UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
INSTITUT ZA MATEMATIKU

Miloš Radovanović

Implementacija programskog jezika LispKit LISP u Javi

Diplomski rad

mentor: dr Mirjana Ivanović

Novi Sad
septembar 2001.

Sadržaj

Predgovor

Programski jezik LispKit LISP nastao je 1980. godine, kao pokušaj da se stvori funