menjati bez promene značenja izraza.

Na ovoj minimalnoj matematičkoj osnovi se dalje gradi Lisp kao sistem za programiranje. Autor prvo definiše klasu simboličkih izraza (*S-expressions*) u terminima uređenih parova i listi. Potom definiše pet elementarnih funkcija i predikata, koristeći kompoziciju, kondicijalne izraze i rekurzivne definicije gradi obimnu klasu funkcija. Dalje pokazuje kako te funkcije same mogu biti izražene kao simbolički izrazi i definiše univerzalnu funkciju *apply* koja omogućava da se vrši izračunavanje izraza neke funkcije za date argumente. Na kraju još definiše neke funkcije koje uzimaju funkcije za argumente i time kompletira svoju ideju o sistemu.

(a) *Klasa simboličkih izraza*
S-expressions se formira korišćenjem specijalnih karaktera:

$$\begin{array}{c} . \\ (\\ ) \end{array} \qquad (1.8)$$

i beskonačnog seta različitih atomičnih simbola. Originalno se kao simboli koriste velika latinična slova i brojevi koji mogu sadržati razmake, tako su neki od dozvoljenih simbola:

$$\begin{array}{r} A \\ ABA \\ OVO\quad JE\quad VALIDAN\quad LISP\quad SIMBOL \end{array} \qquad (1.9)$$

Simbolički izraz se dalje definiše na sledeći način:

1. Atomični simboli su simbolički izrazi.
2. Ako su e_1 i e_2 simbolički izrazi onda je i $(e_1 \cdot e_2)$ simbolički izraz.

Primeri simboličkih izraza su:

$$\begin{gathered} AB \\ (A \cdot B) \\ ((AB \cdot C) \cdot D) \end{gathered} \tag{1.10}$$

S-expression je jednostavno uređeni par čiji članovi mogu biti atomični simboli i prostiji *S-expressoin*-i. U terminima *S-expression*-a se lista sa proizvoljnim brojem elemenata može formulisati na sledeći način. Lista je:

$$(m_1, m_2, ..., m_n) \tag{1.11}$$

predstavljena kao simbolički izraz:

$$(m_1 \cdot (m_2 \cdot (\cdots (m_n \cdot NIL) \cdots))) \tag{1.12}$$

Gde je NIL atomični simbol korišćen za terminiranje lista. Pošto je mnoge simboličke izraze pogodnije predstavljati kao liste uvodi se i *list* notacija u svrhu skraćenja simboličkih izraza. Tako imamo da:

1. (m) znači $(m \cdot NIL)$.
2. $(m_1, ..., m_n)$ znači $(m_1 \cdot (\cdots (m_n \cdot NIL) \cdots))$.
3. $(m_1, ..., m_n \cdot x)$ znači $(m_1 \cdot (\cdots (m_n \cdot x) \cdots))$.

Podizrazi se skraćuju na isti način. Primer jednog skraćenja je:

$$((A, B), C, D \cdot E) \quad \text{za} \quad ((A \cdot (B \cdot NIL)) \cdot (C \cdot (D \cdot E))) \tag{1.13}$$

(b) *Funkcije simboličkih izraza i izrazi koji ih reprezentuju*
U svrhu predstavljanja funkcija simboličkih izraza koristi se konvencionalna funkcionalna notacija. Kako bi se ovi izrazi razlikovali od samih simbličkih izraza za imena funkcija se koriste sekvence malih latiničnih slova, skupovne zagrade umesto običnih i tačka-zarez umesto zareza, a promenljive funkcija su simbolički izrazi. Tako imamo:

$$\begin{gathered} car[x] \\ car[cons[(A \cdot B); x]] \end{gathered} \tag{1.14}$$

U ovim izrazima nazvanim *M-expressions* (meta izrazi) svi simbolički izrazi se uzimaju doslovno.

(c) *Elementarne S-funkcije i predikati*
Danas se neretko postavlja pitanje:

Koliko primitiva je potrebno za izgradnju LISP mašine?

Na njega nije jednostavno odgovoriti jer su Lisp sistemi po pravilu ekstremno fleksibilni i izvedba najviše zavisi od onoga što se želi postići. U inicijalnom Mekartijevom radu se navodi samo pet funkcija koje će biti ovde navedene dok se 2002. godine Pol Grejm (*Paul Graham*) u svom članku[30] osvrće na Mekartijev rad i uz opasku da je Džon Mekarti učinio za programiranje ono što je Euklid za geometriju navodi „sedam primitivih operatora".

1. *atom.* $atom[x]$ ima vrednost T ili F u zavisnosti od toga da li je x atomični simbol, pa tako:
$$\begin{aligned} atom[X] &= T \\ atom[(X \cdot A)] &= F \end{aligned} \tag{1.15}$$

2. *eq.* $eq[x; y]$ je definisano akko su x i y atomični simboli. $eq[x, y] = T$ ako su x i y isti simboli inače je $eq[x, y] = F$. Tako imamo:
$$\begin{aligned} eq[X; X] &= T \\ eq[X; A] &= F \\ eq[X; (X \cdot A)] &\quad \text{je nedefinisano.} \end{aligned} \tag{1.16}$$

3. *car.* $car[x]$ akko x nije atomični simbol. $car[(e_1 \cdot e_2)] = e_1$, a $car[X]$ je nedefinisano.
$$\begin{aligned} car[(X \cdot A)] &= X \\ car[((X \cdot A) \cdot Y)] &= (X \cdot A) \end{aligned} \tag{1.17}$$

4. *cdr.* $cdr[x]$ je takođe definisano samo akko x nije atomični simbol. Imamo $cdr[(e_1 e_2)] = e_2$ i $cdr[X]$ je nedefinisano.
$$\begin{aligned} cdr[(X \cdot A)] &= A \\ cdr[((X \cdot A) \cdot Y)] &= Y \end{aligned} \tag{1.18}$$

5. *cons.* $cons[x; y]$ je definisano za svako x i y. Imamo da je $[e_1; e_2] = (e_1 \cdot e_2)$. Tako imamo:
$$\begin{aligned} cons[X; A] &= (XA) \\ cons[(X \cdot A); Y] &= ((X \cdot A)Y) \end{aligned} \tag{1.19}$$

Jasno se vidi da *car*, *cdr* i *cons* zadovoljavaju sledeću relaciju:

$$\begin{aligned} car[cons[x;y]] &= x \\ cdr[cons[x;y]] &= y \\ cons[car[x];cdr[x]] &= x, \quad \text{pod pretpostavkom da x nije atomično} \end{aligned} \tag{1.20}$$

(d) *Rekurzivne S-funkcije*
Dobijamo mnogo veću klasu funkcija (zapravo sve funkcije koje se mogu računati) ako dozvolimo da se one formiraju *S-expression-ima* uz korišćenje kondicijalnih izraza i rekurzivnih definicija. Slede neke funkcije koje se mogu definisati na ovaj način.

1. *ff[x].* Vrednost ff$[x]$ je prvi atomični simbol simboličkog izraza ignorišući zagrade. Tako je na primer:

$$\text{ff}[((A \cdot B) \cdot C)] = A \tag{1.21}$$

Ova funkcija se rekurzivno može definisati sledećim izrazom:

$$\text{ff}[x] = [atom[x] \to x; T \to \text{ff}[car[x]]] \tag{1.22}$$

Dakle, koraci u izračunavanju izraza iz 1.21 su sledeći:

$$\begin{aligned} \text{ff}[((A \cdot B) \cdot C)] &= [atom[((A \cdot B) \cdot C)] \to ((A \cdot B) \cdot C); \\ &\quad T \to \text{ff}[car[((A \cdot B) \cdot C)]]] \\ &= [F \to ((A \cdot B) \cdot C); T \to \text{ff}[car[((A \cdot B) \cdot C)]]] \\ &= [T \to \text{ff}[car[((A \cdot B) \cdot C)]]] \\ &= \text{ff}[car[((A \cdot B) \cdot C)]] \\ &= \text{ff}[(A \cdot B)] \\ &= [atom[(A \cdot B)] \to (A \cdot B); T \to \text{ff}[car[(A \cdot B)]]] \\ &= [F \to (A \cdot B); T \to \text{ff}[car[(A \cdot B)]]] \\ &= [T \to \text{ff}[car[(A \cdot B)]]] \\ &= \text{ff}[car[(A \cdot B)]] \\ &= \text{ff}[A] \\ &= [atom[A] \to A; T \to \text{ff}[car[A]]] \\ &= [T \to A; T \to \text{ff}[car[A]]] \\ &= A \end{aligned} \tag{1.23}$$

2. *subst[x; y; z].* Ova funkcija zamenjuje sva pojavljivanja simboličkog izraza x za sva pojavljivanja y u z. Definiše se na sledeći način:

$$\begin{aligned} subst[x;y;z] = [atom[z] \to [eq[z;y] \to x; T \to z]; \\ T \to cons[subst[x;y;car[z]]; subst[x;y;cdr[z]]]] \end{aligned} \tag{1.24}$$

Dakle imamo:

$$subst[(X \cdot A); B; ((A \cdot B) \cdot C)] = ((A \cdot (X \cdot A)) \cdot C) \tag{1.25}$$

3. *equal[x; y].* Vrednost ove funkcije je T ako su x i y isti simbolički izrazi, inače je F. Definiše se kao:

$$\begin{aligned} equal[x;y] = [atom[x] \wedge atom[y] \wedge eq[x;y]] \\ \vee [\neg atom[x] \wedge \neg atom[y] \wedge equal[car[x]; car[y]] \\ \wedge equal[cdr[x]; cdr[y]]] \end{aligned} \tag{1.26}$$

Autor daje i pregled elementarnih funkcija u skraćenom *list* fromatu:

(i) $car[(m1, m2, \cdots, mn)] = m1$

(ii) $cdr[(ms, m2, \cdots, mn)] = (m2, \cdots, mn)$

(iii) $cdr[(m)] = NIL$

(iv) $cons[m1; (m2, \cdots, mn)] = (m1, m2, \cdots, mn)$

(v) $cons[m, NIL] = (m)$

Takođe, pošto se kompozicija *cdr* i *car* koristi relativno često uvode se i sledeća skraćenja:

$$\begin{aligned} cadr[x] \quad \text{za} \quad car[cdr[x]], \\ caddr[x] \quad \text{za} \quad car[cdr[cdr[x]]] \end{aligned} \tag{1.27}$$

Uvodi se još i da važi:

$$[e1; e2; \cdots; en] \quad \text{za} \quad cons[e1; cons[e2; \cdots; cons[en; NIL] \cdots]] \tag{1.28}$$

Kao i jedna funkcija korisna za rad sa listama:

$$null[x] = atom[x] \wedge eq[x; NIL] \tag{1.29}$$

što daje listu $(e1, \cdots, en)$ kao funkciju svojih parametara.
Ovo poslednje skraćenje umnogome je uticalo na sintaktički izgled Lisp-a kao

porodice programskih jezika kakvu poznajemo danas, više od pola veka od kada Mekarti piše njegovu formulaciju.
Sledeće funkcije dobijaju svoju upotrebnu vrednost kada se *S-expression*-i shvate kao liste:

1. *append[x; y]*.
   Formuliše se kao:

   $$append[x; y] = [null[x] \rightarrow y; T \rightarrow cons[car[x]; append[cdr[x]; y]]] \quad (1.30)$$

   A primer upotrebe može biti:

   $$append[(A, B); (C, D, E)] = (A, B, C, D, E) \quad (1.31)$$

2. *among[x; y]*
   Ovo je preditkat koji je T ako se x pojavljuje kao element y, a inače je F, ili uz rekurzivno definisanje funkcije:

   $$among[x; y] = \neg null[y] \wedge [equal[x; car[y]] \vee among[x; cdr[y]]] \quad (1.32)$$

3. *pair[x; y]*.
   Je funkcija koja daje listu parova odgovarajućih elemenata iz lista x i y. Mekarti je originalno definiše kao:

   $$\begin{aligned} pair[x; y] = [&null[x] \wedge null[y] \rightarrow NIL; \\ &\neg atom[x] \wedge \neg atom[y] \rightarrow \\ &cons[list[car[x]; car[y]]; pair[cdr[x]; cdr[y]]] \end{aligned} \quad (1.33)$$

   Kao primer možemo navesti:

   $$pair[(A, B, C); (X, (Y, Z), U)] = ((A, X), (B, (Y, Z)), (C, U)) \quad (1.34)$$

   Koliko je ova funkcija zaista korisna pokazuje činjenica da najveći deo programskih jezika koji se danas koriste kako u akademske svrhe tako i u industriji u svom standardnom skupu alata za rad sa kolekcijama (i sličnim strukturama) sadrže upravo *pair* funkciju. *Haskell*, *Erlang*, *Python*, *Ruby*, *Scala* ... ovu funkciju u svojim standardnim skupovima alata zovu *zip*.

4. *assoc[x; y]*
   Ako je y lista u formatu $((u_1, v_1), \cdots, (u_n, v_n))$ i x je jedno od u, onda je $assoc[x; y]$ njemu odgovarajuće v. Imamo:

$$assoc[x; y] = eq[caar[y]; x] \rightarrow cadar[y]; T \rightarrow assoc[x; cdr[y]]] \tag{1.35}$$

   Dakle:

$$assoc[X; ((W, (A, B)), (X, (C, D)), (Y, (E, F)))] = (C, D) \tag{1.36}$$

   Ova funkcija oslikava način na koji je Mekarti video asocijativne strukture podataka u jednom sistemu za programiranje. Jasno je da je ovo upravo način na koji na asocijativne strukture podataka gleda velika većina programskih jezika koji su danas u široj upotrebi. Notacija za predstavljanje ovih struktura kao *S-expression*-a ima mnogo pogodnosti, o nekima će biti reči dalje u ovom tekstu.

5. *sublis[x; y]*
   Pretpostavlja se da je x u formi liste parova. $((u_1, v_1), \cdots, (u_n, v_n))$ gde su svi u atomični simboli, y mogu biti bilo koji *S-expression*-i. Vrednost $sublis[x; y]$ je rezultat zamene svih u za odgovarajuće v u y. Da bismo definisali *sublis* potrebno je definisati jednu pomoćnu funkciju. Imamo:

$$\begin{aligned} sub2[x; z] = [&null[x] \rightarrow z; eq[caar[x]; z] \rightarrow cadar[x]; \\ &T \rightarrow sub2[cdr[x]; z]] \end{aligned} \tag{1.37}$$

   Pa na osnovu toga možemo napisati:

$$\begin{aligned} sublis[x; y] = [&atom[y] \rightarrow sub2[x; y]; \\ &T \rightarrow cons[sublis[x; car[y]]; sublis[x; cdr[y]]] \end{aligned} \tag{1.38}$$

   Dakle, *sublis* možemo koristiti na sledeći način:

$$sublis[((X, (A, B)), (Y, (B, C))); (A, X \cdot Y)] = (A, (A, B), B, C) \tag{1.39}$$

(e) *Predstavljanje S-funkcija S-izraizima.*
   S-funkcije se opisuju *M-izrazima*. Potrebno je napraviti translaciju *M-izraza* u *S-izraze* kako bi dobili *S-funkcije* sa kojima možemo da vršimo izračunavanja u kojima bi one učestvovale.
   Translacija je određena sledećim pravilima u kojima ε predstavlja *M-izraz* a $\varepsilon*$ njegovu translaciju.

1. Ako je ε *S-izraz* onda je $\varepsilon*$ $(QUOTE, \varepsilon)$.
2. Varijable i imena funkcija koja su pisana malim slovima pišu se velikim slovima. Tako je *car* CAR, *subst* $SUBST$...
3. Forme oblika $f[e_1; \cdots ; e_n]$ se transliraju u forme oblika $(f, e_1 * \cdots , e_n*)$. Tako da imamo $cons[car[x]; cdr[x]]*$ postaje

$$(CONS, (CAR, X), (CDR, X)) \tag{1.40}$$

4. Kondicijalni izrazi oblika $[p_1 \to e_1; \cdots ; p_n \to e_n]*$ se prevode na

$$(COND, (p_1*, e_1*), \cdots , (p_n * \cdot e_n*)) \tag{1.41}$$

5. $\lambda[[x_1; \cdots ; x_n]; \varepsilon]*$ je $(LAMBDA, (x_1*, \cdots , x_n*), \varepsilon*)$.
6. $label[a; \varepsilon]*$ je $(LABEL, a*, \varepsilon*)$.

Prethodno navedena prevođenja čine da *M-izraz*

$$\begin{aligned} &label[subst; \lambda[[x; y; z]; [atom[z] \to [eq[y; z] \to x; T \to z]; \\ &\quad T \to cons[subst[x; y; car[z]]; subst[x; y; cdr[z]]]]]] \end{aligned} \tag{1.42}$$

Može biti predstavljen *S-izrazom*

$$\begin{aligned} &(LABEL, SUBST, (LAMBDA, (X, Y, Z), \\ &(COND((ATOM, Z), (COND, (EQ, Y, Z), X), ((QUOTE, T), Z)), \\ &\qquad ((QUOTE, T), (CONS, (SUBST, X, Y, (CAR, Z)), \\ &\qquad (SUBST, X, Y, (CDR, Z))))))) \end{aligned} \tag{1.43}$$

Ovakav način predstavljanja funkcija sa manjim izmenama koristi se i u današnjim implementacijama Lisp-a kao i u drugim programskim jezicima. Pretpostavljam da je upravo ovakav izgled S-izraza doveo do današnje česte upotrebe „lambda izraza“ kao sinonima za „funkciju u jednoj liniji“ iako, kao što je navedeno, ovaj način predstave funkcija iza sebe ima mnogo moćnije koncepte od kojih će neki biti predstavni u nastavku.

(f) *Univerzalna S-funkcija apply*

Mekarti navodi da postoji *S-funkcija apply* sa osobinom da ako je f *S-izraz* za neku *S-funkciju* f' i $args$ je lista elemenata u formatu $(arg_1, \cdots , arg_n)$ gde su $arg1, \cdots , arg_n$ proizvoljni *S-izrazi* onda su $apply[f; args]$ i $f'[arg_1; \cdots ; arg_n]$

definisane za iste vrednosti $arg_1, \cdots arg_n$ koje su jednake ako su definisane. Na primer:

$$\begin{gathered}\lambda[[x;y];cons[car[x];y]][(A,B);(C,D)] \\ = apply[(LAMBDA,(X,Y),(CONS,(CAR,X),Y));((A,B),(C,D))] \\ = (A,C,D)\end{gathered} \tag{1.44}$$

Autor u svom radu [37] daje motivaciju i obrazloženje za uvođenje *apply* funkcije.

Kao posledicu uvođenja ove funkcije imamo da sada možemo implementirati mašinu — interpreter koji treba da implementira samo funkciju *apply* jer njome može da izvrši bilo koji program izražen u obliku *S-izraza*.

(g) *Funkcije sa funkcijama kao argumentima*

Pošto funkcije možemo da predstavimo kao *S-izraze* sada možemo jednoj *S-funkciji* da dajemo kao argumente druge *S-funkcije*. U originalnom radu iz 1960. godine Mekarti navodi neke od takvih funkcija kao primere upotrebe ovog koncepta.

- *maplist[x, f]* je funkcija koja prima kao prvi argument *S-izraz* x, a kao drugi *S-funkciju* y i definiše se na sledeći način:

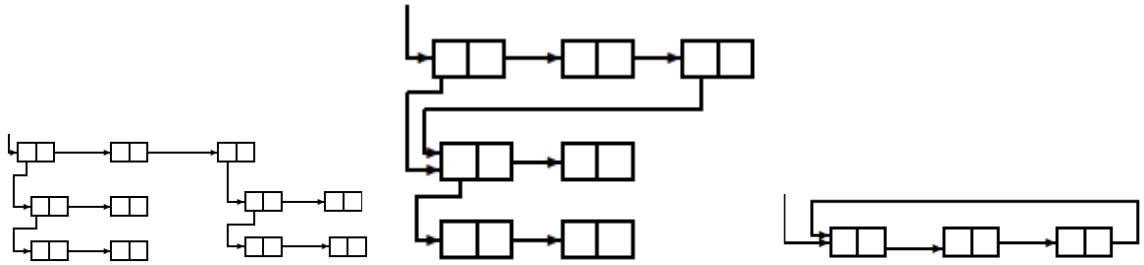
$$maplist[x;f] = [null[x] \rightarrow NIL; T \rightarrow cons[f[x]; maplist[cdr[x];f]]] \tag{1.45}$$

  U kojoj meri se ovaj način razmišljanja pokazao kao ispravan govori i činjenica da danas imamo čitave sisteme za računanje (na jednom procesoru, više njih, ili klasteru računara) koji se zasnivaju na ovom konceptu. Jedan od predstavnika distribuiranih sistema za računanje koji koriste ovaj koncept je i *Apache Hadoop* [2].

- Još jedna funkcija koja je poznata većini programskih jezika koji su u široj upotrebi danas (često kao *find*) je i *search[x; p; f; u]* koja se definiše kao:

$$\begin{gathered}search[x;p;f;u] = [null[x] \rightarrow u; p[x] \rightarrow f[x]; \\ T \rightarrow search[cdr[x];p;f;u]\end{gathered} \tag{1.46}$$

  Dakle, ova funkcija pokušava da pronađe element iz x koji zadovoljava p, ako ga pronađe onda primeni f na njega i vrati dobijenu vrednost, a ako ne vrati u.



(a) Osnovna struktura liste (b) Deljena struktura liste (c) Ciklična struktura liste

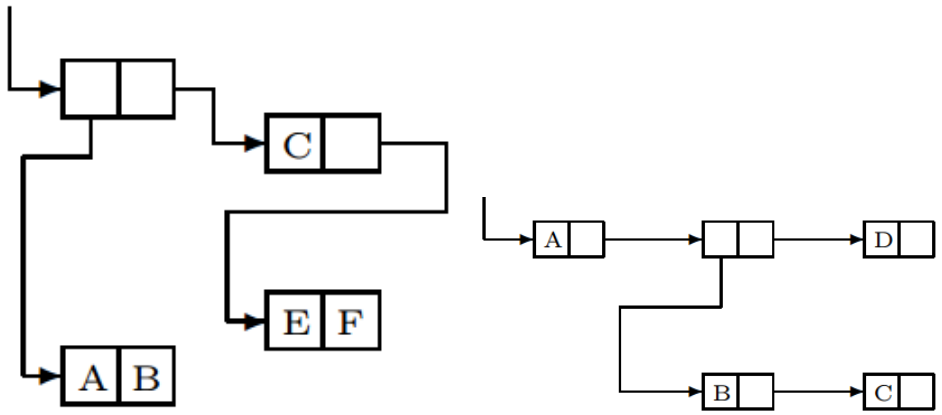
Slika 1.1: Struktura liste

1.2 LISP sistem za programiranje

Sada kada imamo matematičku osnovu koja nam omogućava da sve što želimo da formulišemo možemo da uradimo korišćenjem *S-izraza* (*S-expressions*) postaje jasno da *LISP* kao sistem za programiranje predstavlja sistem za prevođenje *S-izraza* u strukture koje su izvršive na računaru.
Ovo poglavlje će biti orijentisano na način koji je inicijalno primenio Džon Mekarti na računaru *IBM 704* za sistem *Advice Taker*. Kasnije u tekstu biće reči i o drugim načinima da se postigne ista stvar zaključno sa osvrtom na programski jezik *Clojure* [4] i njegovu implementaciju matematičkih osnova koje čine inicijalni Lisp.

(a.) *Predstavljanje S-izraza u strukturi liste*
Struktura liste je kolekcija kompjuterskih reči uređenih kao što je npr. prikazano na slici 1.1(a)

Svaka komjuterska reč je predstavljena podeljenim pravougaonikom. Leva kućica tog pravougaonika predstavlja polje adrese dok desna predstavlja dekrement polje. Strelica od kućice ka drugom pravougaoniku govori da ta kućica sadrži adresu koja odgovara kompjuterskoj reči pravougaonika. Dozvoljeno je da se neka podstruktura pojavljuje na više mesta u strukturi, kao što je na slici 1.1(b), ali nije dozvoljeno postojanje cikličnih struktura, kao na slici 1.1(c). Atomični simboli se predstavljaju posebnim strukturama liste koje se nazivaju *liste asocijacije* simbola. Kućica adrese za prvu reč unutar liste sadrži specijalnu konstantu koja govori da ta reč predstavlja atomični simbol.
S-izraz x koji ne predstavlja atomični simbpol je predstavljen rečju u kojoj prva kućica sadrži lokaciju podizraza $car[x]$ dok druga sadrži $cdr[x]$. Ako koristimo velika slova da obeležimo adrese *lista asocijacije* simbola onda je *S-izraz* $((A \cdot B) \cdot (C \cdot (E \cdot F)))$ predstavljen na slici 1.2(a).



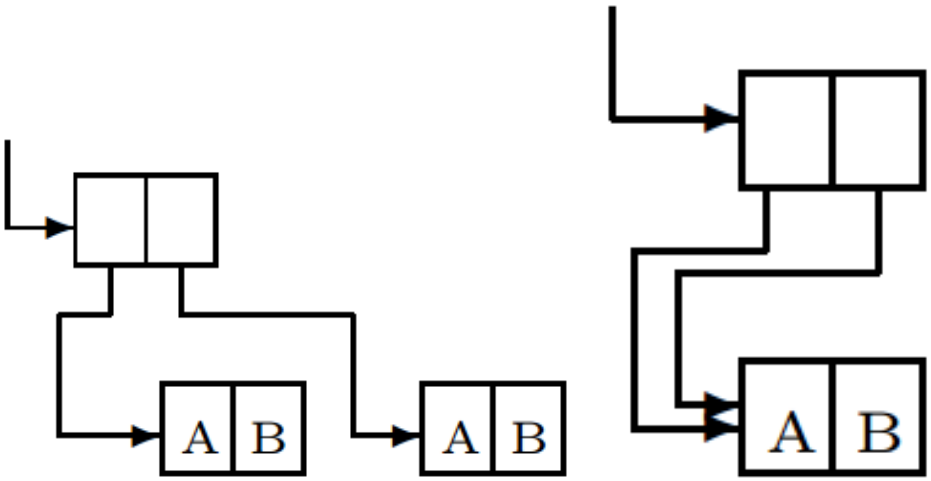
(a) $((A \cdot B) \cdot (C \cdot (E \cdot F)))$ (b) $(A \cdot ((B \cdot (C \cdot NIL)) \cdot (D \cdot NIL)))$

Slika 1.2: Primeri predstave liste

Koristeći *list* oblik *S-izraza* možemo pisati izraz $(A, (B, C), D)$ koji denotira $(A \cdot ((B \cdot (C \cdot NIL)) \cdot (D \cdot NIL)))$ i u LISP sistemu je određen strukturom liste na slici 1.2(b).
Kada strukturu liste posmatramo kao listu koju predstavlja vidimo da svaki njen element zauzima kućicu adrese strukture liste dok njena dekrement kućica pokazuje na sledeći element liste, pri tome poslednji element u svojoj dekrement kućici sadrži NIL.

Naravno, izraz koji sadrži neki podizraz više od jednom može se prestaviti strukturom liste na različite načine. Da li će se podizraz ponavljati ili neće zavisi od toga kako je LISP implementiran i koje su odluke pri tome donesene. Dake, ovo bi mogao biti jedan od momenata divergencije između različitih dijalekata Lisp-a.
Rič Hiki (*Rich Hickey*), kreator programskog jezika *Clojure*, je implementirao neke od svojih ideja oslanjajući se na *S-izraze*. Hikijeve strukture podataka su uz to i *perzistentne* [33], što *Clojure-u* omogućuje veoma „jeftinu“ *imutabilnost* te posledično veoma moćne koncepte konkurentnog programiranja.
Bilo kako bilo, predstava *S-izraza* koji ima ponavljajuće segmente može biti izvedena sa ponavljanjem podstruktura ili bez njega pri čemu to ne utiče na rezultat izvršavanja Lisp programa. Primer dve moguće predstave jednog istog *S-izraza* je $((A \cdot B) \cdot (A \cdot B))$ koji može biti predstavljen kao na slici 1.3(a) ili kao na slici 1.3(b).

U originalnom radu Džon Mekarti navodi nekoliko pogodnosti koje su posledica predstavljanja simboličkih izraza strukturom liste.



(a) Dve predstave ponavljajućeg podizrara (b) Zajednička predstava ponavljajućeg podizraza

Slika 1.3: Različite predstave istog *S-izraza*

1. Dužina ni broj izraza koje će sadržati program ne može se unapred odrediti, stoga je teško odrediti fiksne blokove skladišnog prostora u koje bi bili smešteni.
2. Registri mogu biti vraćeni u *free-storage* listu kada više nisu potrebni. Čak i jedan vraćeni registar je od koristi, ali ako se izrazi čuvaju linearno onda je teško iskoristi blokove različitih dužina koji mogu postati dostupni.
3. Ako želimo efikasnost sa stanovišta iskorišćenog prostora za skladištenje možemo podizrarze koji se pojavljuju u više izraza čuvati samo jednom. To naravno iziskuje i strategiju po kojoj bi se ovaj proces odvijao.

(b.) *Liste asocijacije*[1]

U LISP sistemu za programiranje liste asocijacije simbola koriste se za više stvari nego što zahteva sam matematički model koji je prethodno opisan. Zapravo, svaka informacija koju želimo da asociramo sa nekim simbolom može biti čuvana u okviru liste asocijacije simbola, pa tako može sadržati ime koje se kristi za predstavu simbola, ili broj, *S-izraz* kada simbol na jedan ili drugi način služi kao ime za njega, adresu rutine ako simbol predstavlja ime za rutine mašinskog jezika. To je sve posledica činjenice da mašinski sistem sadrži više primitivnih entiteta nego što je opisano u matematičkom modelu (koji je minimalan).

[1]Kasnije u Common Lisp-u [7], Emacs Lisp-u [11] i drugim dijalektima su nazvane *property lists*.

Kada govorimo o asociranju simbola na druge entitete valja napomenuti i diskusiju:

Lisp-1 protiv Lisp-2.

koja se zapravo svodi na debatu o problemu da li funkcije i ostali entiteti mogu biti označeni istim simbolima istovremeno. Dakle, da li možemo imati simbol A koji ima regularnu vrednost 6 i istovremeno vrednost kao funkciju:

$$\lambda(X,Y)(CONS(CAR(Y),CDR(X))) \tag{1.47}$$

U osnovi, sistemi koji ovakvu postavku smatraju ispravnom nazivaju se *Lisp-2* sistemi i ponašanje implementiraju tako što svaki simbol ima dve kućice (slota) gde jedan predstavlja „regularnu“ vrednost simbola a druga vrednost kao funkciju. Efekat ovakve izvedbe je da zapravo imamo dva prostora imena (*namespace*), jedan za „regularne“ vrednosti, drugi za funkcije. Predstavnici *Lisp-1* grupe dijalekata su *Common Lisp* i *Emacs*.
U *Lisp-1* sistemima imamo da simbol može služiti ili kao funkcija ili kao neka druga „regularna“ vrednost. Nemoguće je da A predstavlja istovremeno obe stvari. Klasičan predstavnik ove grupe dijalekata je *Scheme*[27].
Naravno, oba pristupa imaju kritike, a ovde navodim jednu od njih:

> ***Ako je jedna od poenti funkcionalnog programiranja tretirati funkcije kao prvoklasne vrednosti — first class values onda je konceptualno mnogo čistije tretirati ih kao i sve druge entitete.***

Svakako da ova primedba ne stoji ako dijalekat nema tendenciju da insistira na funkcionalnoj paradigmi kao što je slučaj sa *Common Lispom* koji poseduje i sistem za objektno programiranje *CLOS*[35]. Značajan osvrt na ovu diskusiju daje Kent Pitmen u svom radu *„Technical Issues of Separation in Function Cells and Value Cells“*[38].

(c.) *Liste slobodnog prostora*
U svakom momentu samo je deo memorije rezervisan za čuvanje strukture liste zaista zauzet čuvanjem *S-izraza*. Svi ostali registri se čuvaju u listi koja se zove *Lista slobodnog prostora* (*free-storage list*). Poseban registar *FREE* je namenjen za čuvanje adrese prve reči u ovoj listi. Kada je potrebno rezervisati

novi prostor za smeštanje strukture onda se zauzima prva reč iz pomenute liste a vrednost *FREE* registra se menja tako da pokazuje na prvu sledeću slobodnu reč. Onaj koji piše program ne vodi računa o vraćanju slobodnih registara u *listu slobodnog prostora* već to čini sitem po internom automatizmu.

Ovde će biti opisana logika tog automatizma kako je vidi Džon Mekarti u svom radu iz 1960. godine. Već tada autor ovaj mehanizam naziva *„garbage collection“* i uviđa njegove prednosti i mane. Najveći deo jezika visokog nivoa apstrakcije koji se danas aktivno koriste imaju *„garbage collection“* proces koji funkcioniše na ovom principu.

Postoji fiksni broj baznih registara koji čuvaju lokacije struktura lista koje su dostupne programu. Naravno, zbog raznolikosti struktura lista, o kojoj je ranije bilo reči, arbitraran broj registara može biti uključen u rad programa. Svaki registar koji je dostupan programu je dostupan jer ga je moguće doseći iz jednog ili više osnovnih registara ulančavanjem *CAR* i *CDR* operacija. Kada se sadržaj nekog baznog registra izmeni, može se desiti da registar na koji je bazni registar ranije pokazivao više ne bude dostupan ulančavanjem *CAR* i *CDR* iz bilo kog drugog baznog registra. Takav registar može se smatrati *napuštenim* sa stanovišta programa jer sadržaj tog registra postaje nedostupan za svaki mogući program — znači da njegov sadržaj više nije od interesa i da bi smo želeli da ga ponovo imamo u *listi slobodnog prostora*. To se dešava na sledeći način.

Ništa se ne dešava dok program ne dođe do momenta kada nema više slobodne memorije. Kada je potreban slobodan registar i nema više preostalih u *listi slobodnog prostora* tada se dešava *garbage collection* proces.
Prvo se pronađu svi registri koji su dostupni iz baznih registara i oni se obeleže negativnim brojem. To se postiže tako što se krene od prvog baznog registra i od njega se ulančavanjem *CAR* i *CDR* operacija dolazi do svakog povezanog registra obeležavajući ga negativnim brojem. Kada se završe svi iz prvog ide se na sledeći i tako redom. Kada opisani proces naiđe na neki registar koji je već označen negativnim brojem on pretpostavlja da ga je već jednom obišao.

Nakon što proces obiđe sve bazne registre i promeni im obeležja pristupa zoni memorije u kojoj se čuvaju strukture liste i stavlja sve registre čija obeležja nisu promenjena u prethodnom koraku ponovo u *listu slobodnog prostora* postavljajući obeležja dostupnih registara ponovo na pozitivne brojeve.

Mekarti na kraju iznosi i jasan zaključak da pošto je proces u potpunosti automatski to čini sistem veoma prijatnim okruženjem za programera dok računaru

nameće dodatni posao. Navodi takođe da *garbage collection* proces ima smisla samo u situacijama kada je slobodna memorija značajno veća od memorije zauzete strukturama liste jer sam proces troši određeno vreme računara za obilazak svih registara. Zato proces nema pozitivan efekat ako nakon njegovog izvršenja programer neće na raspolaganju imati značajno veću količinu memorije nego pre njegovog pokretanja.

(d.) *Izvedba elementarnih S-funkcija u računaru*
S obzirom na vrlo moćan matematički model Lisp-a trivijalno je zaključiti kako se implementiraju elementarne *S-funkcije*. Tako se na primer $cdr[e; f]$ može implementirati tako što se radi numeričko poređenje adresa reči koje sadrže simbole e i f. Ovo funkcioniše kada svaki atomički simbol ima samo jednu *listu asocijacije* u kojoj se definiše. $cdr[e]$ se može implementirati prostim uzimanjem dekrement polja odgovarajuće reči. $cons[e; f]$ popunjava memorijski prostor iz liste slobodnog prostora odgovarajućim sadržajem iz svojih argumenata itd.

Velika ideja je u tome što se sve funkcije mogu predstaviti komponovanjem elementarnih *S-funkcija* što implementaciju čitavog sistema svodi na proste koncepte.

(e.) *Reprezentacija S-funkcija od strane programa*
Kompajliranje funkcija koje su kompozicije CAR, CDR i $CONS$ je podjednako trivijalno kroz kompajler koliko i ručno (matematičkim izvođenjem). Kondicijalni izrazi takođe ne predstavljaju problem, osim što je potrebno izvršiti takvo kompajliranje da samo oni p i e koji su neophodni budu izračunavani. Ono gde se stvar malo komplikuje, kako navodi Mekarti, je kompajliranje rekurzivnih funkcija.

Ono što čini ovaj deo kompajliranja malo komplikovanijim je činjenica da kod rekurzivnih funkcija imamo da rezultujuća rutina funkcije sadrži samu sebe kao subrutinu. Na primer, program za $substr[x; y; z]$ koristi samog sebe kao subrutinu da izračuna zamene za podizrazre $car[z]$ i $cdr[z]$. Dok se izračunava $substr[x; y; cdr[z]]$ rezultat predhodnog izračunavanja $substr[x; y; car[z]]$ mora biti sačuvan u privremenom registru. Sada $substr[x; y; cdr[z]]$ pri svom izračunavanju može potraživati taj isti registar da sačuva privremenu vrednost što dovodi do mogućeg konflikta.

Način na koji Mekarti u svom originalnom radu rešava ovaj problem je itekako vredan pomena jer predstavlja još jedan od koncepata koje danas pri pisanju programa uzimamo za aksiom.

Naime, autor navodi da konflikt može biti razrešen korišćenjem *SAVE* i *UNSAVE* rutina koje koriste javnu *push-down* listu. *SAVE* rutina se pokreće na početku rutine za rekurzivnu funkciju sa zadatkom da sačuva dati set uzastopnih registara i ima indeks koji joj govori koliko registara *push-down* liste je trenutno u upotrebi. Ona stavlja sadržaje registara koje treba da sačuva u prve slobodne registre *push-down* liste, uvećava svoj indeks, i vraća program na mesto sa koga je pozvana. Program od tog momenta može da koristi rezervisani deo *push-down* liste kao svoju privremenu memoriju. Pre nego što se rutina završi poziva se *UNSAVE* koja uzima nazad sadržaj privremene memorije sa *push-down* liste i vraća indeks. Dakle, kažemo da je rekurzivna subrutina transparentna na registrima za privremeno čuvanje.

Kasnije će *push-down* lista biti nazvana ***„stack“*** a rutine *SAVE* i *UNSAVE* ***PUSH*** i ***POP***, respektivno.

1.3 Pregled koncepata koje donosi Lisp

U ovom poglavlju biće dat kratak pregled nekih krupnih koncepata koje Lisp formuliše, koji su danas potpuno prihvaćene u velikoj većini programskih okruženja, a koji ranije nisu bili poznati u obliku u kakvom su danas u upotrebi. 1950-tih godina kada je Mekarti dizajnirao Lisp njegove ideje su predstavljale radikalno razmimoilaženje od načina dotadašnjeg razmišljanja o kompjuterskom programu i programiranju pa tako i od programskih jezika od kojih je najznačajniji bio *Fortran*[12]. Pol Grejm u svom članku *„What Made Lisp Different“*[29] iz 2001. godine navodi devet značajnih ideja koje Lisp uspostavlja.

1. ***Kondicijalni izrazi***
   Kondicijalni izraz u formi *if-then-else* je nešto što danas svi uzimamo zdravo za gotovo. Da li bi vas iznenadilo da sistem za programiranje koji je pred Vama nema ovakve konstrukcije? Njih je uveo Mekarti na kursu razvoja Lisp-a. *Fortran* u tom momentu ima samo kondicijalnu *goto* naredbu, čvrsto vezanu za odgovarajuću instrukciju na hardveru na kome se izvršava. Mekarti, koji je se između ostalog bavio i *Algol*[1] programskim jezikom, ga je uveo u *Algol* odakle se dalje širio.

2. ***Tip funkcije***

   U Lisp-u su, kako je prethodno navedeno, funkcije objekcti prve klase (*first class objects*) — one su tim podataka, baš kao što su *integer*, *string* i sl. Funkcije imaju literalnu reprezentaciju, mogu biti čuvane u promenljivim, mogu biti prosleđivane kao argumenti... Ovo je osobina jezika koja obezbeđuje potpuno nov pogled na dekompoziciju problema i danas se navodi kao snaga programskih jezika u kojima je prisutna, takvi su „Haskell“, „Python“, „Ruby“, „JavaScript“ i mnogi drugi. Neki programski jezici koji se nazivaju „jezicima visokog nivoa“ ni danas nemaju *first class functions*.

3. ***Rekurzija***

   Rekurzija kao matematički koncept naravno postoji i pre Lisp-a ali Lisp je prvi programski jezik koji je podržava (može se diskutovati o tome da li je ova osobine implicitno već sadržana u prethodnoj).

4. ***Nov koncept varijabli***

   U Lisp-u su varijable efektivno pokazivači. Vrednosti su te koje poseduju tipove, ne same varijable. Dodela ili uvezivanje (*binding*) znači prosto kopiranje pokazivača a ne onoga na šta oni pokazuju.

5. ***Garbage collection***

   U prethodnom poglavlju je opisano na koji način funkcioniše u Lisp-u, uticaj Lispovog sistema na one danas aktuelne je očigledan.

6. ***Program sačinjen od izraza***

   U Lisp-u su programi stabla izraza, od kojih svaki vraća vrednost (s tim da kod nekih implementacija može da vrati i više vrednosti). To je u suprotnosti sa *Fortranom* i ostalim jezicima koji iz njega proističu — oni prave razliku između ***izraza*** (*expression*) i ***naredbe*** (*statement*).

   U *Fortranu* je bilo prirodno praviti navedenu razliku jer je on bio *line-oriented*, što ne iznenađuje ako znamo da mu je format unosa bio — *bušene kartice*. Nije bilo ugnježdavanja naredbi. Imalo je smisla praviti samo matematičke izraze i oni su jednini vraćali vrednosti jer druge vrednosti ne bi imalo šta da prihvati. Ovo ograničenje je nestalo sa dolaskom blok-struktuiranih jezika, ali tada je već bilo kasno, razlika između izraza i naredbe se zadržala proširivši se sa *Fortrana* na *Algol* te tako i na sve njegove naslednike među kojima i *C*.
   Kada je jezik u potpunosti sastavljen od izraza to znači da možete sastavljati

izraze kakogod želite. Na primer u *Clojure* programskom jeziku možete da napišete:

```
(if foo (= x 1) (= x 2))
;; ili
(= x (if foo 1 2))
;; ili
(->> (if foo 1 2)
     (= x))
```

7. ***Tip simbola***
   U Lisp-u se simboli razlikuju od stringova u smislu da je moguće testirati simbole na jednakost pri čemu se dešava poređenje referenci.

8. ***Notacija za kod***
   Lisp notacija je stablo simbola.

9. ***Ceo jezik je uvek dostupan***
   Nema prave razlike između čitanja (*read-time*), kompajliranja (*compile-time*) i izvršavanja (*runtime*). Možete kompajlirati ili izvršavati kod pri čitanju. Možete čitati kod ili ga izvršavati pri kompajliranju, a možtete i čitati ili kompajlirati dok se program izvršava.

   Izvršavanje za vreme čitanja koda omogućuje da programer reprogramira sintaksu Lisp-a. Izvršavanje koda pri kompajliranju je osnova makro sistema (o čemu će kasnije biti više reči). Kompajliranje za vreme izvršavanje je ono što omogućuje Lisp-u da bude jezik za ekstenzije u sistemima kakav je *GNU Emacs*[11]. Čitanje koda za vreme izvršavanja programa je „već gotovo“ rešenje za komunikaciju preko *S-izraza* — ideja koja je kasnije „ponovo smišljena“ kao *XML*[24], a aktuelna je i danas kroz *EDN*[31] format.

U vreme kada je Lisp formulisan sve ove ideje bile su sklonjene u stranu u okviru programerske prakse kao posledica nemogućnosti hardvera iz 1950-tih godina da podrži nove koncepte.

Vremenom, kako je hardver postajao moćniji tako su i popularni programski jezici polako usvajali ideje Lisp-a. Osobine od 1 do 5 su danas već standard u većini njih, šesta osobina je takođe usvojena u nekima. *Python* na primer podržava i 7. Danas postaju popularni jezici koji pružaju još 8 i 9 (primer je *Elixir*[9]) pa se postavlja pitanje:

Ako neki programski jezik u potpunosti zadovoljava sve ideje Lisp-a, da li je on tada novi programski jezik ili dijalekat Lisp-a?

1.4 Funkcionalno programiranje

Kako softver biva kompleksniji postaje sve važnije struktuirati ga na pravi način. Dobro struktuiran softver je lakši za pisanje, za razumevanje i pronalaženje nedostataka, pruža kolekciju modula koji mogu biti ponovo korišćeni redukujući buduću cenu razvoja. Konvencionalni programski jezici koji naglašavaju proceduralno programiranje imaju očigledne limita sa stanovišta modularizacije, dok oni koji naglašavaju objektno orijentisan način programiranja struktuiranje rešenja nameću paradigmom. U funkcionalnom programiranju alati koji pomažu u uspostavljanju modulariteta leže u drugačijim gradivnim (logičkim) konceptima u odnosu na objektno programiranje.

Postoji čitav spektar programskih jezika u kojima se na više ili manje efektan način mogu demonstrirati koncepti funkcionalnog programiranja. Među njima se svakako ističu *Haskell*, *Erlang*, *Clojure*, a od skora i *Elm*[10] — zanimljivo programsko okruženje koje se izvršava na *JavaScript engine-u*.

U ovom poglavlju će ilustracije koncepata biti date u *Clojure-u* kako bi se izgradila osnova za razumevanje motivacije koja je dovela do skupa ideja izloženih u ovom radu.

1.4.1 Čiste funkcije (*pure functions*)

Funkcija je čista ako zadovoljava sledeće kriterijume:

- Za iste argumente uvek daje isti rezultat. Ova osobina se u teoriji naziva *referencijalna transparentnost* (*referential transparency*).
  Referencijalna transparentnost se u teoriji definiše na domenu širem od same funkcije (u prethodnom poglavlju je objašnjeno u kojoj meri su u Lisp-u srodni *S-izrazi* i *S-funkcije*). Kažemo da je izraz *referencijalno transparentan* kada sve njegove moguće instance proizvode istu vrednost kada se izračunaju — znači da neka instanca izraza uvek u programu može biti zamenjena svojom vrednosti bez da se ponašanje programa pritom promeni. Izraz koji nije referencijalno transparentan naziva se *referencijalno netransparentnim* (*referential opaque*). U matematici su sve funkcije uvek *referencijalno transparentne* ali u programiranju to nije uvek slučaj.

  Referencijalno transparentni programi dobijaju na korektnosti, pojednostavljenju algoritma, olakšavanju refaktorizacije (reorganizacije) koda i mogućnostima

za optimizaciju. Moguće je optimizovati kod korišćenjem *memoizacije*, eliminacije čestih podizraza (što je posebno jednostavno kod Lisp-a), lenje evaluacije (*lazy evaluation*) i paralelizacije.

- Ne može izazivati nikakve sporedne efekte. To znači da funkcija ne sme izazivati bilo kakve promene koje su vidljive van nje same — na primer ne sme menjati spolja dostupan promenljiv (*mutable*) objekat pa čak ni pisati u fajl, tok (*stream*), red (*queue*) niti bilo šta drugo spolja dostupno.

Gorenavedeni kvaliteti nam omogućuju da lakše zaključujemo o programu zato što su funkcije u potpunosti izolovane jedne od drugih i od okruženja pa nije moguć uticaj akcija koje se izvršavaju unutar funkcije na okruženje. Kada radite sa čistim funkcijama ne morate da razmišljate o tome šta sve možete „pokvariti“ menjajući neke delove funkcije. Smanjuje se mogućnost da popravljajući jednu osobinu koja ne radi pokvarite neku drugu koja je do tada radila. Funkcije koje su čiste su takođe i konzistentne i nikada se ne postavlja pitanje: „Zašto sam dobio drugačiji izlaz kada sam prosledio iste argumente?“ — jer se to nikako ne može desiti.

Potreba za čistim funkcijama rođena je iz težnje ka stabilnosti aritmetičkih funkcija. Aritmetičke funkcije su uvek stabilne i prirodno je da se ne postavlja pitanje: „Šta će se desiti ako saberem 2 i 2?“, već one upravo predstavljaju one gradivne delove sistema na koje se uvek možemo osloniti. Ako postignemo da funkcije koje programiramo budu čiste koliko aritmetičke onda funkcionalnosti našeg programa postaju te čvrste gradivne čestice nad kojima gradimo stabilan sistem. To smanjuje broj „nestabilnih delova“ (*moving parts*) sistema i time daje programeru veći kapacitet da im se posveti.
Funkcije koje svoj rezultat grade samo na nepromenljivim (*immutable*) vrednostima su *referencijalno transparentne*. Na primer, pošto je string *„ sam u nju pada.“* nepromenljiv onda je i funkcija *izreka referencijalno transparentna*:

```
(defn izreka
  [pocetak]
  (str pocetak " sam u nju pada."))

(izreka "Ko drugome jamu kopa")
;; => Ko drugome jamu kopa sam u nju pada.
```

Naredna funkcija ne daje uvek isti rezultat za iste argumente i stoga nije *referencijalno transparentna*. Svaka funkcija koja zavisi od nekog pseudoslučajnog generatora

je *referencijalno netransparentna*, i svaka funkcija koja zavisi od barem jedne *referencijalno netransparentna* funkcije je takođe *referencijalno netransparentna*.

```
(defn da-li-da-jedem?
  []
  (if (> (rand) 0.5)
    "Snaga na usta ulazi!"
    "Svako jelo svoga gazdu klelo, sto ne legne da se slegne."))
```

Nije moguće sve potrebne akcije u jednom programu uraditi koristeći samo *referencijalno transparentne* funkcije jer je često potrebno npr. čitati neke podatke sa nekog spoljašnjeg medija. Tako na primer funkcija *analiziraj-fajl* nije *referencijalno transparentna* dok *analiziraj* sama po sebi jeste.

```
(defn analiziraj
  [tekst]
  (str "Broj karaktera:" (count tekst)))

(defn analiziraj-fajl
  [fajl]
  (analiziraj (slurp fajl)))
```

Kada koristite *referencijalno transparentne* funkcije nikada ne morate da razmišljate o mogućim spoljašnjim uslovima koji mogu uticati na vašu funkciju. Ovo je posebno bitno kada se funkcija koristi na više mesta u programu ili je duboko ugnježđena u lancu poziva funkcija, u svakom od ovih slučaja znate da vaša funkcija „uvek radi isto“.

Čiste funkcije takođe ne smeju imate sporednih efekata. Napraviti sporedni efekat znači promeniti asocijaciju između imena i njegove vrednosti u okviru nekog opsega. Ovo je vrlo česta praksa kako u proceduralnim programskim jezicima tako i u objektno orijentisanim. Čak i u nekim koji u određenoj meri podržavaju funkcionalno programiranje (kao što je *Scala*) neretko se u produkcionom kodu nailazi na mnoštvo sporednih efekata. Radi ilustracije sledi primer u *JavaScript-u*:

```
var promenljivObjekat = {
  stanje: "Jedno stanje."
};
```

```
var rizicnaFunkcija = function(objekat){
  objekat.stanje = "Drugo stanje.";
}

rizicnaFunkcija(promenljivObjekat);
promenljivObjekat.stanje;
// => "Drugo stanje."
```

Naravno, program mora imati neke *sporedne efekte*. Programi obično pišu na disk, to menja asocijaciju između imena fajla i kolekcije sektora na disku; da bi prikazao nešto na monitoru mora promeniti *RGB* vrednosti piksela monitora itd. U suprotnom ne bi imalo smisla pokretati ga. Program koji ne bi imao spoljašnje ulaze i izlaze ne bi imao nikakvu komunikaciju sa ostatkom sistema i samo bi trošio računarsko vreme na svoje izvršavanje. Ovo svakako ne znači da zbog navedenog program treba da ima *sporedne efekte* „svuda unaokolo“. Dakle, cilj dobro struktuiranog programa napisanog u duhu funkcionalnog programiranja nije da u bukvalnom smislu otkloni *sporedne efekte* već da ih koristi na minimalan broj mesta — da ih ne koristi tamo gde ne postoji opravdanje za njih, da im pristupa na odgovorniji način.

Sporedni efekti su potencijalno štetni, oni dovode do situacije u kojoj ne znate na šta referiše neko ime u programu. To dovodi do mesta na kome ne možete da zaključite zašto je određeno ime asocirano sa nekom vrednošću i kako je do toga uopšte došlo. Ova situacija dovodi do prave konfuzije pri pokušaju otklanjanja primećene greške, i ne samo to već je i svaka promena koju pri tome pravite potencijalno štetna za funkcionalnosti koje u tom trenutku rade na očekivan način. Kada imate pred sobom čiste funkcije onda je jedino što treba da razmatrate veza između ulaza i izlaza iz neke funkcije.

U teoriji objektno orijentisanog dizajna mnogo je napisano o tome kako se upravlja stanjem programa. Postoje različite strategije koje na ovaj ili onaj način pružaju mogućnost da iskontrolišete sve te promene stanja koja nastaju izvršavanjem programa kojem je prirodno da menja stanja promenljivih (*mutable*) entiteta. Funkcionalno programiranje sa druge strane pruža strategije i mehanizme koji omogućuju da u maksimalnoj meri izbegnete promene stanja sistema.

Programski jezik *Clojure* čini mnogo na tome da korisniku olakša izbegavanje *sporednih efekata*, to između ostalog postiže i time što su sve njegove bazične strukture nepromenljive (*immutable*). U *Clojure-u* je nemoguće npr. izmeniti neku kolekciju, ne možete joj dodati element, obrisati postojeći ili ga izmeniti. Programerima koji se

ranije nisu susretali sa sličnim okruženjima za programiranje ovo na prvi mah može zvučati čudno. Međutim, svi funkcionalni programski jezici pružaju veoma bogat skup alata za rad sa nepromenljivim strukturama i ako usvojite odgovarajući način razmišljanja sve standardne zadatke možete izvršavati sa najmanje istom lakoćom kao i u okruženjima koja naglašavaju mutabilnost (*mutability*).

1.4.2 Rad sa nepromenljivim(*immutable*) strukturama podataka

Nepromenljive strukture podataka obezbeđuju da program nema sporedne efekte. Ovde će biti navedeni neki primeri koji pokazuju kako se postižu efekti nekih standardnih šabloni iz proceduralnog (i objektno orijentisanog) programiranja korišćenjem principa funkcionalnog programiranja u *Clojure-u*.

- ***Rekurzija umesto for/while petlji***
  Pogledajmo jedan čest šablon koji se koristi u *JavaScript-u*:

```
var sviPacijenti = getSviPacijentiKlinike();
var pregledaniPacijenti = [];
var l = sviPacijenti.length;

for(var i=0; i < l; i++) {
    if(sviPacijenti[i].analiziran){
        pregledaniPacijenti.push(sviPacijenti[i]);
    }
}
```

  Možemo primetiti da ovaj kod pravi sporedan efekat na „sviPacijenti“. Pravljenje sporednih efekata na ovaj način — mutiranjem (izmenom) unutrašnjih varijabli — je prilično štetan. Pravi se nova promenljiva umesto da se koristi ona koju smo već prethodno dobavili negde iz programa (u našem slučaju povratnu vrednost „getSviPacijentiKlinike()“). Ovaj deo problema se jednostavno može izbeći i u „JavaScript-u“, ali nam svakako ostaje onaj drugi deo problema — mutiranje dobijene vrednosti.

  Clojure nepromenljive strukture podataka čak ni ne daju mogućnost da uradite ovakve štetne mutacije. *Clojure* nema ni operator dodele, pa ne možete ni da asocirate novu vrednost sa imenom bez uvođenja novog opsega (*scope*):

```
(def ime "jedino ime")

(let [ime "jedino ime u ovom opsegu"]
  ime)
;; => jedino ime u ovom opsegu

ime ; => jedino ime
```

U primeru gore prvo se vezuje (*bind*) ime „ime“ na vrednost „jedino ime“ u globalnom opsegu. Onda se uvodi novi opseg korišćenjem konstrukcije „let“. U tom opsegu se ime „ime“ vezuje na „jedino ime u ovom opsegu“. Kada opseg uveden sa „let“ biva napušten, ponovo se vraća globalni opseg u kome se „ime“ evaluira (izračunava) u „jedino ime“.

– Mehanizam koji je ovde korišćen u teoriji se naziva *leksičko vezivanje* (***lexical binding***).
Vezivanje (***binding***) je veza korespondencije između imana i onoga što ono predstavlja (obično memorijske lokacije).
Leksičko i *dinamičko* vezivanje (***dynamic binding***) odnose se na način na koji se traže varijable po imenu. *Leksičko* se često naziva i *statičko* vezivanje.

Kod dinamičkog vezivanja postiže se efekat kao da se sva imena nalaze u jednoj globalnoj tabeli. Kod leksičkog se pak postiže efekat kao da se za svaki opseg formira nova tabela koja povezuje imena i vrednosti pri čemu je sve to organizovano u strukturu koja se naziva *okruženje* (*the environment*).

Radi ilustracije razlike između ova dva koncepta biće korišćen kod u Emacs Lisp-u.

```
(let ((a 1))                              ; vezivanje (1)
  (let ((f (lambda () (print a))))
    (let ((a 2))                          ;vezivanje (2)
      (funcall f))))
```

Imena koja su leksički vezana se traže samo među vezivanjima unutar leksičkog okruženja imena. U Lisp-u je leksičko okruženje eksplicitno u kodu i predstavlja „let“ *S-izraz* u okviru koga se nalazi ime. Dakle, ako

je „a“ leksički vezano onda će kod ispisati „1“, jer je samo vezivanje (1) unutar leksičkog okruženja od „a“.

Imena koja se dinamički vezuju se traže samo među vezivanjima iz dinamičkog okruženja imena. Dakle, iz skupa svih vezivanja koja su se desila u programu do trenutnog momenta, a koja nisu uništena. Ako je „a“ dinamički vezano kod iznad će ispisati „2“ zato što su se oba vezivanja desila do momenta kada je „a“ zatraženo ali se vezivanje (2) desilo najskorije pa će njegov efekat biti uzet u obzir.

Dinamičko vezivanje dovodi do štetnih mutacija o kojima govorimo.

Clojure, istini za volju, ima mehanizam kojim se može postići dinamičko vezivanje ali je potrebno to eksplicitno navesti i koristiti nestandardne načini vezivanja varijabli. Naime, dinamička varijabla mora sadržati „dynamic“ u svojim meta podacima i uz to je potrebno koristiti *S-izraz* „binding“ umesto „let“. U svakom slučaju, moguće je, ako programer zna šta radi i ako ima pravo obrazloženje za to.

Umesto štetnog mutiranja funkcionalni programski jezici vam pružaju efikasan mehanizam rekurzije[2] kao alternativu za mutaciju. Sledeći primer oslikava generalan način za rešavanje problema korišćenjem rekurzije.

```
(defn suma
  ([brojevi] (suma brojevi 0))
  ([brojevi akumuliran-zbir]
   (if (empty? brojevi)
     akumuliran-zbir
     (suma (rest brojevi) (+ (first brojevi) akumuliran-zbir)))))
```

Ova funkcija prima dva argumenta, kolekciju koju obrađuje („brojevi“) i akumulator („akumuliran-zbir“). Koristi preklapanje arnosti da bi uspostavila 0 kao početnu vrednost za „akumulirani-zbir“.

Kao i većina rešenja rekurzijom ova funkcija ima sledeću strukturu; proverava kolekciju koju obrađuje na osnovni uslov, ako ustanovi da je cela kolekcija obrađena onda vraća akumulator, a ako ne onda poziva samu sebe prosleđujući kolekciju bez elementa koji je upravo obrađen i sa promenjenim akumulatorom tako da se obrađeni element uzima u obzir. Taj ostatak kolekcije se u teoriji naziva *rep* (***tail***). Pozivi funkcije „suma“ izgledaju ovako, redom:

[2]U poglavlju 1.2 je opisana osnova mehanizma rekurzije u Lisp-u.

```
(suma [39 5 1]) ; pocetni poziv sa arnosti 1,
                ; 0 se uzima za akumuliran-zbir
(suma [39 5 1] 0)
(suma [5 1] 39)
(suma [1] 44)
(suma [] 45)   ; osnovni uslov je zadovoljen, vraca se akumulator
```

Svaki rekurzivni poziv formira novi opseg u kome „brojevi“ i „akumuliran-zbir“ imaju nove vrednosti, bez potrebe da se menjaju originalne vrednosti koje su prosleđene funkciji i bez potrebe za bilo kakvom internom mutacijom.

Zapravo, u *Clojure-u* je umesto samog imena funkcije pri rekurziji bolje koristiti „recur“ konstrukciju, zbog boljih performansi.

Clojure ne pruža *tail call* optimizaciju (*TCO*) po automatizmu već ima eksplicitnu konstrukciju za tu funkcionalnost — „recur“, koja je implementirana na nivou *Clojure* kompajlera i moguća je samo za *self-calls* (kada funkcija na kraju — repu poziva samu sebe), ne i za *tail-call* na druge funkcije — zbog ograničenja *JVM-a*[3] na kome se *Clojure* izvršava. „recur“ ponovo koristi jedan *stack frame* i time omogućuje brže izvršavanje i izbegava prelivanje steka. Ova konstrukcija je moguća samo kao poslednja stvar (*tail* pozicija) u nekom pozivu i uvek se odnosi na poslednju tačku rekurzije (*recursion point*). Tačku rekurzije može uvesti neka od formi za definisanje funkcije („fn“, „letfn“, „defn“ ...) ili „loop“ forma.

Evo primera istog koda sa korišćenjem „recur“ konstrukcije:

```
(defn suma
  ([brojevi] (suma brojevi 0))
  ([brojevi akumuliran-zbir]
   (if (empty? brojevi)
     akumuliran-zbir
     (recur (rest brojevi) (+ (first brojevi) akumuliran-zbir)))))
```

Ako se obrađuju relativno male kolekcije onda korišćenje „recur“ konstrukcije i nije obavezno, ali ako se radi o milionima elemenata kolekcije onda je potrebno koristiti ovu konstrukciju kako ne bi došlo do prelivanja steka.

[3]Neke od razloga navodi Brajan Gec, *Java Language Architect* iz *Oracle-a*, u jednom od svojih izlaganja[28].

Sada se postavlja legitimno pitanje — Ako pravimo toliko međuvrednosti zar to nije opterećenje za *garbage collector*? Odgovor je — ne. Razlog je što su *Clojure* nepromenljive strukture podataka „ispod haube“ implementirane korišćenjem deljenja strukture (*structural sharing*), o čemu će biti više reči u poglavlju 1.5.1.

- ***Kompozicija funkcija umesto mutiranja atributa***
  Drugi zadatak za čije se rešavanje često koristi mutacija je kada želimo da izgradimo finalno stanje nekog objekta. Evo jednog primera iz *Python* programskog jezika:

```
class Prodavnica:
    def __init__(self, broj_klijenata):
        self.broj_klijenata = broj_klijenata
        self.koriguj_broj_klijenata()

    def koriguj_broj_klijenata(self):
        self.broj_klijenata = int(self.broj_klijenata)
        self.broj_klijenata += 10;

prodavnica = Prodavnica(133.22)
prodavnica.broj_klijenata # => 143
```

Ovde „Prodavnica“ enkapsulira znanje o tome kako se koriguje „boroj_klijenata“. Pri konstruisanju objekta, prodavnici se dodeljuje broj klijenata koji se dalju u njegovom životnom ciklusu progresivno mutira u svakom momentu oslikavajući trenutno stanje prodavnice. Evo kako bismo to uradili u *Clojure-u*:

```
(defn koriguj-broj-klijenata
  [prodavnica]
  (+ 10 (int (:broj-klijenata prodavnica))))

(koriguj-broj-klijenata {:broj-klijenata 133.22}) ; => 143
```

Umesto progresivnog mutiranja objekta, funkcija „koriguj-broj-klijenata“ funkcioniše tako što prosleđuje nepromenljivu mapu koja ima ključ „:broj-klijenata“ i vrednost 133.22 čistoj funkciji „int“ koja zaokruži 133.22 na ceo broj 133 i tu vrednost prosledi funkciji „+“ koja je uveća za 10 i vrati drugu nepromenljivu

vrednost 143.

Kombinovanje funkcija na ovakav način — gde se vrednost iz jedne funkcije prosleđuje sledećoj i tako redom se naziva ***kompozicija funkcija***. I kod rekurzije je se dešava ista stvar — rekurzija neprekidno prosleđuje povratnu vrednost iz jednog poziva u drugi samo što se radi o pozivima iste funkcije. Generalno govoreći, funkcionalno programiranje ohrabruje programera da gradi jednostavne funkcije i da ih kompozicijom uvezuje u kompleksnije.

Ovo poređenje sa OOP (objektno orijentisano programiranje) pokazuje i neka ograničenja tog koncepta. U OOP jedna od uloga klasa je u tome da čuva enkapsulirane podatke od nekonzistentnih promena — sa nepromenljivim podacima nemamo taj problem. Zbog toga se metode čvrsto vezuju za klase što ograničava mogućnost njihovog ponovnog korišćenja (*reusability*).

Razlika između načina na koji se piše kod u objektno orijentisanom i funkcionalnom programiranju ukazuje na dublju razliku između ova dva načina razmišljanja. Recimo da je programiranje manipulisanje podacima u različite svrhe. U OOP programer na podatke gleda kao na nešto što se može „upakovati“ u objekte od kojih se kasnije vaja željeni oblik, i kao što vajar teško može dobiti oblik koji je imao na početku tako ni programer ne može rekonstruisati podatke sa početka (osim ako o tome vodi posebnu brigu). Sa druge strane, u funkcionalnom programiranju programer o podacima misli kao o nečemu nepromenljivom i na osnovu tih nepromenljivih informacija izvodi nove. Tokom tog procesa početni podaci ostaju netaknuti kao što broj 1 ostaje netaknut kada na njega dodate 2.

Naravno, OOP daje bogatstvo semantičkih apstrakcija koje takođe pomažu u dekompoziciji i modelovanju problema, ali treba znati da funkcionalno programiranje takođe pruža moćne koncepte apstrakcije koji variraju u zavisnosti od programskog jezika. Ovi koncepti nisu ništa siromašniji od svojih OOP parnjaka ali su svakako prilagođeni drugom načinu razmišljanja. U *Clojure-u* se problem od gore može rešiti (između ostalog) i na sledeći način:

```
(defrecord Prodavnica [broj-klijenata])

(defprotocol KorekcijaKlijenata
  (koriguj-broj-klijenata [prodavnica]))
```

```
(extend-type Prodavnica
  KorekcijaKlijenata
  (koriguj-broj-klijenata [prodavnica]
    (+ 10 (int (:broj-klijenata prodavnica)))))

(koriguj-broj-klijenata (->Prodavnica 133.22)) ; => 143
```

Deo koda gore definiše novi nepromenljivi entitet „Prodavnica“, definiše protokol „KorekcijaKlijenata“ sa jednom funkcijom „koriguj-broj-klijenata“. Proširuje entitet „Prodavnica“ protokolom „KorekcijaKlijenata“ i definiše ponašanje njegove jedine funkcije (sama funkcija je ista kao u prethodnom primeru). *S-izraz* $(->$ *Prodavnica* 133.22) će u osnovi formirati nepromenljivu mapu i dalje će se sve odvijati na isti način kao što je prethodno opisano. Na ovom mestu nećemo dublje ulaziti u koncepte koji se ovde koriste.

- ***Funkcije partial, complement i comp***
  Ideja funkcionalnog programiranje je da omogući izvođenje novih funkcija na isti način kao što omogućuje izvođenje novih podataka iz već postojećih. Ovde će biti predstavljene neke od standardnih funkcija koje čine ovaj posao lakšim.

  Funkcija „partial“ kao argumente prima osnovnu funkciju i bilo koji broj argumenata manji od onog koji prima sama osnovna funkcija. Vraća novu funkciju. Kada se poziva ta nova funkcija, zapravo poziva se ona osnovna funkcija sa parametrima koji su joj dodeljeni kroz „partial“ zajedno sa onim prosleđenim samim pozivom.

```
(defn kompletna-funkcija
  [x y z]
  (+ x y z))

(def zapoceta-funkcija (partial kompletna-funkcija 10 20))

(zapoceta-funkcija 30) ; => 60
```

Ova funkcija se u jezicima iz Lisp familije može veoma prosto implementirati korišćenjem bazičnih funkcija iz poglavlja 1.1. Međutim, ona nije imanentna

Lisp-u već je jedan od bitnih gradivnih koncepata u funkcionalnom programiranju pa je većina funkcionalnih programskih jezika ima u svojoj standardnoj biblioteci. U *Clojure-u* bismo je mogli implementirati na sledeći način:

```
(def zapoceta-funkcija (partial kompletna-funkcija 10 20))

(zapoceta-funkcija 30) ; => 60

(defn moj-partial
  [parcijalizovana-fn & argumenti]
  (fn [& jos-argumenata]
    (apply parcijalizovana-fn (into argumenti jos-argumenata))))

(def zapoceta-funkcija (moj-partial kompletna-funkcija 10 20))

(zapoceta-funkcija 30) ; => 60
```

Ovde funkcija „into“ ima istu ulogu kao „CONS“ u izvornoj Lisp specifikaciji i mogla bi se ovde njome zameniti.

„partial“ je dobro koristiti kada se kombinacija funkcija sa argumentima često pojavljuje u različitim kontekstima.

Funkcija „complement“ prima kao argument osnovnu funkciju i vraća novu funkciju koja vraća logički suprotnu vrednost u odnosu na osnovnu funkciju. Sledeći primer je ilustracija rada „complement“ funkcije.

```
(defn paran?  [broj] (zero? (rem broj 2)))

(def neparan? (complement paran?))

(defn izdvoji-parne
  [brojevi]
  (filter paran? brojevi))

(defn izdvoji-neparne
  [brojevi]
  (filter neparan? brojevi))
```

```
(izdvoji-parne (range 10))    ; => (0 2 4 6 8)
(izdvoji-neparne (range 10)) ; => (1 3 5 7 9)
```

Ovu funkciju ne treba mešati sa negacijom. Negacija se obavlja nad vrednostima, izvršava se momentalno i vraća vrednosti. Zbog toga se koristi u proceduralnim strukturama za kontrolu toka programa kod provere uslova. Komplement sa druge strane radi nad funkcijama i vraća funkcije i služi pri komponovanju funkcija.

Ako su funkcije koje imate na raspolaganju čiste onda je uvek bezbedno da ih komponujete i dobijate nove funkcije jer jedino o čemu treba da vodite računa je veza između ulaza i izlaza. Kompozicija je prirodna akcija u funkcionalnom programiranju kao što je mutiranje atributa u OOP. Zato funkcionalni jezici imaju funkcije koje to olakšavaju. *Clojure* ima „comp“ koja prima proizvoljan broj funkcija i vraća njihovu kompoziciju g.
g je kompozicija funkcija $f_1, f_2, \ldots, f_n$ akko

$$g(x_1, x_2, \ldots, x_m) = f_1(f_2 \ldots (f_n(x_1, x_2, \ldots, x_m))) \tag{1.48}$$

U nastavku je primer koji oslikava korišćenje kompozicije u *Clojure-u*.

```
(def kurs 123.40)

(def korisnik {:ime     "Pera"
               :prezime "Peric"
               :racun   {:broj            123345
                         :dinarsko-stanje 234455}})

(def likvidnost-u-evrima
  (comp int #(/ % kurs) :dinarsko-stanje :racun))

(likvidnost-u-evrima korisnik) ; => 1899
```

Funkcija „likvidnost-u-evrima“ je kompozicija funkcije koja uzima račun korisnika, one koja uzima dinarsko stanje računa, one koja množi dinare sa kursom i funkcije „int“ koja taj broj zaokruži na ceo.

- ***Memoizacija***

  Još jedna pogodnost koju dobija dizajn ako je zasnovan na čistim funkcijama je i mogućnost *memoizacije*. *Memoizacija* je mehanizam za pamćenje rezultata poziva funkcija. Ovo je moguće jer su čiste funkcije *referencijalno transparentne*, a *referencijalna transparentnost* praktično znači da je moguće zameniti izraz poziva funkcije njegovim rezultatom. To znači da npr. možemo zameniti:

  ```
  (- (* 12 (+ 1 2)) 10)
  ```

  sa 26 i program se neće promeniti.
  Na *memoizaciju* se može gledati kao na jednu vrstu *cache-a* koje je na raspolaganju kao koncept programskog jezika. Može se koristiti kako bi se uštedelo na vremenu kada ne postoji realna potreba da se neka funkcija koja je jednom izvršena izvršava ponovo. U nastavku je primer koji je dat u svrhu ilustracije:

  ```
  (defn saberi-prvih-n-brojeva
    [n]
    (apply + (range (inc n))))

  (time (saberi-prvih-n-brojeva 999999999))
  ;; => "Elapsed time: 38398.991135 msecs"

  (def memo-saberi-prvih-n-brojeva (memoize saberi-prvih-n-brojeva))

  (time (memo-saberi-prvih-n-brojeva 999999999))
  ;; => "Elapsed time: 36693.186257 msecs"
  (time (memo-saberi-prvih-n-brojeva 999999999))
  ;; => "Elapsed time: 0.131076 msecs"
  ```

  Funkcija „saberi-prvih-n-brojeva“ sa parametrom 999999999 pri pozivu troši oko 38 sekundi. Međutim, ako na osnovu nje memoizacijom napravimo novu funkciju „memo-saberi-prvih-n-brojeva“ pri njenom prvom pozivu sa istim parametrom biće potrošeno isto vreme kao kod nememoizovanog oblika ali će se svaki sledeći put kada bude pozvana sa istim parametrom izvršavati praktično momentalno.

1.4.3 Lenje izvršavanje — *lazy evaluation*

U teoriji programskih jezika lenje izvršavanje (evaluacija) ili poziv po potrebi (*call-by-need*) je strategija evaluacije koja odlaže evaluaciju izraza do momenta kada to bude potrebno (*non-strict*) i pri tome se izbegava ponovna evaluacija (*sharing*). Lenja evaluacija može smanjiti trajanje neke funkcije za eksponencijalni faktor u odnosu na neke druge strategije *non-strict* evaluacije kao što je poziv po imenu (*call-by-name*).

Lenja evaluacija može dovesti do manjeg korišćenja memorije jer se vrednosti ne čuvaju u njoj do momenta kada postanu potrebne. Međutim, ona je teška za kombinovanje sa imperativnim konstrukcijama kao što su obrade izuzetaka jer redosled operacija pri lenjem izvršavanju nije poznat pre momenta evaluacije.

Suprotno od lenje evaluacije je momentalno izvršavanje (*eager evaluation*) ponekad nazivano i samo — striktna evaluacija (*strict evaluation*). Ova strategija evaluacije je prisutna u većini programskih jezika.

Ponekad se za jezike koji za podrazumevanu strategiju evaluacije imaju lenju evaluaciju kaže da su — lenji jezici (*lazy languages*). Takvi su neki funkcionalni programski jezici kakav je *Haskell*.

Clojure nije lenj jezik ali podržava lenju evaluaciju sekvenci. Ovo znači da elementi sekvence nisu dostupni momentalno već da se proizvode kao rezultat odložene evaluacije. Ta evaluacija se dešava samo kada je potrebna. Evaluacija lenjih sekvenci se naziva *realizacija*(***realization***). Ovakve sekvence se u *Clojure-u* nazivaju lenje sekvence (***lazy sequences***).

Pomoću lenjih sekvenci mogu se predstaviti beskonačne sekvence (ili kako se to naziva u nekim programskim jezicima — generatori). Ako je lenja sekvenca konačna i ako se njena evaluacija završi ona se onda naziva — potpuno evaluirana sekvenca (***fully realized***). Kada je programeru potrebno da potpuno evaluira lenju sekvencu *Clojure* pruža mehanizam kojim se može naložiti njena realizacija.

Pored toga što se mehanizmom lenjih sekvenci mogu predstaviti one beskonačne postoji još jedan snažan benefit koj dobija na značaju u okruženju funkcionalnih koncepata — potpuna evaluacija među-rezultata može biti izbegnuta te njihovo stvaranje postaje jeftino.

U *Clojure-u* se lenje sekvence stvaraju funkcijama. Takve funkcije ili koriste osnovni makro za stvaranje novih kolekcija „lazy-seq“ ili koriste neku drugu funkciju koja vraća lenje sekvence.

„lazy-seq“ prima jednu ili više formi koje proizvode sekvencu ili „nil“ i vraća sekvencijalnu strukturu podataka koja izvršava telo samo prvi put kada je potrebna

neka vrednost i kešira (smešta u *cache* memoriju) je. Dakle, ovaj makro se kombinuje sa memoizacijom. U nastavku je primer formiranja beskonačne sekvence Fibonačijevih brojeva:

```
(defn fib-sekvenca
  ([]
   (fib-sekvenca 0 1))
  ([a b]
   (lazy-seq
    (cons b (fib-sekvenca b (+ a b))))))

(def fibonacijevi-brojevi (fib-sekvenca))

(time (nth fibonacijevi-brojevi 50))
;; => "Elapsed time: 0.333904 msecs"
;;    20365011074
(time (nth fibonacijevi-brojevi 50))
;; => "Elapsed time: 0.14576 msecs"
;;    20365011074
```

Kada zatražimo 50. element u Fibonačijevom nizu prvi put on se evaluira i vraća se rezultat, svaki sledeći uzastopni put se vraća keširana vrednost.

Realizacija lenjih sekvenci može biti izvršena korišćenjem funkcija „dorun“ i „doall“. Razlika je u tome što „dorun“ ne vraća nikakvu vrednost već je namenjena samo za izvršavanje eventualnih sporednih efekata dok „doall“ vraća realizovanu sekvencu. Beskonačne sekvence naravno nema smisla realizovati. Sledeći primer ilustruje razliku između dve opisane funkcije.

```
(doall (map inc (range 10))) ; => (1 2 3 4 5 6 7 8 9 10)
(dorun (map inc (range 10))) ; => nil
```

Mnoge funkcije iz *Clojure* standardne biblioteke vraćaju lenje sekvence. Među njima su i „map“, „reduce“, „filter“, „remove“ i druge.

Postoje dve osnovne strategije za implementaciju lenjih sekvenci — realizovanje elemenata sekvence jedan po jedan (*one-by-one*) i realizovanje elemenata u grupama (*chunks*, *batches*). *Clojure* od verzije 1.1 koristi drugu strategiju (*chunked* realizaciju). Uzmimo

```
(take 10 (range 1 999999999))
```

Ovo je funkcija koja uzima prvih 10 brojeva iz raspona od 1 do 999999999. Realizacija jedan po jedan bi se desila 10 puta — za svaki zatraženi element po jednom. Kod *chunked* realizacije elementi će se realizovati unapred, 32 elementa od jednom (na primeru *Clojure-a*). Obe strategije naravno imaju očigledne prednosti i mane u koje ovde nećemo dublje zalaziti.

1.5 Programski jezik Clojure

Po rečima autora Riča Hikija (*Rich Hickey*)[4] *Clojure* je dinamički, programski jezik opšte namene koji kombinuje pristupačnost i interaktivni način razvoja skripting jezika sa efikasnom i robusnom infrastrukturom za konkurentno programiranje. *Clojure* je kompajliran programski jezik, ostajući pri tome potpuno dinamičan — sve osobine podržane od strane jezika su dostupne u vreme izvršavanja. *Clojure* pruža lake mehanizme komunikacije sa Java okruženjem sa opcionim nagoveštavanjem tipova (*type hints*) i zaključivanjem tipova (*type inference*) koji omogućavaju da Java pozivi izbegnu refleksiju.

Clojure je dijalekat *Lisp-a* i sa njim deli filozofiju „kod su podaci“ (*code-as-data*) kao i moćan sistem makroa. *Clojure* je predominantno funkcionalan programski jezik i pruža širok skup nepromenljivih, perzistentnih struktura podataka. Kada je potrebno promenljivo stanje (*mutable state*) *Clojure* daje na raspolaganje sistem softverske transakcione memoriji (*software transactional memory*) kao i reaktivan sistem agenata koji obezbeđuju čist, korektan, konkurentan dizajn.

Clojure ima nekoliko implementacija. Osnovna implementacija se izvršava na *JVM*[15]. Postoji još *ClojureScript*[5] koji se izvršava na *JavaScript* izvršavačima (*engines*) i *Clojure-clr* koji se izvršava na *Common language runtime*[6] virtualnoj mašini.

Clojure je nastao 2007. godine i predstavlja dinamičko okruženje u kome se čitav razvoj, od kodiranja preko testiranja i debagovanja, odvija u okviru *REPL-a*[4] (*Read–eval–print loop*). Kao što je navedeno u prethodnim poglavljima, mnoge osobine Lisp-a su vremenom preuzimane od strane drugih programskih jezika ali ono što i dalje razdvaja Lisp(pa tako i *Clojure*) od ostalih programskih jezika je princip „kod su podaci“ zajedno sa makro sistemom koji se na tome zasniva. Uz ovu osobinu

[4] *REPL* je još jedan od koncepata koji je uveden kroz Lisp [21].

Clojure dodaje i neke novine — mape, skupovi i vektori su prvoklasni objekti kao što su u izvornom Lisp-u liste.

S obzirom da danas, sa porastom proja procesora u računarima, konkurentno programiranje predstavlja osnovnu postavku *Clojure* u Lisp dodaje i nove koncepte koji konkurentno programiranje dižu na viši nivo apstrakcije u odnosu na imperativno programiranje. Pošto su sve bazične strukture podataka u *Clojure-u* nepromenljive, njihovo deljenje između niti postaje veoma jednostavno. Postoje mehanizmi kojim se može izvesti mutiranje stanja ali se uz njega dostavljaju i mehanizmi koji obezbeđuju da to stanje ostane konzistentno skidajući obavezu sa programera da o tome vodi računa koristeći npr. zaključavanje.

Iako je *Clojure* sa stanovišta funkcionalnog programiranja „prljav“ (*impure*) on i dalje čvrsto stoji iza filozofije da su programi koji su više funkcionalni ujedno i robusniji. Zbog toga promoviše bogat skup koncepata kojim se izbegava mutiranje stanja, daje funkcije kao objekte prvog reda i naglašava rekurzivno iteriranje u odnosu na iteriranje zasnovano na sporednim efektima.

Na raspolaganju je i polimorfizam u vreme izvršavanja (*Runtime polymorphism*). Sistemi koji koriste ovaj mehanizam su lakši za menjanje i proširivanje. *Clojure* poseduje i mehanizam protokola kojima se postiže jednostavno, moćno i fleksibilno apstrahovanje i nezavisni su od platforme tj. implementacije samog jezika.

Clojure je dizajniran da bude „gostujući“ (*hosted*) jezik. Sa izvršavačem na koji je ugošten deli sistem tipova (*type system* ako postoji), *garbage collector*, sistem niti.

1.5.1 Perzistenti vektori

Clojure perzistentni vektori su strukture podataka uvedene od strane Riča Hikija, a inspirisane radom[25] Fila Bagvela (*Phil Bagwell*)[5]. Daju praktično $O(1)$ vreme izvršavanja za operacije dodavanja na kraj, izmene postojećeg, pretragu i pronalaženje podvektora. Pošto su perzistentna struktura, svaka modifikacija kreira novi vektor umesto menjanja starog. Ova struktura podataka omogućuje *Clojure-u* da naglašava imutabilnost i time postiže čitav niz pogodnosti. U ovom poglavlju će biti opisan princip na kome se zasniva rad sa perzistentnim vektorima.

Tradicionalni mutabilni nizovi koji se apstahuju kao niz uzastopnih lokacije koje se mogu produžiti, izmeniti ili smanjiti predstavljaju dobru osnovu kada želimo da postignemo mutabilnost. Kada je ideja postići perzistentnost i imutabilnost onda tradicionalni nizovi ne predstavljaju dobro rešenje. Ako bi koristili tradicionalne

[5]Ilustracije u ovom poglavlju su iz članka *Understanding Clojure's Persistent Vectors*[36].

nizove morali bi smo pri svim akcijama raditi kopiranje čitavog niza što bi operacije činilo suviše sporim i memorijski zahtevnim. Bilo bi dobro da u maksimalnoj meri da izbegnemo redundantnost bez gubitka performansi kada tražimo elemente. Pored toga bitno nam je da sve operacije budu brze. Upravo je to ono što rade perzistentni vektori u *Clojure-u* koristeći balansirana, uređena stabla.

Ideja je implementirati strukturu koja je nalik na N-arno stablo. Jedina razlika je što unutrašnji čvorovi imaju reference na najviše N podčvorova i pri tome ne sadrže nikakve elemente sami po sebi. Listovi sadrže najviše N elemenata. Elementi su uređeni, što znači da je prvi element onaj koji je prvi u krajnjem levom listu. Poslednji element je poslednji u krajnje desnom listu.

Ovde će, zbog jednostavnosti, biti uzeto da je $N = 2$ (binarno stablo) i da se svi listovi nalaze na istoj dubini. Ova ograničenje ne utiče bitno na suštinu opisanih mehanizama.

Za početak, pogledajmo primer na slici 1.4; vektor sadrži cele brojeve od 0 do 8, gde je 0 prvi element a 8 poslednji. Vektor je dužine 9.

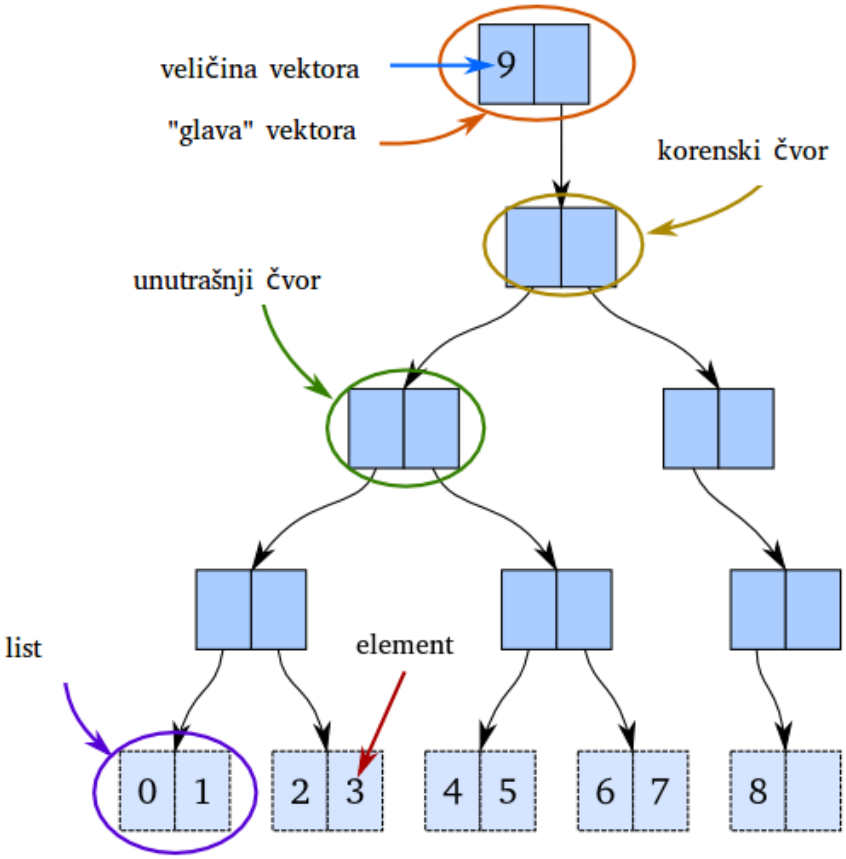
Postavlja se pitanje — kako se dodaje element na kraj vektora?
Ako je mutablinost prihvatljiva za nas onda možemo prosto da dodamo vrednost u krajnji desni element krajnje desnog lista, kao na slici 1.5.

Ovo, međutim, ne možemo da uradimo ako želimo da imamo perzistenciju. Morali bismo da kopiramo čitavu strukturu kako bismo to postigli. Kako bismo smanjili kopiranje možemo da preduzmemo kopiranje putanje — kopiranje svih čvorova koji se nalaze na putanji do čvora koji želimo da izmenimo ili dodamo. Kada stignemo do željenog čvora ostaje samo da uradimo samu operaciju promene ili dodavanja (ovde nećemo diskutovati algoritme za pronalaženje željeneog čvora). Na slici 1.6 je prikazana situacija nakon uzastopnih dodavanja brojeva 7, 8 i 9 na vektor koji sadrži brojeve od 0 do 6 (nakon svakog od ovih dodavanja nastaje novi vektor, na slici je prikazan samo poslednji). Ružičastom bojom su označeni čvorovi koji se dele između prvobitnog vektora i krajnjeg, narandžasti su oni koji su samo u prvobitnom, a plavi su oni koji su samo u krajnjem vektoru.

Sada kada nam je poznat koncept kopiranja putanje možemo da pređemo na diskusiju o osnovnim operacijama nad vektorom — izmene (*update*), dodavanja (*append*), uklanjanje poslednjeg (*pop*).

- **Izmena**

  Ova operacija se u *Clojure* programskom jeziku radi funkcijama „assoc“ ili



Slika 1.4: Primer perzistentnog vektora.

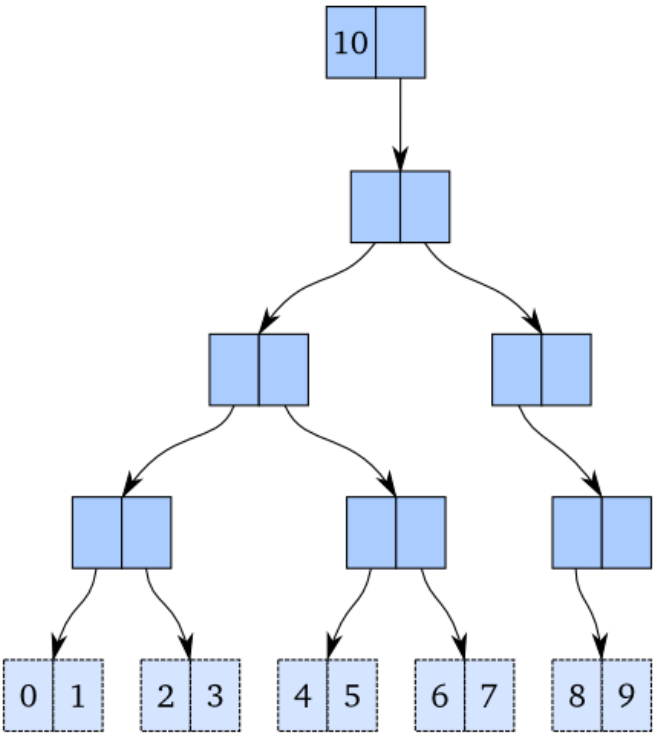
„update“/„update-in“. Da bismo izmenili element potrebno je da prođemo kroz stablo do čvora u kome se nalazi element koji menjamo i pritom radimo kopiranje putanje da bi smo obezbedili perzistenciju. Kada dođemo do tog čvora, kopiramo ga menjajući željeni element i vratimo novonastali vektor.

Uzmimo za primer da želimo da promenimo peti element u vektoru koji sadrži vrednosti od 0 do 8.

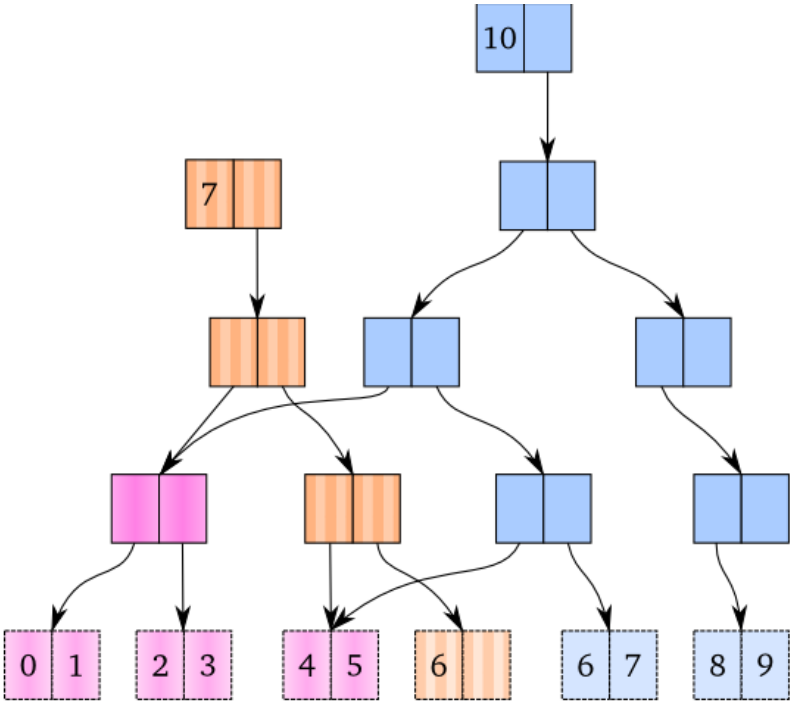
```
(def narandzasti [0 1 2 3 4 5 6 7 8])
(def narandzasti-sa-plavim (assoc narandzasti 5 'beef))
```

Na slici 1.7 su plavom bojom označeni čvorovi koji su stvoreni kroz kopiranje putanje, a narandžasti su deljeni.

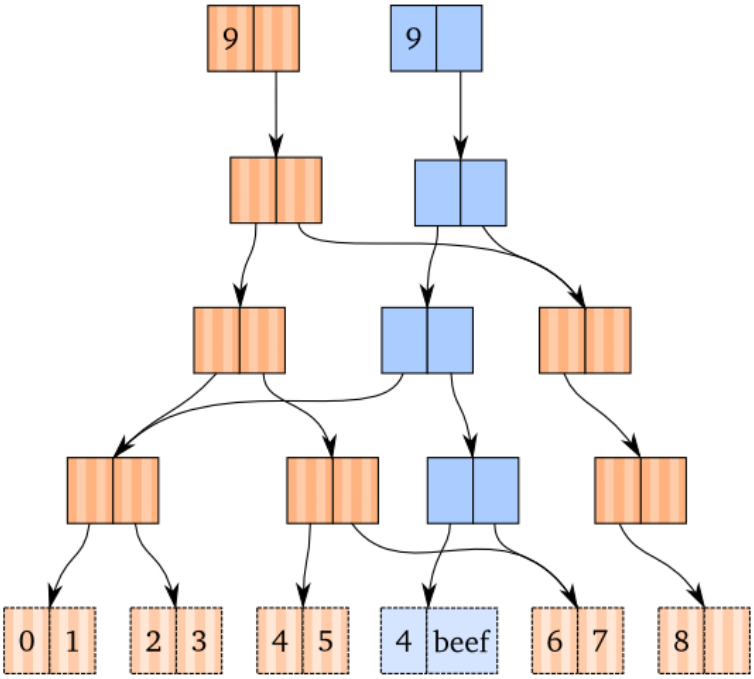
- **Dodavanje**
  Dodavanje nije bitno različito od izmene, osim što kod dodavanja imamo situacije kada moramo da dodamo novi čvor kako bi imali gde da dodamo novi element. Dakle, razlikujemo tri situacije:



Slika 1.5: Dodavanje kada je mutabilnost prihvatljiva.



Slika 1.6: Kopiranje putanje.



Slika 1.7: Izmena vektora.

1. Ima mesta za novu vrednost u krajnje desnom listu.
2. Ima mesta u korenskom čvoru ali nema u krajnje desnom listu.
3. Nema dovoljno mesta u korenskom čvoru.

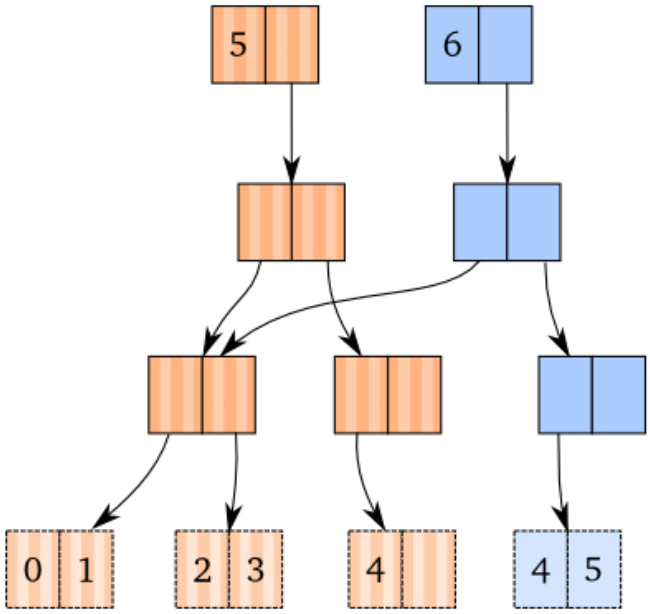
Proćićemo kroz sve tri situacije.

1: **Slično kao izmena**
Kadagod ima dovoljno mesta u krajnje desnom listu potrebno je samo da dođemo do njega, pritom radeći kopiranje putanje. Nakon toga ostane da željenu vrednost stavimo u krajnje desni element tog čvora. Na primer operaciju:

```
(conj [0 1 2 3 4] 5)
```

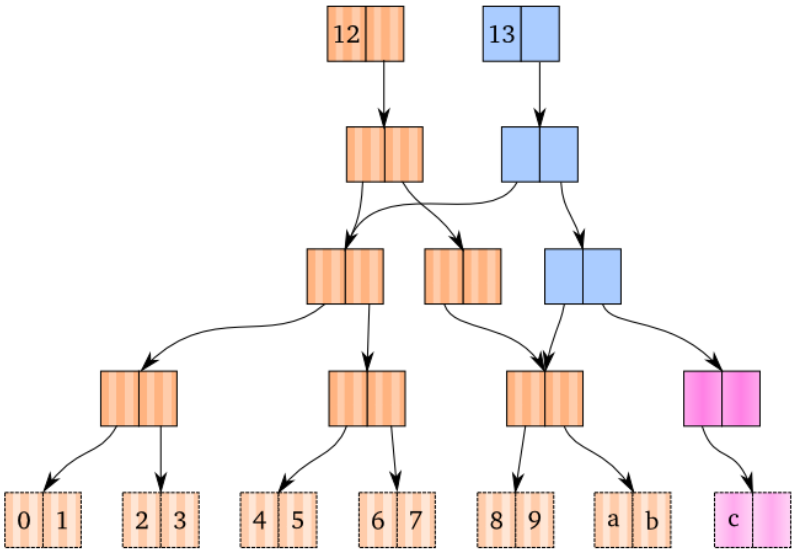
Možemo da predstavimo slikom 1.8. Narandžasti su elementi starog vektora, plavi su iz novog.

2: **Generisanje čvorova kada su potrebni**
Algoritmi kojim se spuštamo do lista u stablu nam uvek obezbeđuju da znamo da smo pozicionirani na pravi čvor. Ako ustanovimo da čvor na koji želimo da siđemo još uvek ne postoji (npr. pokazivač je „null“) to znači da treba da ga napravimo. Novonastali čvor obeležimo kao kopirani.



Slika 1.8: Prikaz operacije „conj“ nad vektorom.

Na slici 1.9 je prikazana situacija dodavanja pri čemu su plavom bojom označeni kopirani čvorovi mehanizmom kopiranja putanje, a ružičastom oni koje smo morali da dodamo.

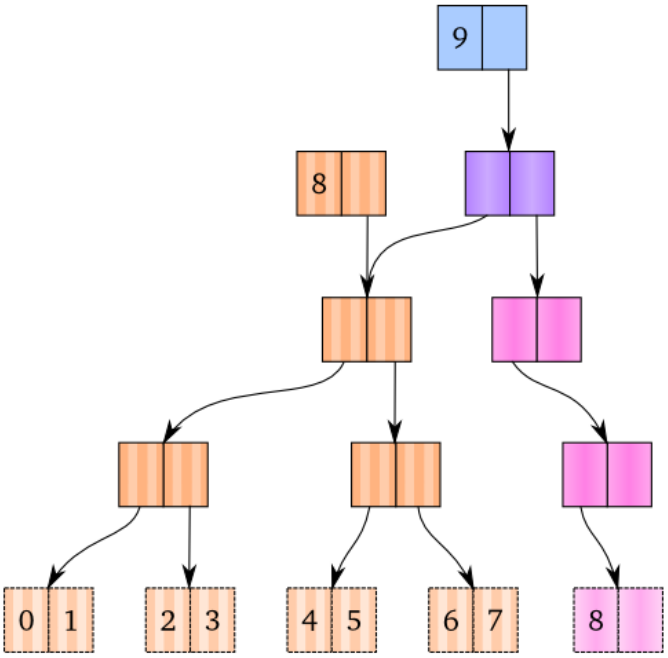


Slika 1.9: Dodavanje novog čvora.

3: **Prelivanje korenskog čvora**

Kada imamo situaciju da u korenskom čvoru nema mesta za nove čvorove (to se dešava kada svi čvorovi imaju N podređenih) onda moramo da pra-

vimo novi korenski čvor. To radimo tako što napravimo novi čvor koji postavimo kao roditeljski korenskom čvoru. Od tog čvora radimo generisanje novih kada su potrebni, kao što je prethodno opisano. Na slici 1.10 je ljubičasti čvor taj koji smo dodali kao novi korenski čvor, a ljubičaste smo potom dobili kopiranjem putanje.



Slika 1.10: Dodavanje novog korenskog čvora.

Pored toga kako se rešava ova situacija bitno je i zaključiti kako do nje dolazi kako bi se proces mogao optimizovati. Ako je faktor grananja N onda do ove sitaucije dolazi kada je veličina starog vektora stepen N. U našem pojednostavljenom slučaju — ako je stepen dvojke.

- **Uklanjanje poslednjeg**
  Uklanjanje poslednjeg je slično sa dodavanjem u tome što takođe razlikujemo tri slučaja:

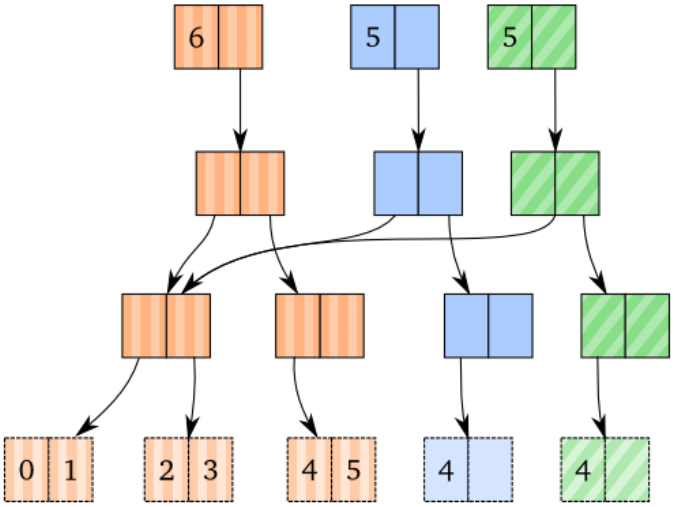
  1. Krajnje desni list ima više od jednog elementa.
  2. Krajnje desni list ima jedan element (nijedan nakon uklanjanja).
  3. Korenski element ima samo jedan podređeni nakon uklanjanja.

  1: **Obično uklanjanje**
  Slučaj kada list sa koga treba da radimo uklanjanje ima više od jednog

elementa je trivijalan — potrebno je samo da siđemo do njega kopirajući putanju i na krajnji list kopiramo bez elementa koga želimo da obrišemo. Na slici 1.11 se je oslikana situacija iz sledećeg koda:

```
(def narandzasti [0 1 2 3 4 5])
(def plavi (pop narandzasti))
(def zeleni (pop narandzasti))
```

Primetimo da će uzastopne primene iste operacije dovesti do stvaranja više vektora koji će biti jednaki ali to svakako nije isti vektor.



Slika 1.11: Obično uklanjanje poslednjeg. Dve uzastopne operacije.

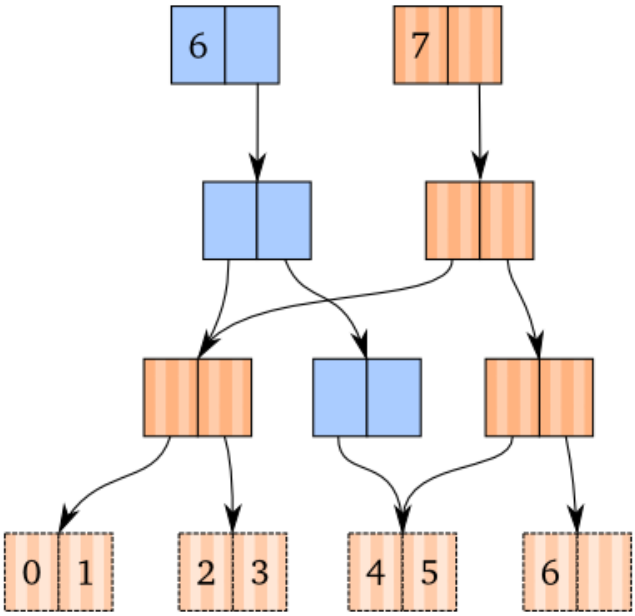
2: **Uklanjanje praznih čvorova**

Želimo da izbegnemo prazne čvorove po svaku cenu. Zato imamo specifičnu situaciju kada list ima samo jedan element. Kadagod naiđemo na prazan čvor potrebno je da ga izbacimo. To npr. možemo uraditi tako što ćemo umesto praznog čvora vratiti „null“ i onda će njegov roditelj imati prazan pokazivač umesto da pokazuje na prazan čvor. Na primeru sa slike 1.12 imamo da nakon uklanjanja broja 6 sa kraja njegov roditelj biva prazan čvor. Vidimo da rezultujući vektor (plavi) ne sadrži taj čvor.

Ovde nećemo diskutovati algoritam kojim se postiže propagacija „null“ pokazivača na gore i samo uklanjanje praznog čvora. Dovoljno nam je da ilustrujemo tu mogućnost i da uvidimo njenu moguću efikasnonst.

3: **Uništavanje korena**

Nakon što uradimo uklanjanje poslednjeg potrebno je da proverimo da li korenski čvor ima samo jednog podređenog. Ako je to tačno i ako on nije



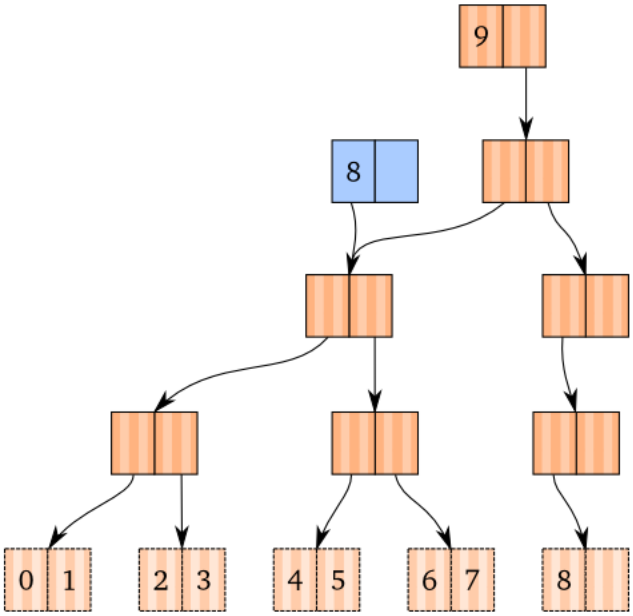
Slika 1.12: Uklanjanje praznog čvora.

ujedno i list onda je potrebno da korenski čvor obrišemo i promovišemo njegovog podređenog za novi korenski čvor. Ako je poštujemo našu početnu pretpostavku o faktoru grananja N i istoj dubini na kojoj su svi listovi onda nakon uklanjanja poslednjeg iz vektora od 10 elemenata imamo situaciju kao na slici 1.13. Narandžasti su čvorovi iz vektora nakon uklanjanja broja 9 sa kraja, plavi čvor je novi koren nakon uklanjanja korena.

Clojure koristi faktor grananja $N = 32$ što rezultuje veoma plitkim stablom. Zapravo, ta stabla će biti duboka svega 6 čvorova za vektore koji imaju manje od milijardu elemenata. Pošto imamo veoma plitka stabala, možemo reći da su modifikacija i pretraga vektora „efektivno“ u konstantnom vremenu. U teoriji one su $O(\log_{32} n)$ što je praktično $O(\log n)$.

1.5.2 Upotrebljivost *Clojure* ekosistema

Svo bogatstvo Lisp-a i funkcionalnog programiranja je potpuno dostupno unutar standardnog *Clojure* ekosistema u kome programer provodi vreme. Na raspolaganju je praktično sve, od razvojnih okruženja do *Web* servera, biblioteka za programiranje dronova, okruženja za rad sa neuronskim mrežama, *rule engine-a*, sistema za ispitivanje distribuiranih sistema, okruženja za statističko računanje (sličnih kao *R*),



Slika 1.13: Uklanjanje korena.

monitoring sistema za distribuirano računarstvo, baza podataka, okruženja za *frontend* programiranje i dr. sve do okruženja za interaktivno komponovanje muzike i pravljenje animacija.

Integrisana razvojna okruženja koja su dostupna za *Emacs*, *VIM*, *IntelliJ*, *Eclipse*, *Light Table* i još neke editore su na različitom nivou razvoja. Najnaprednije od njih je svakako *CIDER*[3] za *Emacs* i pruža veoma kvalitetan skup alata i mogućnosti. Ako znamo da je *Clojure* dinamički jezik, a da je pravljenje alata za takve jezike donekle ograničeno u odnosu na statičke, moramo primetiti da je *CIDER* trenutno na visokom stupnju razvoja. Itekako je uporediv sa integrisanim okruženjima za razvoj u drugim dinamičkim jezicima kao što su *Python* i *Ruby*.

Kada govorimo o skupu tehnologija za razvoj *Web* aplikacija, stanje je takođe veoma dobro. Postoji čitav niz pristupa, neki od njih su proistekli iz zajednice, drugi su pak rezultat potreba kompanija koje aktivno koriste Clojure. Situacija je jednaka kako na *backend-u* (gde se koristi standardna *JVM* implementacija — *Clojure*) kao i na *frontend-u* (gde se koristi *JavaScript* implementacija — *ClojureScript*).

Raznovrsnost u pristupima kandiduje *Clojure* kao veoma interesantan alat za neke nestandardne probleme na *Web-u* i u distribuiranom računarstvu. Kao posledica dobro podržanih koncepata funkcionalnog programiranja, Lisp-a i novinama uvedenim u samom *Clojure-u*, razvoj rešenja je izuzetno elegantan i brz. Zbog visokog nivoa apstrakcije nasleđenog kroz Lisp lozu, *Clojure* omogućuje da se rešenja izražavaju na

nivou apstrakcije koji je već sam po sebi ilustracija dizajna.

Danas, kada skoro svaki program rukuje ogromnom količinom raznorodnih podataka iz različitih izvora, koncept „podaci su kod“ postaje upotrebljiviji nego što je ikada ranije bio.

Problem?

Naravno, tu su i problemi. Najveći problem koji stoji na putu *Clojure-a* da postane „*mainstream*“ jezik je upravo jedan od njegovih najvećih kvaliteta — to što je Lisp. Za veliki broj programera, Lisp i dalje (nakon 60 godina) predstavlja „čudan“ način zapisa. Oni u velikoj meri razumeju moć koju time dobijaju ali je njihov argument za odbijanje po pravilu „komplikovana sintaksa sa previše zagrada“. Zapravo Lisp „sintaksa“ nije ništa komplikovanija od npr. one u *JavaScript-u*:

```
var nekaFunkcija = function(objekat) {
    console.log(objekat);
}

(function () {
    var vrednost = 11;
    nekaFunkcija({polje1: "vrednost", polje2: vrednost})})();
```

Ovaj deo koda u *JavaScript-u* ima 18 zagrada (i pokazuje da *JavaScript* takođe ima korene u Lisp-u).

```
(defn neka-funkcija
  [objekat]
  (println objekat))

(let [vrednost 11]
  (neka-funkcija {:polje1 "vrednost" :polje2 vrednost}))
```

Ovaj deo koda u *Clojure-u* ima 14 zagrada.

Bilo kako bilo, ovaj problem svakako nije nešto što bi *Clojure* kao programski jezik trebalo da reši, već pre zajednica okupljena oko njega kroz komunikaciju sa drugim programerima i demistifikaciju *Clojure* koncepata.

Drugi, mnogo realniji problem, je brzina započinjanja projekata. Vreme koje je potrebno za uspostavljanje jedne standardne npr. *Web* aplikacije je duže nego u *Ruby* ili *Python* ekosistemima. Ako se programer odluči da koristi neki već gotov skup biblioteka za rad na *Web-u* kao što su *Pedestal*[19] ili *Luminus*[17] onda su problemi poput postavljanja *Web* servera, deskripcije *REST* servisa ili komunikacija *backend-a* i *frontend-a* rešeni. Vreme potrebno za slične zadatke je slično kao u navedenim okruženjima.

Međutim, ako programer ima potrebu za velikim brojem manje-više standardnih entiteta i ako želi da ih povezane čuva u nekoj od standardnih baza podataka, vreme potrebno da uspostavi svoj projekat može u startu da se produži. U *Clojure-u* trenutno ne postoji neko standardno rešenje za onaj problem koji se u objektnom programiranju rešava pomoću *objektno relacionog mapiranja*. Dakle, nema stvari sličnih *Active Record-u* u *Ruby-ju* ili *Django Model* u *Python-u*.

Clojure, kao i svi iz Lisp familije, ima razvijenu kulturu pisanja internih jezika specifičnih za domen. U ovom radu će biti predstavljen jedan takav jezik kao predlog za rešavanje gorenavedenog problema.

Glava 2

Pregled stanja u oblasti

U ovom poglavlju će biti opisano trenutno stanje u oblasti perzistencije podataka kao i *Web* programiranju u okviru *Clojure* ekosistema. Biće predstavljeni koncepti od onog na najnižem nivou apstrakcije komunikacije sa bazama podataka, preko domen specifičnih jezika koji podižu nivo apstrakcije do okruženja (*framework-a*) za *Web* programiranje.

Pošto je *Clojure* ugošten na *JVM*, on može da koristi *Java* biblioteke. Na niskom nivou komunikacije sa bazama podataka to je obično i slučaj.

2.1 *Java Database Connectivity — JDBC*

JDBC je Java *API* koji definiše kako klijent komunicira sa bazom podataka. Deo je *Java Standard Edition* od verzije *1.1* iz 1997. godine. Klase koje spadaju u *JDBC* nalaze se u paketu „java.sql“. Namenjen je u glavnom za rad sa relacionim bazama podataka. Od verzije *3.1* razvoj *JDBC* je pod procesom razvoja u okviru *Java* zajednice (*Java Community Process*).

JDBC dozvoljava da postoji nekoliko implementacija i da one budu korišćene od strane iste aplikacije. *API* ima mehanizam koji obezbeđuje da se dinamički učitavaju korektni *Java* paketi kroz *JDBC Driver Manager*. Taj *Manager* se koristi kao fabrika za stvaranje *JDBC* konekcija (ali se u praksi češće koriste *connection pools* nego direktno uzimanje konekcija iz *JDBC Driver Manager-a*). *JDBC* omogućuje davanje naredbi bazama podataka.

Naredbe mogu biti *update*, kao što su u *SQL CREATE, INSERT, UPDATE* i *DELETE*, ili upitne kao što je *SQL SELECT*.

Takođe je moguće izvršavati i uskladištene procedure.

JDBC predstavlja naredbe sledećim klasama:

- **Statement** — naredba se šalje serveru svaki put.
- **PreparedStatement** — naredba se kešira i potom se izračunava putanja njenog izvršavanja kako bi se kasnije mogla izvršavati više puta na efikasniji način.
- **CallableStatement** — koristi se za uskladištene procedure.

Update naredbe kao što su *INSERT, UPDATE* i *DELETE* vraćaju broj promenjenih redova.

Upitne komande vraćaju *JDBC row result set*. Kroz ovu strukturu se može prolaziti i dobijati rezultate upita, pri tom se kolonama može pristupati po indeksu ili po imenu. Struktura takođe ima i meta podatke koji govore o tome kog su tipa kolone.

2.1.1 *Clojure* implementacije *JDBC*

U *Clojure* ekosistemu se ističu dve implementacije *JDBC API-ja*. Oba su u upotrebi i biblioteke koje koriste *JDBC* teže ka tome da imaju podršku za obe.

2.1.1.1 clojure.java.jdbc

clojure.java.jdbc[22] je namenjen da bude *wrapper* niskog nivoa oko različitih *JDBC* drajvera za baze podataka.

Generalna ideja je da se postavi izvor podataka — „database spec“ i da se potom on prosleđuje raznim *CRUD* (*create, read, update, delete*) operacijama koje pruža „java.jdbc“. Po automatizmu, svaka operacija će otvoriti konekciju ka bazi podataka i izvršiti se unutar transakcije. Moguće je raditi više operacija unutar jedne konekcije, bilo kroz transakciju, bilo kroz *connection pooling* ili kroz deljenu konekciju. Primer korišćenja je dat u nastavku:

```
(def db-spec
  {:classname   "com.mysql.jdbc.Driver"
   :subprotocol "mysql"
   :subname     "//127.0.0.1:3306/moja_baza_podataka"
   :user        "nalog"
   :password    "lozinka"})
```

Definišemo specifikaciju baze koja sadrži ime klase koja predstavlja *JDBC driver*, subprotokol, ime baze („subname“) i korisničko ime i lozinku. Sada možemo na osnovu specifikacije napraviti *pool* konekcija, ako to želimo:

```
(def pooled-db-spec
  {:datasource (napravi-pool db-spec)})
```

Gde je „napravi-pool“ funkcija koja vraća *pool* konekcija. Nju možemo implementirati korišćenjem različitih *Java* biblioteka. Neke koje su se kroz vreme pokazale kao najkorišćenije su *HikariCP* i *c3p0*.

Drugi način za ponovno korišćenje jedne iste konekcije na bazu podataka su transakcije. U nastavku je primer kako možemo da iskoristimo „with-db-connection“ makro iz „java.jdbc“ da bismo to postigli.

```
(ns db-primer
  (:require [clojure.java.jdbc :as jdbc]))

(with-db-connection [db-con db-spec]
  (let [redovi (jdbc/query db-con
                           ["SELECT * FROM tabela WHERE ime like \'%?%\'" "ike"])]
    (jdbc/insert! db-con :table (dissoc (first redovi) :id))))
```

Ovde učitavamo sve redove čija kolona „ime“ sadrži slova „ike“, potom uzimamo prvu od njih i snimamo u bazu njen duplikat. Ideja ovog primera je da pokaže da se obe navedene operacije odvijaju unutar jedne transakcije i da smo time koristili samo jednu konekciju — samo jedan put je otvarana konekcija prema serveru baze podataka.

Razlog zašto name je potrebno ponovno korišćenje konekcija na baze podataka je u tome što svako otvaranje i zatvaranje konekcije troši određeno vreme. To vreme je jedinično vrlo malo u odnosu na poslove koje inače rade aplikacije koje rade na *Web-u* ali je komunikacija sa bazom podataka obično vrlo česta. Vreme koje se akumulira u otvaranju i zatvaranju svih konekcija posle nekog vremena rada aplikacije postaje nezanemarljivo. *Connection pooling* je strategija kojom se obezbeđuje da aplikacija uvek ima na raspolaganju optimalan broj konekcija tako što uvodi pametniji raspored otvaranja i zatvaranja, kao i mehanizam ponovnog korišćenja. Klasičan način rada je da aplikacija uzme konekciju iz *connection pool-a* kada joj je ona potrebna i da je vrati tamo kada sa njom završi — sam posao otvaranja, zatvaranja i odabira konekcije se ostavlja u nadležnost biblioteci za *connection pooling*.

2.1.1.2 conman

Pri korišćenju „java.jdbc“ ako želite da koristite *pool* konekcija sama biblioteka ne pruža nikakvu pomoć. Na programeru (ili drugoj biblioteci) je da to uradi za sebe. Osnovna ideja je pribaviti biblioteku za *connection pooling* u svoj projekat.

Potom definisati funkciju koja prima specifikaciju baze podataka, vraća mapu koja sadrži ključ „:datasource“ čija je vrednost „DataSource“ objekat. Potom tu povratnu vrednost koristiti na mestu gde se koristi obična specifikacija baze. Jedna od biblioteka koja vam ovo omogućuje koristeći *HikariCP* je *conman*[8]. Više reči o ovoj biblioteci i načinu njenog funkcionisanja biće u poglavlju 2.2.2.2.

2.1.1.3 clojure.jdbc

Još jedna implementacija *JDBC* za *Clojure*. Za nju svakako postoje razlozi:

- Upravljanje konekcijama u „clojure.jdbc“ je jednostavnije i eksplicitnije. „java.jdbc“ ne pravi jasnu razliku između specifikacije baze i same konekcije na nju. To može izgledati kao fleksibilnije rešenja ali ono za posledicu ima da svaka funkcija koja prima konekciju mora da ima svu kompleksnost obrade tog ulaznog parametra.
- „clojure.jdbc“ dolazi sa sređenim upravljanjem transakcijama koje u potpunosti podržava ugnježdene transakcije i zadavanje transakcionih strategija. Nasuprot nje je „java.jdbc“ koja dolazi samo sa jednom transakcionom strategijom i ne podržava ugnježdene transakcije („poravnava“ transakcije — čini da se sve transakcije pretvore u jednu što može dovesti do pogrešnih sigurnosnih ograničenja).
- Dolazi sa ugrađenom podrškom za *connection pooling*.
- Uglavnom ima bolje performanse zbog bolje implementiranog *JDBC*. „clojure.jdbc“ implementacija je čistija i ima manje nepotrebnog (*boilerplate*) koda.

Koristi se na sličan način kao i „java.jdbc“ uz to da je veoma lako umesto obične konekcije koristiti *connection pool*.

```
(def dbspec {:vendor   "postgresql"
             :name     "moja_baza_podataka"
             :host     "localhost"
             :port     5432
             :user     "ime"
             :password "lozinka"})

(with-open [konekcija (jdbc/connection dbspec)]
  (jdbc/execute konekcija
```

```
                "CREATE TABLE moja_tabela (id serial, ime text);")
  (jdbc/execute konekcija
                ["insert into moja_tabela (ime) values (?);" "Pera"]))
```

Ako sada hoćemo da koristimo npr. *HikariCP* potrebno je samo da definišemo mapu koja opisuje naš željeni *connection pool*, to prosledimo funkciji „make-datasource“ iz *HikariCP* biblioteke i potom u „with-open“ bloku koristimo njenu povratnu vrednost. Ovaj niz koraka je prikazan u nastavku:

```
(require '[hikari-cp.core :as hikari])

(def ds (hikari/make-datasource
         {:connection-timeout 30000
          :idle-timeout       600000
          :max-lifetime       1800000
          :minimum-idle       10
          :maximum-pool-size  10
          :adapter            "postgresql"
          :username           "ime"
          :password           "lozinka"
          :database-name      "moja_baza_podataka"
          :server-name        "localhost"
          :port-number        5432}))

(with-open [konekcija (jdbc/connection ds)]
  ;; akcije nad konekcijom
  )
```

Ovo svakako predstavlja jednostavniji način za korišćenje *connection pooling-a* od onog koji nam je na raspolaganju kod „java.jdbc“ biblioteke.

Rad sa transakcijama je jednostavniji i podržava razne nivoe izolacije koje podržava i sam *JDBC*. Da biste izvršili akcije u transakciji potrebno je samo da iskoristite „atomic“ makro.

```
(jdbc/atomic konekcija
             (prva-akcija konekcija)
             (druga-akcija konekcija))
```

Ovaj makro radi na način koji je mnogo prikladniji funkcionalnom programiranju. Umesto da koristi dinamičke promenljive lokalne za nit izvršavanja da čuva stanje transakcije u konekciji kao što to radi „java.jdbc“, „clojure.jdbc“ koristi prepisivanje (*overwrite*) u leksičkom okruženju promenljive za konekciju novom konekcijom koja ima transakciono stanje. U same detalje implementacije nećemo ovde dublje zalaziti.

Što se tiče nivoa izolacije, na raspolaganju su: „:read-uncommited“, „:read-commited“, „:repeatable-read“, „:serializable“ i „:none“ kao podrazumevano. Ove ključne reči se dodaju u mapu koja opisuje konekciju pod ključem „:isolation-level“. Naravno, ne podržavaju svi *JDBC* drajveri sve nivoe izolacije.

2.2 Jezici za komunikaciju sa relacionim bazama podataka

Veoma raznovrsna *Clojure* zajednica je proizvela mnogo rešenja za komunikaciju sa relacionim bazama podataka putem *SQL-a*. Sva ta rešenje sa sa aspekta jezika specifičnih za domen mogu podeliti u dve grupe. Prva grupa je ona koja sledi klasičnu Lisp filozofiju pravljenja internih JSD, druga je grupa koja uvodi novu ideju koja se zasniva na proširenju *SQL-a* i dinamičkom generisanju funkcija na osnovu toga.

Ako bismo ovu podelu posmatrali sa stanovišta Lisp-a onda bismo mogli reći da je prva ideja proistekla iz fleksibilnosti Lisp sistema *S-izraza* dok je druga inspirisana Lisp-ovim (dinamičkim) shvatanjem funkcionalnog programiranja.

2.2.1 *Clojure* interni JSD

Ova grupa JSD se zasniva na zapisivanju koncepata relacionih baza podataka na nivou *SQL-a* ili na nivou samih entiteta i relacija. Predstavlja ideju koja je prvobitno korišćena u *Clojure* zajednici. Postoji mnogo različitih rešenja koja koriste ovaj koncept i sva unose neke novine. U ovom poglavlju biće predstavljeno nekoliko takvih rešenja.

2.2.1.1 HoneySQL

Ideja *HoneySQL-a* je predstavljanje *SQL-a* kao *Clojure* struktura podataka. Upiti se posledično mogu generisati i u vreme izvršavanja bez potrebe za spajanjem stringova. Svi upiti se definišu na nivou *Clojure* mapa. Tako na primer jedan prost upit izgleda ovako:

```
(def sql-mapa {:select [:ime :prezime :nadimak]
               :from   [:covek]
               :where  [:= :covek.mesto_stanovanja "Novi Sad"]})
```

Moguće je pozvati funkciju „honeysql.core/format“ nad nekom ovakvom mapom i dobiti parametrizovan *SQL* vektor koji će biti poslat preko *JDBC*.

```
(require '[honeysql.core :as sql])

(sql/format sql-mapa)
;; => ["SELECT ime, prezime, nadimak FROM covek WHERE covek.mesto_stanovanja = ?"
;;     "Novi Sad"]
```

Na raspolaganju je i funkcija „honeysql.core/build“ kojom se mogu dinamički generisati upiti.

```
(def upit-prvi (sql/build :select :*
                          :from :covek
                          :where [:= :covek.ime :like "va"]))
```

Redosled parova ključ-vrednost nije bitan jer će biti vraćena *Clojure* mapa koja svakako kao struktura podataka ne poštuje redosled elemenata. *HoneySQL* će izgraditi dobar upit u svakom slučaju.

```
(def upit-drugi (sql/build :from :covek
                           :where [:= :covek.ime :like "va"]
                           :select :*))

(= upit-drugi upit-prvi) ; => true
```

U prostoru imena „honeysql.helpers“ postoje i funkcije koje praktično pokrivaju sve moguće *SQL* uslove tako da je moguće formirati upite i ulančavanjem ili komponovanjem funkcija.

```
(-> (select :*)
    (from :covek)
    (where [:> :godine 21] [:> :saradnici 4]))
;; => {:select (:*), :from (:covek), :where
;;     [:and [:> :godine 21] [:> :saradnici 4]]}
```

Ako neke operacije u *WHERE* klauzuli nisu podržane postoji mogućnost da ga definišete sami dodavanjem metoda u „honeysql.format/fn-handler“ multimetod[1]. Ako ne postoji ni klauzula koju želite da dobijete i nju možete dodati sličnim mehaniznom uz proširivanje multimetoda „honeysql.format/format-clause“.

2.2.1.2 SQLingvo

Interni JSD koji se može koristiti iz *Clojure* ili *ClojureScript*. Kompatibilan je sa „clojure.jdbc“, „clojure.java.jdbc“, „postgres.async“ i „node-postgres“ (kroz *ClojureScript*) drajverima. Osnovna namena mu je da se koristi sa *Postgres*[20] bazom podataka, ali ako se ne koriste *Postgres* specifične komande onda se može koristiti i sa ostalim bazama podataka.

Koristi konfiguracije baza podataka da podesi na koji način se radi navođenje identifikatora (*quoting*) i kako se prevode imena kolona i tabela između *Clojure* i baze podataka. Konfiguracije za neke standardne baze podataka kao što su *PostgreSQL* i *MySQL* su već data u prostoru imena „sqlingvo.db“. Kod u nastavku formira konfiguraciju baze podataka sa strategijom imenovanja i navođenja za *PostgreSQL*.

```
(require '[sqlingvo.db :as db])
(def baza (db/postgresql))
```

Ovakva konfiguracija baze podataka je potrebna za sve funkcije koje vraćaju *SQL* naredbe. Kod u nastavku koristi gorenavedenu konfiguraciju da izgradi jednostavan upit.

```
(sql (select baza [:first-name] (from :people)))
;; => ["SELECT \"first-name\" FROM \"people\""]
```

Vektori dobijeni iz „sql“ funkcije su oni koji se šalju u „clojure.java.jdbc“ ili „clojure.jdbc“ na izvršavanje.

Ako je potrebno napraviti neki parametrizovani upit to se radi uz pomoć funkcija i Lisp citiranja sintakse (*syntax quoting*[2]). U nastavku je dat primer koda u kome je definisan jedan parametrizovani upit.

[1] Multimetodi su uz protokole način za postizanje polimorfizma u *Clojure-u*[32].

[2] U ovom radu nećemo dublje zalaziti u Lisp *evaluation process*, više o citiranju sintakse i *Reader-u* u *Clojure* dokumentaciji[34]

```
(defn pretrazi-po-mestu-stanovanja
  [baza grad]
  (sql (select baza [:ime :prezime]
         (from :covek)
         (where `(= :grad ~grad)))))

(pretrazi-po-mestu-stanovanja baza "Novi Sad")
;; => ["SELECT \"ime\", \"prezime\" FROM \"covek\"
;;       WHERE (\"grad\" = ?)" "Novi Sad"]
```

U *SQLingvo* kao i u *HoneySQL* ideja je da se daju sve funkcije koje predstavljaju *SQL* koncepte i da onda korisnik na osnovu njih izgrađuje svoje upite (i uopšte naredbe bazi podataka). U nastavku je dat jedan kompleksniji upit koji zahteva i spajanje tabela.

```
(select baza [:*]
  (from :vremenska-prognoza)
  (join :grad '(on (= :grad.ime :vremenska-prognoza.grad))
        :type :inner)
  (group-by :grad.drzava)
  (order-by :vremenska-prognoza.stepeni))
;; => ["SELECT * FROM \"vremenska-prognoza\" INNER JOIN \"grad\"
;;      ON (\"grad\".\"ime\" = \"vremenska-prognoza\".\"grad\")
;;      GROUP BY \"grad\".\"drzava\"
;;      ORDER BY \"vremenska-prognoza\".\"stepeni\""]
```

SQLingvo i *HoneySQL* su u principu dva veoma slična pristupa uz tu razliku što *HoneySQL* pruža mogućnost definisanja naredbi i u formatu *Clojure* mape što daje na jednostavnosti. Unificiran pristup obično znači i veću fleksibilnost, mada se ni pristup opredeljen na obične funkcije i citiranje sintakse nikako ne može nazvati nefleksibilnim.

2.2.1.3 Korma

Ovaj interni JSD ide izvan okvira prethodna dva. Pored toga što *SQL* predstavlja kao *S-izraze* on uvode i koncepte entiteta i relacija između njih. Korisnik ne mora da razmišlja na nivou samih *SQL* naredbi već može da apstrahuje do nivoa entiteta i relacija. Realni projekti često nisu jednostavni i formulisanje problema na višem nivou

apstrakcije često može da bude od koristi. Na *Korma* se već donekle može gledati kao na sisteme koji rade objektno-relaciono mapiranje u jezicima koji podržavaju objektno programiranje kao što su *Ruby*, *Python* i *Java*.

Da biste koristili neku bazu podataka potrebno je da definišete njenu specifikaciju (istu onu kao za „clojure.java.jdbc“ opisanu u poglavlju 2.1.1.1). Kako bi olakšala ovaj deo, biblioteka pruža niz predefinisanih funkcija koje će vratiti konfiguracije za neke često korišćene baze podataka kao što su *PostgreSQL*, *MySQL*, *MSSQL*, *Oracle*, *SQLite* i *H2*. Potom da iskoristite „korma.db/defdb“ makro kako biste definisali bazu koju ćete dalje koristiti. Primer konfigurisanja jedne baze je dat u nastavku.

```
(use 'korma.db)

(def spec-baze (postgres {:make-pool? false}))

(defdb moja-baza spec-baze)
;; =>
;; {:pool
;;  {:classname "org.postgresql.Driver", :subprotocol "postgresql",
;;   :subname   "//localhost:5432/",     :make-pool?  false},
;;  :options
;;  {:naming          {:keys   #function[clojure.core/identity],
;;                     :fields #function[clojure.core/identity]},
;;   :delimiters      ["\"" "\""],
;;   :alias-delimiter " AS ",
;;   :subprotocol     "postgresql"}}
```

Dakle, funkcija „postgres“ je već dala neku standardnu specifikaciju za *PostgreSQL* i makro „defdb“ ju je iskoristio da napravi konstantu koju ćemo nadalje koristiti za sve upite.

U nastavku će biti dat primer u svrhu ilustracije koncepata koje donosi *Korma*. Za početak ćemo definisati entitete *korisnik, email, adresa, država, nalog* i *objava*.

```
(declare korisnik email adresa drzava nalog objava)

(defentity korisnik
  (pk :id) ; opciono, podrazumevano je :id
  (table :korisnik) ; opciono, podrazumevan je naziv entiteta
  (database moja-baza) ; opciono, poslednje definisana baza je podrazumevana
```

```
  (entity-fields :ime :prezime) ; podrazumevana polja za selektovanje

  ;; Mutacije

  ;; funkcija koja se uvek poziva pre snimanja u bazu
  ;; ovde menja prvo slovo prezimena da bude veliko
  (prepare (fn [{prezime :prezime :as v}]
             (if prezime
               (assoc v :prezime (str/upper-case prezime)) v)))
  ;; funkcija koja se primenjuje na svaki rezultat iz skupa rezultata
  ;; ovde menja prvo slovo imena da bude veliko
  (transform (fn [{ime :ime :as v}]
               (if ime
                 (assoc v :ime (str/capitalize ime)) v)))

  ;; Veze entiteta

  (has-one adresa)
  (has-many email)
  (belongs-to nalog)
  ;; zadaje se ime many to many tabele
  (many-to-many objava :korisnik_objava))

(defentity email
  (belongs-to korisnik))

(defentity adresa
  (pk :my_pk)
  (belongs-to korisnik)
  ;; dodavanje imena stranog kljuca, opciono
  (belongs-to drzava {:fk :id_drzava}))

(defentity drzava
  (has-many adresa))

(defentity nalog
  (has-one korisnik))

(defentity objava
  (many-to-many korisnik :korisnik_objava))
```

Sada možemo postavljati upite koji koriste definisane entitete.

```
(-> (select* korisnik)
    (where {:ime "pera"})
    (where {:prezime "peric"})
    (with adresa)
    (as-sql))
;; =>
;; SELECT "korisnik"."ime", "korisnik"."prezime", "adresa".* FROM "korisnik"
;; LEFT JOIN "adresa" ON "adresa"."korisnik_id" = "korisnik"."id"
;; WHERE ("korisnik"."ime" = ?) AND ("korisnik"."prezime" = ?)
```

Gore vidimo kakav će biti rezultujući *SQL* upit, a ako želimo zaista da izvršimo ovaj upit potrebno je da umesto funkcije „korma.core/as-sql“ koristimo funkciju „korma.core/exec“ (ili „korma.core/select“).

Ono što je prednost *Korma* u odnosu na *HoneySQL* i *SQLingvo* je svakako koncept entiteta i mogućnost definisanja upita preko njih. Treba primetiti i da *Korma* pruža mogućnost implicitnog spajanja tabela kroz makro „korma.core/with“. Ovaj koncept je moguć posredstvom entiteta i upravo predstavlja onaj deo koji *Korma* približava sistemima kao što su *Active record* ili *Django model*.

Ono što svakako nedostaje ovom JSD-u da bi dostigao nivo praktičnosti kakav imamo u popularnim programskim jezicima jeste mogućnost da iz definisanih entiteta i njihovih relacija generiše DDL (*Data definition language*) naredbe.

2.2.2 Proširenja SQL-a

Druga grupa jezika ima drugačiji pristup — ne proširuje (direktno) *Clojure* već sam *SQL* konceptima koji su potrebni kako bi se iz SQL-a mogle napraviti *Clojure* funkcije.

Ako korisnik ima potrebu da piše *SQL* naredbe, možda mu je prirodnije da ih piše direktno nego posredstvom internih JSD. Sa druge strane, izvedba samog rešenja koje bi proširilo *SQL* svakako je (barem u početku) jednostavnija jer nema potrebu da formuliše sve koncepte koje *SQL* ima već ih može koristiti direktno (*nativno*).

Kada govorimo o implementaciji npr. *Korma*, moramo imati u vidu da su svi makroi iz „korma.core“ (kod *SQLingvo* su to funkcije) morali negde biti implementirani. Taj prostor imena sadrži i makroe kao što su „where“, „join“, „order“, „limit“. Svi u sebi sadrže korak koji ih prevodi na izvorne koncepte *SQL-a*. Sa konceptualnog stanovišta to nije neophodno — ti koncepti već postoje u samom *SQL-u*. Kada direktno pišemo *SQL* onda ovaj među korak nije potreban.

Opet, ne bi bilo dobro na račun pisanja *nativnog SQL-a* izgubiti fleksibilnost i mogućnosti *Clojure-a* kao programskog jezika u formiranju naredbi koje šaljemo bazama podataka. U narednim poglavljima će biti predstavljene biblioteke koje nastoje da korisnicima omoguće direktno pisanje *SQL-a* bez gubitka fleksibilnosti na koju su navikli u samom *Clojure-u*.

2.2.2.1 Yesql

Osnovno stanovište ove biblioteke je da je *Clojure* odličan za pisanje internih JSD, ali da za ovu namenu već postoji jedan poprilično stabilan i proveren - *SQL. S-izrazi*

su odlični, ali u ovom konkretnom slučaju ne donose nikakav benefit. Biblioteka kao rešenje nudi da *SQL* ostane to što jeste i tu gde jeste — u zasebnim fajlovima. Ti fajlovi se kasnije čitaju i na osnovu njih se prave funkcije sa kojima korisnik dalje u Lisp okruženju može da radi sve što želi.

Iako rešenja sa internim JSD koji se prevode na *SQL* na prvi pogled deluju kao nešto što se savršeno i bespogovorno uklapa u filozofiju Lisp-a, to i nije baš tako. Naime, Lisp je po svojoj prirodi sklon programiranju na visokom nivou apstrakcije (što nikako ne znači nužno i gubitak performansi). Programirati na visokom nivou apstrakcije obično znači i uže se koncentrisati na osnovni problem koji je ispred. Kada se razvija neki sistem koji se izvršava u *Web* okruženju, „mehanika“ komunikacije sa bazom podataka (*SQL*) često nije osnovni problem. Opisivanje komunikacije na nivou *SQL* naredbi udaljava programera od osnovnog problema — što je upravo suprotno od filozofije Lisp-a. Naravno, komunikacija sa bazom nije osnovni problem sve dok to ne postane. Kada se to desi, potrebno je staviti na raspolaganje sve alate koji mogu biti od pomoći. To je upravo ono što radi *Yesql* — drži *SQL* u zasebnim fajlovima, pruža ga na korišćenje kroz funkcije i daje sve alate koji dolaze sa samim *SQL-om*. To se već uklapa u koncept programiranja na visokom nivou apstrakcije, kao i funkcionalne paradigme.

Dakle, prvo je potrebno napraviti *SQL* fajl u kome će se pisati standardni kod. Jedini dodatak koji *Yesql* zahteva je komentar iznad *SQL* naredbe koji opisuje funkciju kojom će ta naredba biti apstrahovana. Na primer, napraviti fajl „moj_sql.sql“ sa sadržajem:

```
-- name: users-by-country
SELECT *
FROM korisnik
WHERE grad = :grad
```

Potom učitati taj fajl kroz funkciju „yesql.core/defqueries“ (uz prosleđivanje specifikacije baze podataka u stilu „clojure.java.jdbc“):

```
(defqueries "putanja/do/moj_sql.sql"
   {:connection specifikacija-baze})
```

I potom u *Clojure-u* dobijamo mogućnost korišćenja funkcije koju smo definisali u *SQL* fajlu.

```
(users-by-country {:grad "Novi Sad"})
;; => ({:ime "Pera" :prezime "Peric" :grad "Novi Sad"} ...)
```

Kada se *Clojure* i *SQL* drže u odvojenim fajlovima tada korinik nikada ne mora da postavlja sebi pitanje — Koji izraz treba da upotrebim u ovom JSD da bih dobio ovakav SQL?. Uvek može da piše ono što želi i to ne mora da radi kroz posebne funkcije („raw-sql“ funkcija je način na koji se to obično radi u internim JSD). Time dobije mogućnost korišćenja standardnih editora i alata za pisanje *SQL-a*. Ovo sve omogućuje korisniku da rade lakše popravljanje performansi svojih upita kada za to dođe vreme.

Kada se govori o radu u velikim timovima gde postoje ljudi koji se isključivo bave bazama podataka onda je pristup sa zasebnim fajlovima mnogo adekvatniji. *SQL* fajlovi se lako mogu deliti među programerima i inžinjerima baza podataka. Takođe, običaj je da se ovakav *SQL* deli između timova koji se bave razvojem različitih delova sistema, a koji koriste bazu podataka za koju je kod pisan. Na ovaj način se postiže ponovno korišćenje onih upita koje se pokažu da imaju najbolje performanse.

Kada Yesql nije najbolje rešenje?

Kada imamo potrebu da naš *SQL* radi sa više vrsta baza podataka od jednom. Ako postoji potreba da neki upit bude upotrebljiv u više različitih dijalekata *SQL-a* onda *Yesql* svakako nije rešenje — već je potrebno, kao i obično, uvesti novi nivo apstrakcije iznad *SQL-a*.

Yesql pristup je inspirisao slična rešenja i u mnogim drugim programskim jezicima. Tako postoje biblioteke sa istim ciljem za programske jezike: *Go*, *Erlang*, *Python*, *Ruby*, *JavaScript*, *C#* i *PHP*.

2.2.2.2 HugSQL

HugSQL je biblioteka koja proklamuje isti koncept kao i *Yesql* sa tim što pruža neke nove mogućnosti koje je čine fleksibilnijom. *HugSQL* će biti korišćen u ovom radu. U nastavku je dat princip korišćenja i opis nekih osnovnih koncepata. Primeri će biti tati za *H2* bazu podataka, ali *HugSQL* radi sa svim bazama podataka za koje postoji *JDBC* drajver.

Da bismo kreirali bazu možemo napisati *SQL* fajl koji sadrži:

```
-- :name napravi-covek-tabelu
-- :command :execute
```

```
-- :result :raw
-- :doc Kreira tabelu covek u bazi podataka
create table covek (
id            integer auto_increment primary key,
ime           varchar(40),
specijalnost  varchar(40),
created_at    timestamp not null default current_timestamp
)
```

Svaka *SQL* naredba mora biti napisana tako da se poštuje sledeće:

- mora imati komentar iznad sebe
- taj komentar mora imati barem jednu liniju koja sadrži ključnu reč „:name“ iza koje sledi ime rezultujuće funkcije
- komentar može sadržati liniju koja počinje ključnom rečju „:doc“ iza koje sledi dokumentacija rezultujuće funkcije.
- komentar može sadržati liniju koja počinje ključnom rečju „:command“ i iza nje sledi jedna od vrednosti:
  - „:query“ ili „:?“ (podrazumevano) — običan upit sa skupom rezultata.
  - „:execute“ ili „:!“ — bilo koja naredba
  - „:returning-execute“ ili „:<!“ — predstavlja podršku za konstrukcije tipa „INSERT ... RETURNING“
  - „:insert“ ili „:i!“ — podrška za „INSERT“ i *JDBC* „.getGeneratedKeys“.

  „:query“ i „:execute“ zapravo služe da signaliziraju „clojure.java.jdbc“ da li se radi o njegovoj „query“ ili „execute!“ funkciji, ili kod „clojure.jdbc“ da li se radi o „fetch“ ili „execute“.
- komentar može sadržati liniju koja počinje ključnom rečju „:result“ i iza nje sledi jedna od vrednosti:
  - „:one“ ili „:1“ — vraća se jedna vrednost kao *Clojure* mapa.
  - „many“ ili „:*“ — vraća se više rezultata kao vektor mapa
  - „:afected“ ili „:n“ — broj izmenjenih redova u tabeli (koristi se za „INSERT“, „UPDATE“ i „DELETE“)

  - :raw (podrazumevano) — vraća netaknut skup rezultata kakav je dobijen od drajvera.

- ako se koriste skraćene verzije „:result“ i „:command“ onda se mogu pisati i u komentaru koji počinje sa „:name“ nakon imena funkcije. To bi za primer iznad značilo:

```
-- :name napravi-covek-tabelu :!
```

- unutar samog *SQL-a* se mogu nalaziti parametri koji će se prevesti u prarametre funkcije.

Parametri koji se mogu naći u samom *SQL* kodu su ili u formatu „:ime-parametra“ ili u formatu „:tip-parametra:ime-parametra“. Oni se u vreme izvršavanja zamenjuju sa odgovarajućim vrednostima iz mape koja se prosleđuje kao poslednji parametar funkciji koju *HugSQL* generiše. *HugSQL* podržava parametre koji predstavljaju vrednosti, identifikatore i *SQL* ključne reči. Takođe je podržan i mehanizam kojim se mogu napraviti novi tipovi parametara. U nastavku će biti opisani oni koji su podrazumevano podržani.

- **Vrednosti** — „:value“ ili „:v“, podrazumevano
  Vrednosti se u vreme izvršavanja zamenjuju sa odgovarajućim *SQL* tipovima podataka. Konverzija iz *Clojure* u *SQL* tipova podataka je ostavljena u nadležnost drajveru baze podataka koji se nalazi ispod koristeći *SQL* vektor format. Primer je dat u nastavku.

```
-- :name dodaj-coveka :! :n
insert into covek (ime, specijalnost)
values (:ime, :specijalnost)

-- :name selektuj-coveka :? :*
select * from covek where id = :v:id
```

Ove funkcije se mogu koristiti iz *Clojure* na sledeći način:

```
(dodaj-coveka db {:ime "Pera" :specijalnost "moler"})
;; => 1

(selektuj-coveka db {:id 1})
```

```
;; =>
;; ({:id 1, :ime "Pera", :specijalnost "moler",
;;   :created_at #inst "2016-08-27T13:47:46.886000000-00:00"})
```

Funkcija kojom se može dobiti *SQL* vektor koji će biti poslat drajveru može biti generisana korišćenjem makroa „hugsql.core/def-sqlvec-fns“ na sledeći način:

```
(hugsql/def-sqlvec-fns "putanja/do/ime.sql")
```

Sama funkcija se koristi na sledeći način:

```
(dodaj-coveka-sqlvec {:ime "Pera" :specijalnost "moler"})
;; =>
;; ["insert into covek (ime, specijalnost)\nvalues ( ? , ? )"
;;  "Pera" "moler"]
```

- **Liste vrednosti** — „:value*“ ili „:v*“
  Slični su sa kao vrednosti ali rade sa listama vrednosti koje su potrebne za *SQL* naredbe koje sadrže „in (...)“. Predstavljaju sekvencu od nula ili više elemenata koji su vrednosti. Svaki element se koristi kao pojedinačna vrednost, a lista se objedinjuje umetanjem zareza.

```
-- :name selektuj-ljude-sa-imenima
select * from covek where ime in (:v*:imena)
```

Potom to funkciju možemo koristiti na sledeći način:

```
(selektuj-ljude-sa-imenima db {:imena ["Pera" "Mika"]})
;; =>
;; ({:id 1, :ime "Pera", :specijalnost "moler",
;;   :created_at #inst "2016-08-27T13:47:46.886000000-00:00"}
;;  {:id 2, :ime "Mika", :specijalnost "stolar",
;;   :created_at #inst "2016-08-27T15:16:06.977000000-00:00"})
```

- **Ntorke (*Tuples*)** — „:tuple“
  Slični su kao liste sa tim što se vrednosti prosleđene kao ntorka u rezultujućem *SQL* pretvaraju u niz razdvojen zarezima oko koga su obične zagrade.

```
(ntorka-param-sqlvec {:id-ime [1 "Pera"]})
;; => ["select * from covek\nwhere (id, ime) = (?,?)" 1 "Pera"]
```

- **Liste ntorki** — „:tuple*“
  Svaki element ove liste se tretira kao ntorka, one su u rezultujućen *SQL* međusobno razdvojene zarezima.

```
(lista-torki-sqlvec {:ljudi [[1 "Pera"] [2 "Mika"] [3 "Zika"]]})
;; ["insert into covek (id, ime)\nvalues (?,?),(?,?),(?,?)"
;;  1 "Pera" 2 "Mika" 3 "Zika"]
```

- **Identifikatori** — „:identifier“ ili „:i“
  Parametri ovog tipa se u vreme izvršavanja zamenjuju sa *optionally-quoted SQL* identifikatorima. Moguće strategije navođenja su: „:ansi“, „:mysql“, „:mssql“ i „off“. One se navode pri definisanju *Clojure* funkcija na osnovu *SQL-a*. Identifikatori koji sadrže tačku će biti razdvojeni po njoj, navedeni individualno i potom ponovo spojeni.

- **Liste identifikatora** — „:identifier*“ ili „:i*“
  Logički su slične sa listama vrednosti, sa tim što su elementi identifikatori. Mogu se koristiti da objedine imena kolona u „SELECT“, „GROUP BY“, „ORDER BY“ klauzulama. Na primer, ako je potrebno da rezultujuća *Clojure* funkcija prima kolone koje želimo da izdvojimo to možemo uraditi na sledeći način.

```
-- :name selektuj-sa-imenima-i-prezimenima :? :*
select :i*:imena-kolona, count(*) as broj
from covek
group by :i*:imena-kolona
order by :i*:imena-kolona
```

  Onda rezultujuću funkciju možemo koristiti na sledeći način:

```
(selektuj-sa-imenima-i-prezimenima
 db {:imena-kolona ["ime", "specijalnost"]})
;; =>
;; ({:ime "Marko", :specijalnost "zidar", :broj 2}
;;  {:ime "Mika", :specijalnost "stolar", :broj 3}
;;  {:ime "Pera", :specijalnost "moler", :broj 1})
```

- **Bilo kakav SQL** — „:sql“
  Ovi parametri se zamenjuju *SQL-om*. Onakav kakav string primi rezultujuća

funkcija, takav *SQL* će se naći na mestu ovakvih parametara. Treba voditi računa da je korišćenje ovakvih parametara zahteva dodatne validacije kako bi se izbegao *SQL injection*.

Nakon što korektno definišemo *SQL* naredbu, potrebno je da pozovemo funkciju koja će pretvoriti to u *Clojure* funkcije:

```
(require '[hugsql.core :as hugsql])

(hugsql/def-db-fns "putanja/do/ime.sql"
  {:quoting :strategija-navodjenja})
```

i od tog momenta možemo da koristimo dobijene funkcije. Za našu funkciju sa početka mogli bi da uradimo sledeće:

```
(create-covek-table specifikacija-baze)
```

HugSQL ima još jednu pogodnost koja je od koristi u ovom radu — moguće je ubaciti *Clojure* izraze unutar komentara u *SQL* kodu. Taj kod biće izvršen neposredno nakon početnog parsiranja ulaznih parametara rezultujuće funkcije.

Clojure izrazi (*S-izrazi*) unutar *SQL-a* moraju da vraćaju neki string ili „nil“. Mogu da koriste dva simbola koja su im dostupna u vreme izvršavanja: „params“ (mapa parametara) i „options“ (mapa opcija).

Komentari u jednoj liniji moraju da počnu sa „– “ i u tom slučaju ceo *Clojure* izraz mora biti napisan u toj jednoj liniji.

```
-- :name primer-komentar-u-jednoj-liniji :? :1
select
--~ (if (seq (:kolone params)) ":i*:kolone" "*")
from covek
order by id
```

Mogući su i komentari u više linija i linje koje ih sadrže moraju početi sa „/* “ i završavati se sa „ */“. Kada želimo da naznačimo da će se neki izraz nastaviti u nekoj narednoj liniji, nakon *SQL-a*, onda je potrebno da koristimo jednu liniju sa praznim komentarom. Primer ovog slučaja dat je u nastavku.

```
-- :name primer-komentara-u-vise-linija :? :1
select
/*~ (if (seq (:kolone params)) */
:i*:kolone
/*~*/
*
/*~ ) ~*/
from covek
order by id
```

U ovakvom kodu moguće je zahtevati neke *Clojure* biblioteke. Za to se koriste klasični *SQL* komentari u više linija. U njima je potrebno navesti „:requre“ ključnu reč i iza nje vektore koji opisuju zahtevanje biblioteka, isto ka što se inače radi u „ns“ makrou u *Clojure-u*.

```
-- :name genericki-update :! :n
/* :require [clojure.string :as string]
            [hugsql.parameters :refer [identifier-param-quote]] */
update :i:tabela set
/*~
(string/join ","
  (for [[polje _] (:azuriranja params)]
    (str (identifier-param-quote (name polje) options)
      " = :v:azuriranja." (name polje))))
~*/
where id = :id
```

2.3 Mesto *Larve* u *Clojure* ekosistemu

Jedan od problema sa *Clojure* ekosistemom je započinjanje projekata. Ovaj problem je posebno istaknut kada se radi o manje ili više klasičnim aplikacijama koje koriste relacione baze podataka. Ako bismo želeli relativno brzo da započnemo neki novi projekat ovakvog tipa za početak bi trebalo da se odlučimo za jedno od rešenja koja su predstavljena u ovom poglavlju. Opet, ako je naš najveći kriterijum brzina razvoja onda bi verovatno logična odluka bila da uzmemo *Korma* jer ona ide najdalje prema praktičnosti *ORM* sistema u objektno orijentisanom programiranju. Ako bismo na primer poredili *Korma* sa *Django model* okruženjem primetili bi neke razlike

koje bi mogle činiti da naše vreme razvoja ne bude u okvirima onoga što pruža *Django model*. Neki od tih problema bi mogli biti:

1. *Korma* nam ne omogućuje da definišemo model svog sistema na nivou podataka i iz toga dobijemo početnu šemu baze podataka. Morali bismo sami da napravimo šemu baze podataka i onda u *Korma* notaciji da definišemo entitete i upite nad njima.

   Koristeći *Django model* okruženje bismo mogli da napravimo svoju model klasu:

   ```
   class Project(models.Model):
       title = models.CharField(max_length=50, unique=True)
       description = models.CharField(max_length=255, default="')
       users = models.ManyToManyField(User)
   ```

   Potom bi pokrenuli odgovarajuću *Django* komandu:

   ```
   manage.py migrate
   ```

   Nakon toga bi naša baza podataka već sadržala šemu tabele „Project“ (i svih ostalih entiteta koje smo opisali objektima). Od tog momenta pa nadalje sve što radimo je klasično rukovanje objektima. Na primer, ako želimo da dobijemo sve korisnike sa projekta mogli bismo da uradimo sledeće:

   ```
   project1.users.all()
   ```

   Ovo se čini elegantnim mehanizmom.

   > ***Programiramo objektno, entitete podataka posmatramo kao objekte, njima rukujemo kao objektima.***

2. Kada koristimo *Korma* nivo apstrahovanja koji možemo da postignemo je niži.

   > ***Iako pišemo Clojure, entitete opisujemo konceptima koji donekle liče na objekte, a same upite mešavinom koncepta entiteta i proceduralnosti iz SQL-a.***

Pošto *Clojure* naglašava predstavljanje podataka u okviru osnovnih struktura podataka i funkcionalno programiranje bilo bi prirodnije da možemo da opišemo svoje podatke preko neke strukture podataka i da nakon toga dobijemo funkcije koje su adekvatne za rukovanje tim podacima. Ono što bismo želeli je da:

Opisujemo podatke u okviru osnovnih Clojure struktura podataka, koristimo ih kroz funkcije.

Uz to se još treba osvrnuti na zaključke iz poglavlja 2.2.2.1 — *SQL* je već sam po sebi dovoljno stabilan JSD za svoj domen, nije loše držati ga odvojenim fajlovima.

Ono što ***Larva*** radi je upravo rezultat navedenih zaključaka. Ono što nam ona omogućuje je da:

Opišemo podatke kao Clojure mape i dobijemo iz toga funkcije za rad nad njima i DDL kod koji će izvršiti uspostavljanje šeme baze na osnovu opisanih podataka.

Glava 3

Implementacija

Larva je izvedena kao kod generator. Ulaz u taj generator je opis podataka izražen u internom *Clojure* JSD, a izlaz su funkcije koje rukuju opisanim podacima i *DDL* kod koji uspostavlja odgovarajuću šemu baze podataka.

U ovom poglavlju će biti opisana kompletna ideja ***Larva*** generatora kao i način njegove implementacije. Pregled koncepata će se odvijati od JSD kojim se opisuju podaci prema načinu korišćenja funkcija koje se na kraju dobijaju.

3.1 *Larva* JSD

Larva JSD je prost *Clojure* interni JSD. Zapravo, on predstavlja jednu *Clojure* perzistentnu mapu u kojoj su ključevi i njihove vrednosti unapred opisane i pre generisanja koda se validiraju. *Clojure* ekosistem ima nekoliko biblioteka koje bi mogle biti od koristi pri validiranju pomenute mape. Najsveobuhvatnija među njima je „plumatic/schema“ koja će biti upotrebljena u inicijalnoj implementaciji ovog rada.

Clojure će od verzije 1.9 (koja je trenutno u alfa izdanju) predstaviti „clojure.spec“ sistem koji će potom, u okvirima Larva sistema, biti razmatran kao zamena za „plumatic/schema“.

Sam Larva JSD koji je tehnički *Clojure* perzistentna mapa se potom pretvara u usmereni graf koji služi kao osnova za donošenje zaključaka potrebnih za generisanje koda.

3.1.1 Meta model Larva DSL i plumatic/schema

„plumatic.schema“ je *Clojure* i *ClojureScript* biblioteka za deklarativno opisivanje podataka i validaciju.

Clojure je u osnovi dinamički tipiziran programski jezik. To u ovom praktičnom slučaju znači da nema ugrađeni sistem tipova koji bi nam mogao pomoći da eventualno validiramo neke delove Larva JSD u vreme kompajliranja. Ono što nam donosi „plumatic.schema“ je da opišemo kog su oblika podaci koji ulaze u neku funkciju i kog su oblika oni koji iz nje izlaze. Ako neki poziv ove funkcije ne zadovoljava osobine opisane šemama, „plumatic.schema“ će baciti grešku koja opisuje koji deo prosleđenih podataka ne odgovara definisanoj šemu za funkciju. Tako funkciju koja prevodi Larva JSD u graf možemo definisati na sledeći način:

```
(s/defn ^:always-validate ->graph :- ubergraph.core.Ubergraph
  "Transforms standard application program (uni-model)
   to its graph representation."
  [program :- Program]
  ...)
```

Listing 1: Funkcija „larva.graph/->graph“.

I time ćemo postići da ovoj funkciji ne može biti prosleđena ni jedna struktura koja ne zadovoljava pravila Larva DSL-a opisanog u šemi „Program“.

Sama „Program“ šema je opisana kroz podšeme. Krenućemo od same „Program“ šeme pa ćemo je potom redom razlagati na delove (*top-down*). U nastavku je izgled program šeme.

```
(s/def Program
  "Schema for standard app program."
  {(s/optional-key :about) About
   (s/optional-key :meta)  Meta
   :entities               Entities})
```

Ova šema govori da je „Program“ mapa koja ima ključ „:entities“ čija je vrednost šema „Entities“, opciono ima ključ „:about“ čija je vrednost šema „About“ i opciono ključ „:meta“ čija je vrednost šema „Meta“.

- šema **Meta** je data na sledeći način:

  ```
  (s/def Meta
    "Schema for meta section of program."
    {(s/optional-key :db) {:type DBTypes
                           (s/optional-key :sql) SQLTools}})
  ```

Znači, ova šema je mapa koja samo jedan opcioni ključ (može biti i prazna) „:db“ koja konkretna baza podataka je ciljna. Vrednost ovog ključa je nova mapa koja ima ključ „:type“ čija vrednost je šema „DBTypes“ koja je data kao:

```
(s/def DBTypes
  (s/enum :postgres :mysql :h2 :sqlite :mongodb))
```

Dakle, ta šema predstavlja jednu od reči navedenih u „s/enum“ *S-izrazu*.

Drugi, opcioni, ključ „:sql“ mape „:db“ je dat kao:

```
(s/def SQLTools
  (s/enum :hugsql :yesql))
```

Dakle, jedna od dve navedene ključne reči.

- Šema **About** je data na sledeći način:

```
(s/def About
  "Schema for about section of program."
  {:name                    s/Str
   (s/optional-key :author)  s/Str
   (s/optional-key :comment) s/Str})
```

Znači, ona je mapa koja ima obavezan ključ „:name“ (ime programa) čija je vrednost string i dva opciona ključa „:author“ (autor) i „:comment“ (komentar) čije su vrednosti takođe stringovi.

- Šema **Entities** je data kao:

```
(s/def Entities
  [Entity])
```

Dakle, kao lista jednog ili više entiteta koji su dati šemom „Entity“.

```
(s/def Entity
  {:signature               s/Str
   :properties              Properties
   (s/optional-key :plural) s/Str})
```

Koja predstavlja mapu sa ključem „:signature“ koji je string, opcionim ključem „:plural“ koji je takođe string i ključem „:properties“ čija je vrednost je pema „Properties“.

```
(s/def Properties
  [Property])
```

Ova šema dakle predstavlja listu nula ili više članova koji su svi dati šemom „Property“.

```
(s/def Property
  {:name                    s/Str
   (s/optional-key :type)   PropertyDataType)
```

Svaki „Property“ je mapa sa obaveznim ključem „:ime“ koji je string i opcionim ključem „:type“ koji je dat šemom „PropertyDataType“.

```
(s/def PropertyDataType
  (s/conditional
   keyword? SimpleDataType
   string? CustomDataType
   #(and (map? %) (= 1 (count %))) Collection
   :else SomethingWithReference))
```

Ova konstrukcija znači sledeće — ako je vrednost na mestu „PropertyDataType“ ključna reč, onda ona mora zadovoljiti šemu „SimpleDataType“.

```
(s/def SimpleDataType
  (s/enum :str :text
          :num :float :bignum
          :datetime :date :timestamp
          :bool :geo :json :binary :pass))
```

Što znači da mora biti jedna od navedenih ključnih reči u „s/enum“ izrazu.

Ako je string onda mora zadovoljiti šemu „CustomDataType“, koja samo govori da je vrednost string.

```
(s/def CustomDataType s/Str)
```

Ako je mapa dužine jedan onda mora zadovoljiti šemu „Collection“.

```
(s/def Collection
  "Simple collection property."
  {:coll DataType})
```

Ono što je na mestu „Collection“ mora biti mapa koja ima ključ „:coll“ sa vrednošću „DataType“.

```
(s/def DataType
  (s/cond-pre SimpleDataType CustomDataType))
```

Što govori da mora biti zadovoljiti jednu od šema „SimpleDataType“ i „CustomDataType“ koje su prethodno opisane.

Konačno, ako je na mestu „PropertyDataType“ nešto što ne zadovoljava ništa od prethodnog onda to mora zadovoljiti šemu „SomethingWithReference“.

```
(s/def SomethingWithReference
  "Something which refers to collection of entities
   or single entity determined by :signature."
  (s/conditional
   #(contains? % :coll) CollectionWithReference
   :else ReferenceToSingleEntity))
```

Ova šema govori da ako se na njenom mestu nalazi mapa koja sadrži ljuč „:coll“ onda ona mora zadovoljiti šemu „CollectionWithReference“.

```
(s/def CollectionWithReference
  {:coll (s/enum :reference)
   :to   SignatureProperty})
```

Tačnije mapu koja ima ključ čija vrednost je ključna reč „:reference“ i ključ „:to“ čija je vrednost šema „SignatureProperty“.

```
(s/def SignatureProperty
  (s/conditional
   #(or (= (count %) 2) (= (count %) 1))
   [(s/one s/Str "signature") (s/optional s/Str "property")]))
```

Dakle, mora biti *Clojure* perzistentni vektor dužine jedan ili dva gde će prva vrednost biti string koji predstavlja potpis entiteta a drugi njegovu osobinu.

Ako vrednost na mestu „SomethingWithReference“ ne sadrži ključ „:coll“ onda mora zadovoljiti šemu „ReferenceToSingleEntity“.

```
(s/def ReferenceToSingleEntity
  {:one (s/enum :reference)
   :to  SignatureProperty})
```

Što znači da mora biti mapa koja ima ključ „:one“ sa vrednošću „:reference“ i ključ „:to“ čija vrednost mora zadovoljiti gorenavedenu šemu „SignatureProperty“.

Kada znam kako je definisan meta model Larva JSD u terminima „plumatic/schema“ šema, možemo dati jedan primer opisa entiteta u Larva JSD.

```
{:about
 {:name    "Pilot model"
  :author  "Pera Peric"
  :comment "Model muzickog festivala."}
 :entities
 [{:signature  "Muzicar"
   :properties [{:name "ime" :type :str}
                {:name "prezime" :type :str}
                {:name "nadimak" :type :str}
                {:name "priznanja" :type {:coll :str}}
                {:name "bend" :type {:one :reference
                                     :to  ["Bend"]}}]}
  {:signature  "Bend"
   :properties [{:name "ime" :type :str}
                {:name "zanr" :type :str}
                {:name "velicina" :type :str}
                {:name      "clanovi" :type {:coll :reference
                                             :to   ["Muzicar" "bend"]}}]}
  {:signature  "Festival"
   :properties [{:name "ime" :type :str}
                {:name "lokacija" :type :geo}]}]}
```

Listing 2: Primer opisa podataka u Larva JSD.

3.1.2 Graf reprezentacija i Ubergraph

Videli smo da funkcija „->graph“ iz listinga 1 vraća „Ubergraph“. *Ubergraph* je svestrana struktura tipa grafa za opštu upotrebu. Pružena je kroz istoimenu biblioteku[23]. Predstavlja dopunu i proširenje *Loom*[16] biblioteke koja predstavlja kolekciju graf protokola i algoritama. Pored toga što nam pruža mogućnost uspostavljanja graf struktura podataka daje nam i mogućnost grafičkog predstavljanja tih grafova kroz *Graphviz*[13] biblioteku.

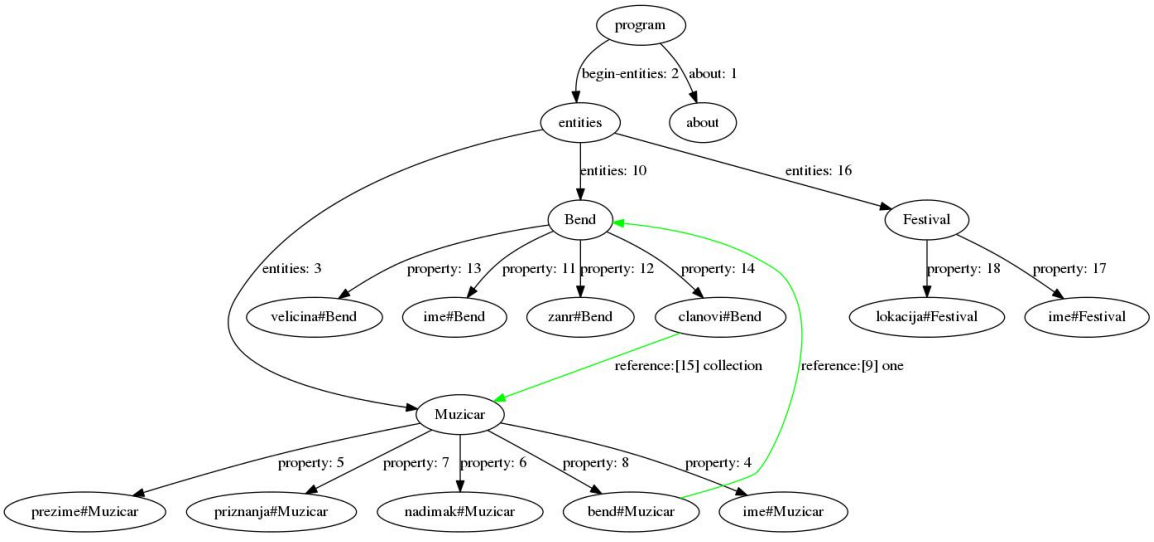
Ova mogućnost *Ubergraph-a* nam omogućuje da svoje entitete predstavljene kroz Larva JSD iscrtavamo kao grafove (stabla) i time dobijemo solidnu vizuelizaciju na kojoj možemo da uvidimo veze između entiteta.

Ako želimo da predstavimo naš primer iz listinga 2 to možemo da uradimo koristeći funkciju „ubergraph.core/viz-graph“. Dakle, prvo svoj model „festival“ prosledimo funkciji koja to pretvori u graf a potom graf vizualizujemo preko *Graphviz-a*.

```
(require '[ubergraph.core :as u])

(u/viz-graph (->graph festival))
```

Rezultat ove funkcije je sledeća slika.



Slika 3.1: Graf prikaz primera iz listinga 2.

Na slici vidimo da postoji korenski čvor koji uvek ima naziv „program“. Taj čvor ima dva direktno podređena: „entities“ koji predstavlja korenski čvor svih entiteta i „about“ koji predstavlja sekciju sa informacijama o samom modelu.

Dalje možemo primetiti da je svaki entitet predstavljen jednim čvorom koji je označen potpisom („:signature“) samog entiteta. Svaki entitetski čvor ima više podređenih gde svaki od njih predstavlja jednu osobinu entiteta. Čvorovi osobina imaju nazive u formatu „ime_osobine#potpis_entiteta“.

Svaka ivica u ovom grafu ima jedinstveni identifikator u vidu broja. Ovi brojevi rastu po vremenu nastanka čvora — čvorovi koji potiču od entiteta koji su navedeni pre imaju ulazne ivice označene manjim brojevima. Ova osobina grafa daje informaciju o tome kojim su redom navođene stvari u okviru Larva JSD. Pošto je čitava struktura formulisana u Larva JSD prevodi u stablo nema potrebe za deklarisanjem potpisa entiteta unapred kao što je to slučaj sa *Korma* sistemom iz poglavlja 2.2.1.3 u kojem se u ovakvim situacijama koristi „declare“ makro kako bi zauzela korišćena imena unapred.

Ulazne ivice mogu imati sledeće formate:

- **begin_entities: jedinstven_broj_ivice** — ivica koja ide od čvora „program“ ka čvoru „entities“.
- **about: jedinstven_broj_ivice** — ivica koja ide od čvora „program“ do čvora „about“.
- **entities: jedinstven_broj_ivice** — ivice koje idu od „entities“ čvora ka konkretnim entitetima.
- **property: jedinstven_broj_ivice** — ivice koje povezuju konkretne entitete sa njihovim osobinama.
- **reference:[jedinstven_broj_ivice] „one“ ili „collection“** — ivice koje prikazuju reference jednog entiteta na neki drugi (ili samog sebe). Reč „one“ se nalazi na kraju onih koje predstavljaju referencu na jednu instancu nekog entiteta, „collection“ na kraju onih koje predstavljaju referencu na više instanci nekog entiteta.

Funkcije koje služe za obilazak grafa i donošenje zaključaka potrebnih za generisanje koda se nalaze u prostoru imena „larva.program-api“. Tako na primer imamo funkciju „property-reference“ koja je data na sledeći način:

```
(s/defn ^:always-validate property-reference :- APIPropertyReference
  "..."
  ([entity :- s/Str property :- APIProperty]
```

```
  (get-property-reference (model->program) entity property))
 ([entity :- s/Str property :- APIProperty
   {:keys [model-path model] :as model-options}]
  (get-property-reference (resolve-program model-options)
                          entity property)))
```

Ova funkcija vraća mapu koja se sastoji od ključa koji predstavlja tip relacije čija je vrednost potpis entiteta koji je referenciran, eventualno osobinu tog referenciranog entiteta koja referencira nazad polazni entitet i indikator da li je veza rekurzivna. Vidimo da funkcija ima dva načina pozivanja. Prvi je uz prosleđivanje entiteta i osobine — pri tome će korišteni model biti automatski dobavljen pretragom u seledećem redosledu:

- koristi se „larva.program-api/program-model“ *Clojure* atom ako je postavljen.
- ako nije postavljen učitava se model sa podrazumevane lokacije unutar projekta — „larva_src/larva.clj“.

Drugi način za pozivanje ove funkcije je uz prosleđivanje entiteta, osobine i mape koja može sadržati ključeve „:model-path“ i „:model“. Ako je „:model“ istinitosno tačna vrednost onda će ona biti korišćena kao model. Ako „:model“ to nije i ako je „:model-path“ istinitosno tačna vrednost onda će model sa zadate putanje biti korišten.

Dakle, ako primenimo opisanu funkciju na listing 2 za entitet „Muzicar“ i njegovu osobinu „bend“ dobićemo seldeće:

```
(require '[larva.program-api :as api])

(let [osobina {:name "bend" :type {:one :reference
                                   :to  ["Bend"]}}]
     (api/property-reference "Muzicar" osobina))
;; => {:one-to-many "Bend", :back-property "clanovi"}
```

Ovo znači da su entiteti „Muzicar“ i „Bend“ povezani „jedan na više“ relacijom preko osobine „bend“ kod muzičara i osobine „clanovi“ kod benda.

U prostoru imena „larva.program-api“ nalaze se i ostale funkcije koje koriste drugi slojevi sistema. Iznad sloja koji predstavlja grupu funkcija za rukovanje internom reprezentacijom modela nalazi se sloj za generisanje koda.

3.1.3 Generisanje početnih migracija

Svi nivoi generisanja koda su međusobno nezavisni. Jedini deljeni resurs među njima je sam model. Svi koriste model kroz internu reprezentaciju (graf) i ona je sama imutabilna (što je objašnjeno u poglavlju 1.5.1). Dakle, svi nivoi generisanja koda mogu biti paralelizovani na trivijalan način — korišćenjem samo ugrađenih *Clojure* alata za konkurentno programiranje. U daljem tekstu ćemo navoditi nivoe generisanja koda po logičkom redosledu koji se ne mora poklapati sa fizičkim redosledom.

Ulazna funkcija u generator za *Luminus* okruženje je

`larva.frameworks.luminus.build/make`

Ona prima ključ „:model“ pod kojim se navodi opis u Larva JSD i ostale parametre.

U ovom poglavlju će biti opisan sloj generatora koji obezbeđuje da korisnik dobije pripremljene početne migracije koje uspostavljaju željenu šemu baze podataka.

Funkcija koja generiše sav *SQL* kod je:

`larva.frameworks.luminus.build/add-database-layer`

Ona je u osnovi funkcija koja obilazi graf interne reprezentacije modela i kroz prolaze popunjava međustrukturu koja se pretvara u kod. Pretvaranje međustrukture u kod se dešava na kraju ovog procesa.

Pre nego što počnemo generisati kod migracija potrebno je da zaključimo za koju konkretno bazu podataka treba da pravimo migracije. Za ovo nam služi funkcija „larva.db.utils/infer-db-type“. Ona će vratiti jednu od ključnih reči: „:postgres“, „:mysql“, „:h2“, „:sqlite“. Konkretna baza podataka se zaključuje u sledećem redosledu:

- Ako je u *meta* sekciji modela naveden ključ „:type“ sa nekom od mogućih vrednosti, ta ključna reč će biti vraćena.
- Ako u *meta* sekciji nije naveden tip baze podataka onda se obradom „project.clj“ fajla u korenu korisničkog projekta zaključi koje drajvere baze podataka je dobavio u svoj projekat i na osnovu toga se vrati odgovarajuća ključna reč.

Funkcija koja se bavi generisanjem konkretnih naredbi za definisanje šema tabela u bazi podataka je „larva.db.tables/build-db-create-table-string“. Ona se primenjuje za svaki entitet modela i prima sledeće parametre: potpis entiteta, njegove osobine,

tip baze podataka, indikator da li da napravi nov konfiguracioni fajl tipova („:force“) i ostale argumente.

Pre nego što počne da pravi string koji predstavlja odgovarajuću „CREATE TABLE ...“ *SQL* naredbu, potrebno je da pročita mapiranja Larva tipova na konkretne tipove podataka neke baze podataka. Za ovo je zadužena funkcija „larva.db.utils/make-db-data-types-config“ koja će, ako je potrebno, napraviti poseban fajl u kome će biti data mapiranja tipova za onu bazu podataka koja je prethodno zaključena. Fajl će biti generisan na standardnoj lokaciji „larva_src/db_types_config.clj“. Taj fajl npr. za *Postgres* izgleda ovako:

```
{:postgres
 {:binary "BYTEA",
  :geo "POINT",
  :bignum "BIGINT",
  :bigfloat "FLOAT8",
  :str "VARCHAR(30)",
  :id "SERIAL",
  :datetime "TIMESTAMPTZ",
  ...
  :json "JSON",
  :text "TEXT"}}
```

Ako korisnik sada želi promeniti tip podataka koji će biti korišćen za čuvanje *JSON-a* on u ovom fajlu može izmeniti vrednost ključa „:json“ u npr. „JSONB“ *Postgres* tip. Kadagod želi da ponovo koristi podrazumevanu konfiguraciju tipova za neki tip baze može za „:force“ parametar proslediti neku istinitosno tačnu vrednost i ovaj fakl će biti ponovo izgenerisan.

Moguće je i za konkretnu osobinu nekog entiteta proslediti tip podataka koji nije obuhvaćen definicijom Larva DSL. To se postiže na jednostavan način, tako što se na mestu vrednosti za ključ „:type“ umesto predefinisane ključne reči napiše string. Sadržaj tog stringa će doslovce biti korišćen kao tip podataka u bazi — biće korišćen u „CREATE TABLE ...“ generisanoj *SQL* naredbi.

Funkcija „larva.db.tables/build-db-create-table-string“ vraća string za kreiranje tabele, strukturu koja sadrži osobine koje predstavljaju reference i strukturu koja sadrži osobine koje treba da se predstave kolonom u tabeli. Ovako izgleda primena ove funkcije na entitet „Bend“ u modelu iz listinga 2:

```
(let [entitet (nth (api/all-entities) 1)
      osobine (api/entity-properties entitet)]
      (tbl/build-db-create-table-string entitet osobine :postgres
                                        true nil))
```

Vraćena struktura je:

```
["(id SERIAL PRIMARY KEY,\n ime VARCHAR(30),\n zanr VARCHAR(30),
  \n velicina VARCHAR(30))"
  {"Bend"
   [{:name "clanovi",
     :type {:coll :reference, :to ["Muzicar" "bend"]}}]}
  {:needed-columns
   [{:name "ime", :type :str}
    {:name "zanr", :type :str}
    {:name "velicina", :type :str}],
   :entity "Bend"}]
```

Struktura daje string koji će biti upotrebljen u početnoj migraciji, govori da bend ima referencu na kolekciju muzičara preko osobine „bend“ i da su kolone koje će imati rezultujuća tabela „ime“, „zanr“ i „velicina“.

Na ovaj način se dobijaju *SQL* naredbe za kreiranje samo osnovnih tabela dok se one koje potiču od relacija „više na više“ i rekurzivnih relacija (i druge) kreiraju posebnom funkcijom „larva.db.tables/build-additional-templates-keys“. Pomenuta funkcija koristi reference dobijene iz „larva.db.tables/build-db-create-table-string“.

U *Luminus* okruženju za izgradnju *Web* aplikacija za migriranje baza podataka koristi se biblioteka *Migratus*[18]. Ona obezbeđuje da se migracije odvijaju po hronološkom rasporedu — za „up“ migracije prvo one sa manjim brojem, za „down“ migracije obrnuto. Larva zbog toga mora da vodi računa o redosledu kreiranja fajlova koji predstavljaju migracije kako bi sve tabele koje učestvuju u naknadnim ograničenjima bile dostupne u vreme postavljanja ograničenja.

Svi migracioni fajlovi generisani za model iz listinga 2 su:

```
20160831145758401-add-Musicians-table.down.sql
20160831145758401-add-Musicians-table.up.sql
20160831145758437-add-Bands-table.down.sql
20160831145758437-add-Bands-table.up.sql
20160831145758463-add-Festivals-table.down.sql
20160831145758463-add-Festivals-table.up.sql
20160831145758502-add-Musicians__honors__smpl_coll-table.down.sql
20160831145758502-add-Musicians__honors__smpl_coll-table.up.sql
20160831145758505-alter-tables.down.sql
20160831145758505-alter-tables.up.sql
```

Listing 3: Izgenerisani fajlovi migracija.

Bitno je da migracija „20160831145758437-add-Bands-table.up.sql“ koja sadrži:

```
CREATE TABLE Bands
(id SERIAL PRIMARY KEY,
 ime VARCHAR(30),
 zanr VARCHAR(30),
 velicina VARCHAR(30));
```

počinje manjim brojem nego „20160831145758505-alter-tables.up.sql“ koja sadrži:

```
ALTER TABLE Musicians
ADD CONSTRAINT FK__Musicians__Bands__bend FOREIGN KEY (bend)
REFERENCES Bands(id);
```

Kako bi referencirana tabela bendova bila dostupna u momentu referenciranja u „ALTER TABLE ...“ naredbi.

Korisnici *Luminus* okruženja mogu da pokrenu migracije kroz komandnu liniju na standardan način:

```
lein run migrate
```

Dakle, sve što je potrebno da korisnik uradi kako bi dobio spremnu šemu baze podataka jeste da napiše Larva JSD model, pokrene generator i potom komandu navedenu iznad. Za razliku od *Korma* sistema gde bi se od njega zahtevalo da piše sve fajlove navedene u listingu 3.

3.1.4 Generisanje funkcija

Pored toga što daje migracije kojima se uspostavlja baza podataka *Larva* pruža i niz funkcija koje omogućavaju korisniku da na idiomatski način pravi i koristi podatke (poštujući šemu ako ona postoji). Ovo se postiže tako što se korisniku pruža niz funkcija za rukovanje tim podacima koje su generisane tako da oslikavaju oblik (ili šemu) podataka opisanu pomoću *Larva DSL*-a.

Funkcije sa kojima korisnik radi su dobijene generisanjem *HugSQL* koda. Generisani kod se nalazi na standardnoj lokaciji za *framework* koji se koristi. Za *Luminus framework* standardna lokacija je:

`koren_projekta/resources/sql/`

U ovom direktorijumu će se nalaziti nekoliko *SQL* fajlova koje je potrebno da korisnik učita na način koji to zahteva korišćeni *framework*. U *Luminus framework*-u se to radi tako što se proširi standardni fajl:

`koren_projekta/src/clj/naziv_projekta/db/core.clj`

U ovaj fajl je potrebno dodati konstrukciju koja učitava sve izgenerisane *HugSQL* fajlove i od njih pravi funkcije koje odmah postaju dostupne u okviru standardnog *REPL*-a. To se radi na sledeći način:

```
(ns larva-postgres.db.core
  (:require
    [conman.core :as conman]
    ...)
    ...)

(conman/bind-connection *db*
                        "sql/fajl_1.sql"
                        "sql/fajl_2.sql"
                        ...)
```

3.1.5 Imenovanje izgenerisanih funkcija

Lisp kao u osnovi dinamički programski jezik u mnogome insistira na pravilnom imenovanju elemenata programa. Kako bi se funkcije koje *Larva* generiše uklopile u celokupno *Clojure* okruženje potrebno je da prate način imenovanja.

Svi entiteti opisani u *Larva DSL-u* imaće sledeće četiri osnovne funkcije:

- ***create-ime_ entiteta!***
  Kao parametar prima mapu sa ključevima i vrednostima koji odgovaraju osobinama entiteta.
- ***update-ime_ entiteta!***
  Prima istu mapu parametara kao prethodna funkcija uz dodatni „:id“ ključ.
- ***get-ime_ entiteta***
  Kao parametar prima mapu koja sadrži samo „:id“ ključ sa odgovarajućom vrednošću.
- ***delete-ime_ entiteta!***
  Prima istu mapu parametara kao prethodna funkcija.

Primetimo da se imana nekih funkcija završavaju znakom uzvika. *Clojure* kao funkcionalan programski jezik teži da izbegne korišćenje funkcija koje imaju sporedne efekte. Sa druge strane, *Clojure* ističe svoj pragmaticizam i neće „otežavati“ programeru da napiše funkcije koje imaju sporedne efekte. Zato postoji pravilo (u okviru zajednice *Clojure* programera) da se funkcije koje imaju sporedne efekte nazivaju imenima koja se završavaju znakom uzvika. Funkcije koje prave, menjaju ili brišu podatke iz baze podataka su upravo takve funkcije.

Pored četiri standardne funkcije koje imaju svi entiteti biće izgenerisano još funkcija koje potiču od međusobnih relacija između njih. Ako posmatramo primer iz listinga 2 vidimo da je muzičar u vezi sa bendom, a bend u vezi sa više muzičara. Takođe vidimo i da su priznanja predstavljena kako kolekcija stringova. Ovo će *Larva* u kombinaciji sa relacionom bazom podataka interpretirati tako što će napraviti novu tabelu koja će imati kolonu stringova i koja će biti u „jedan prema više“ relaciji sa tabelom muzičara. Tako ćemo za ovaj jednostavan primer dobiti još nekolicinu funkcija:

- ***get-muzicar-priznanja***
  Koja prima mapu sa „:muzicar“ ključem koji predstavlja njegov identifikator i vraća listu svih njegovih priznanja.

- ***assoc-muzicar-priznanja!***
  Prima mapu sa ključem „:muzičar“ i ključem „:priznanja“ koja prima listu identifikatora njegovih priznanja.

- ***dissoc-muzicar-priznanja!***
  Prima identičnu mapu parametara i služi da izbaci određena priznanja iz korelacije sa zadatim muzičarem.

- ***dissoc-all-muzicar-priznanja!***
  Prima mapu sa ključem „:muzičar“ i izbacuje iz korelacije sva priznanja.

- ***get-muzicar-bend***
  Prima mapu sa ključem „:muzičar“ i vraća njegov bend.

- ***get-bend-clanovi***
  Prima mapu sa ključem „:bend“ i vraća sve članove tog benda.

- ***assoc-muzicar-bend!***
  Prima mapu sa ključevima „:bend“ i „:muzičar“ i zadaje muzičaru određeni bend.

- ***assoc-bend-clanovi!***
  Prima mapu sa ključevima „:bend“ i „:clanovi“ koji predstavlja listu identifikatora muzičara i dodeljuje bendu te članove.

- ***dissoc-muzicar-bend!***
  Prima mapu sa ključevima „:bend“ i „:muzičar“ i izbacuje bend iz korelacije sa zadatim muzičarem.

- ***dissoc-bend-clanovi!***
  Prima mapu sa ključem „:clanovi“ i izbacuje ih iz korelacije sa njihovim bendom.

- ***dissoc-all-bend-clanovi!***
  Prima mapu sa ključem „:bend“ i izbacuje iz asocijacije sve muzičare koji su bili u bendu.

Motiv za korišćenje termina assoc dissoc potiče iz samog programskog jezika *Clojure* koji za rukovanje svojim asocijativnim strukturama podataka (perzistentnim mapama) koristi funkcije *assoc* i *dissoc*. Na primer, ako imamo sledeću mapu:

```
(def jedna-mapa {:kolicina 12 :boja 'crvena :broj 89 :ime "Neko"})
```

I želimo da promenimo njen ključ „:boja“ u simbol „'zelena“ to ćemo idiomatski uraditi na sledeći način:

```
(assoc jedna-mapa :boja 'zelena)
;; => {:kolicina 12, :boja zelena, :broj 89, :ime "Neko"}
```

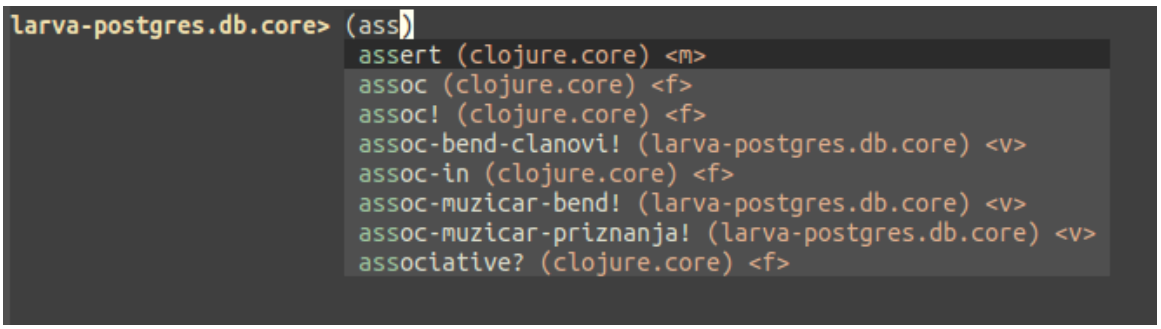
Ako želimo da izbacimo ključ „:ime“ to možemo uraditi na sledeći način:

```
(dissoc jedna-mapa :ime)
;; => {:kolicina 12, :boja crvena, :broj 89}
```

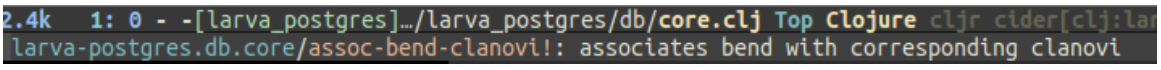
Otuda i imena funkcija koje generiše *Larva* imaju imena koja počinju sa *assoc* i *dissoc* a u nastavku imaju imena entiteta na koje se odnose.

3.2 Larva i *REPL* kao razvojno okruženje

Osnovna ideja prethodno opisanih funkcija jeste da se na prirodan način uklope u proces razmišljanja *Clojure* programera. *Clojure* programsko okruženje, kao *Lisp*, naglašava razvoj uz korišćenje *REPL-a*. To u svom nastojanju podrazumeva da programer u svakom trenutku može da isproba svoje ideje i tako inkrementalno dolazi do najboljeg mogućeg rešenja. Svi alati programskog jezika su apsolutno dostupni u *REPL-u*, kao i kontekst izvršavanja. *Larva* teži da se u potpunosti uklopi u ovakav koncept razvoja. Funkcije koje larva pruža su u potpunosti dostupne u *REPL-u* pa su posledično podržane i od strane razvojnih okruženja (*IDE*[14]). Larva pored logike funkcije teži i da im dodeli što smisleniju dokumentaciju. Na kraju se dobija da korisnik u *REPL-u* ima automatsko dovršavanje imena funkcija kao i njihovu dokumentaciju kao što se vidi na slikama 3.2 i 3.3.



Slika 3.2: Dovršavanje imena funkcije u *CIDER* razvojnom okruženju.



Slika 3.3: Dokumentacija funkcije u *CIDER* razvojnom okruženju.

Glava 4

Zaključak

Predmet ovog rada je bila implementacija sistema za perzistenciju podataka u *Clojure* okruženju. Ovaj rad je motivisan idiomatskim i praktičnim nedostatcima uočenim pregledom i analizom stanja u oblasti. *Clojure* ekosistem se vremenom sve više razvija podržavajući mnoge nove koncepte koji proizilaze kako iz industrijske prakse tako i iz radova u oblasti programskih jezika i razvoja softvera. Uočene praznine u ovom ekosistemu se uglavnom odnose na neke manje-više klasične koncepte kao što je relacioni model podataka. S obzirom da su klasični koncepti i dalje u aktivnoj upotrebi i razvoju *Clojure* ekosistem, zarad većeg stepena prihvatanja u široj programerskoj zajednici, ima potrebu da ih podrži. Duh inovativnosti koji *Clojure* nosi sa sobom (a delom crpi i iz svojih korena u *Lisp-u*) donosi mnoga tehnički i konceptualno dominantna rešenja. Njihova vidljivost u širokoj programerskoj zajednici je smanjena zbog nešto gore podrške klasičnim konceptima na koje su programeri u glavnom navikli. *Larva* predstavlja težnju da se u *Clojure* okruženju idiomatski podrži izražavanje i perzistencija podataka koncipiranih po ideji relacionog modela podataka.

Prednosti *Larva* sistema pri rukovanju i perzistenciji podataka u odnosu na ostala rešenja u *Clojure* ekosistemu su:

- Izražavanje modela podataka u obliku perzistentne mape — Ostala rešenja kao što je *Korma* koriste makroe. Makro sistem jeste velika moć *Lisp-a* ali predstavu oblika podataka bez gubitka semantike možemo svesti na mapu i time dobiti na jednostavnosti. Takođe je time postignuta i integracija sa različitim alatima za validaciju i zadavanje šeme strukturama podataka, što predstavlja trenutnu tendenciju u okviru *Clojure* ekosistema.

- Podrška za migracije — *Larva* na osnovu modela podataka generiše migracije za uspostavljanje šeme baze podataka (kod relacionih baza podataka). Ostala trenutno dostupna rešenja nemaju tu mogućnost.
- Pruža mogućnost integracije sa okvirima (*frameworks*) — Trenutna implementacija generiše kod koji poštuje strukturu projekta koju nalažu okviri (trenutno samo *Luminus web framewrok*).
- *Larva* funkcije za rukovanje podacima su napravljene sa osnovnom idejom da podrže idiomatske konstrukcije *Clojure* programskog jezika a ne relacionih baza podataka. Već postojeća rešenja daju korisnicima mogućnost da koriste koncepte samih relacionih baza podataka (*HugSQL*, *Yesql*).
- *Larva* ima svoju internu reprezentaciju oblika podataka (graf) i iz njega može generisati podršku i za druge sisteme za perzistenciju podataka.

Osnovni doprinos ove teze je konstrukcija osnovne verzije sistema za perzistenciju podataka koji poštuje principe *Clojure* programskog jezika kao i njegov pogleda na funkcionalno programiranje.

4.1 Dalji pravci razvoja Larve

U budućim verzijama razvoj *Larve* bi mogao biti koncentrisan na sledeće aspekte:

- Raščlanjivanje trenutne monolitne implementacije na osnovnu logiku *Larva* generatora i adaptere za različite okvire.
- Podrška za dinamičko generisanje funkcija bez *HugSQL-a*. Treba zadržati i trenutnu mogućnost ali je potrebna dopuna za one kojima je osnovna funkcionalnost *Larve* dovoljna i ne žele da njihov projekat sadrži sav generisan kod.
- Potrebna je optimizacija generisanog *SQL-a* sa stanovišta performansi. Ovo je moguće uraditi jer interna graf reprezentacija oblika podata sadrži sve potrebne informacije za interpretaciju šeme baze podataka kod relacionih sistema baza podataka.
- Implementacija podrške za različite *NoSQL* baze podataka.
- Zamena *plumatic/schema* biblioteke sa *clojure.spec* standardnom bibliotekom koja je najavljena za verziju *Clojure 1.9*. Ovo može pružiti osnovu za generativno testiranje funkcija koje koriste podatke opisane *Larva JSD-om*.

- Generisanje *Larva* modela podataka iz već postojećih šema relacionih baza podataka.

Bibliografija

[1] Algol programski jezik. http://groups.engin.umd.umich.edu/CIS/course.des/cis400/algol/algol.html. [Online; accessed 13-Avgust-2016].

[2] Apache hadoop. http://hadoop.apache.org/. [Online; accessed 13-Avgust-2016].

[3] Cider razvojno okruženje. https://cider.readthedocs.io/en/latest/. [Online; accessed 13-Avgust-2016].

[4] Clojure programski jezik. https://clojure.org/. [Online; accessed 13-Avgust-2016].

[5] Clojurescript. https://github.com/clojure/clojurescript. [Online; accessed 13-Avgust-2016].

[6] Common language runtime. https://en.wikipedia.org/wiki/Common_Language_Runtime. [Online; accessed 13-Avgust-2016].

[7] Common lisp. https://common-lisp.net/. [Online; accessed 13-Avgust-2016].

[8] Conman. https://github.com/luminus-framework/conman. [Online; accessed 13-Avgust-2016].

[9] Elixir programski jezik. http://elixir-lang.org/. [Online; accessed 13-Avgust-2016].

[10] Elm programski jezik. http://elm-lang.org/. [Online; accessed 13-Avgust-2016].

[11] Emacs lisp. https://www.gnu.org/software/emacs/. [Online; accessed 13-Avgust-2016].

[12] Fortran programski jezik. http://groups.engin.umd.umich.edu/CIS/course.des/cis400/fortran/fortran.html. [Online; accessed 13-Avgust-2016].

[13] Graphwiz. http://www.graphviz.org/. [Online; accessed 13-Avgust-2016].

[14] Integrated development environment. https://en.wikipedia.org/wiki/Integrated_development_environment. [Online; accessed 13-Avgust-2016].

[15] Java virtual machine. https://en.wikipedia.org/wiki/Java_virtual_machine. [Online; accessed 13-Avgust-2016].

[16] Loom. https://github.com/aysylu/loom. [Online; accessed 13-Avgust-2016].

[17] Luminus web. http://www.luminusweb.net/. [Online; accessed 13-Avgust-2016].

[18] Migratus. https://github.com/yogthos/migratus. [Online; accessed 13-Avgust-2016].

[19] Pedestal. https://github.com/pedestal/pedestal. [Online; accessed 13-Avgust-2016].

[20] Postgresql. https://www.postgresql.org/. [Online; accessed 13-Avgust-2016].

[21] Read–eval–print loop. https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop. [Online; accessed 13-Avgust-2016].

[22] Standard clojure jdbc implementation. https://github.com/clojure/java.jdbc. [Online; accessed 13-Avgust-2016].

[23] Ubergraph. https://github.com/Engelberg/ubergraph. [Online; accessed 13-Avgust-2016].

[24] Xml. https://www.w3.org/XML/. [Online; accessed 13-Avgust-2016].

[25] Phil Bagwell. Ideal hash trees. *Es Grands Champs*, 1195, 2001.

[26] Alonzo Church. The calculi of lambda-conversion. *Princeton University Press*, 1941.

[27] R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, 4th edition, Jul 2009.

[28] Brian Goetz. Stewardship: the sobering parts. https://www.youtube.com/watch?v=2y5Pv4yN0b0&t=1h02m18s, 2014. [Online; accessed 13-Avgust-2016].

[29] Paul Graham. What made lisp different. http://www.paulgraham.com/diff.html, Decembar 2001. [Online; accessed 13-Avgust-2016].

[30] Paul Graham. The roots of lisp. http://lib.store.yahoo.net/lib/paulgraham/jmc.ps, Januar 2002. [Online; accessed 13-Avgust-2016].

[31] Rich Hickey. Edn - extensible data notation. https://github.com/edn-format/edn. [Online; accessed 13-Avgust-2016].

[32] Rich Hickey. Multimethods and hierarchies. http://clojure.org/reference/multimethods. [Online; accessed 13-Avgust-2016].

[33] Rich Hickey. Persistent data structures and managed references. https://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey. [Online; accessed 13-Avgust-2016].

[34] Rich Hickey. Reader. http://clojure.org/reference/reader. [Online; accessed 13-Avgust-2016].

[35] Sonya E. Keene. *Object-Oriented Programming in COMMON LISP: A Programmer's Guide to CLOS*. Addison-Wesley Professional, 1st edition, Januar 1989.

[36] Jean Niklas L'orange. Understanding clojure's persistent vectors. http://hypirion.com/musings/understanding-persistent-vector-pt-1. [Online; accessed 13-Avgust-2016].

[37] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 1960.

[38] Kent Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1), Jun 1988.

УНИВЕРЗИТЕТ У НОВОМ САДУ ● **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**
21000 НОВИ САД, Трг Доситеја Обрадовића 6

КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

Редни број, **РБР**:	
Идентификациони број, **ИБР**:	
Тип документације, **ТД**:	Монографска документација
Тип записа, **ТЗ**:	Текстуални штампани материјал
Врста рада, **ВР**:	Завршни (Мастер) рад
Аутор, **АУ**:	Новак Бошков
Ментор, **МН**:	Доц. др Игор Дејановић
Наслов рада, **НР**:	Систем за перзистенцију података у програмском језику *Clojure*
Језик публикације, **ЈП**:	српски / латиница
Језик извода, **ЈИ**:	српски / енглески
Земља публиковања, **ЗП**:	Република Србија
Уже географско подручје, **УГП**:	Војводина
Година, **ГО**:	2016.
Издавач, **ИЗ**:	Ауторски репринт
Место и адреса, **МА**:	Нови Сад; трг Доситеја Обрадовића 6
Физички опис рада, **ФО**: (поглавља/страна/ цитата/табела/слика/графика/прилога)	4 / 92 / 3 / 0 / 16 / 0 / 0
Научна област, **НО**:	Рачунарске науке
Научна дисциплина, **НД**:	Језици специфични за домен
Предметна одредница/Кqучне речи, **ПО**:	
УДК	
Чува се, **ЧУ**:	У библиотеци Факултета Техничких наука, Нови Сад
Важна напомена, **ВН**:	
Извод, **ИЗ**:	У овом раду је имплементиран систем за перзистенцију података у програмском језику *Clojure.* Систем је имплементиран као интерни (енг. Domain Specific Language) користећи начела и праксу самог програмског језика и функционалне парадигме програмирања.
Датум прихватања теме, **ДП**:	
Датум одбране, **ДО**:	

Чланови комисије, **КО**:	Председник:	др Гордана Милосављевић, ванредни професор, ФТН Нови Сад
	Члан:	др Ђорђе Пржуљ, доцент, ФТН Нови Сад
	Члан, ментор:	др Игор Дејановић, доцент, ФТН Нови Сад

Потпис ментора

UNIVERSITY OF NOVI SAD ● **FACULTY OF TECHNICAL SCIENCES**
21000 NOVI SAD, Trg Dositeja Obradovića 6

KEY WORDS DOCUMENTATION

Accession number, **ANO**:	
Identification number, **INO**:	
Document type, **DT**:	Monographic publication
Type of record, **TR**:	Textual printed material
Contents code, **CC**:	Master Thesis
Author, **AU**:	Novak Boškov
Mentor, **MN**:	Igor Dejanović, PhD, assist. prof.
Title, **TI**:	Data persistence system in Clojure programming language
Language of text, **LT**:	Serbian
Language of abstract, **LA**:	Serbian / English
Country of publication, **CP**:	Republic of Serbia
Locality of publication, **LP**:	Vojvodina
Publication year, **PY**:	2016.
Publisher, **PB**:	Author's reprint
Publication place, **PP**:	Novi Sad, Dositeja Obradovića sq. 6
Physical description, **PD**: (chapters/pages/ref./tables/pictures/graphs/appendixes)	4 / 92 / 3 / 0 / 16 / 0 / 0
Scientific field, **SF**:	Computer science
Scientific discipline, **SD**:	Domain Specific Language
Subject/Key words, **S/KW**:	
UC	
Holding data, **HD**:	The library of Faculty of Technical Sciences, Novi Sad, Serbia
Note, **N**:	
Abstract, **AB**:	This work presents implementation of data persistence system in programming language Clojure. System is implemented as a internal Domain Specific Language using principles and practice of programming language itself and functional programming paradigm.

Accepted by the Scientific Board on, **ASB**:

Defended on, **DE**:

Defended Board, **DB**:	President:	Gordana Milosavljević, PhD, assoc. prof.
	Member:	Đorđe Pržulj, PhD, assist. prof.
	Member, Mentor:	Igor Dejanović, PhD, assist. prof.

Menthor's sign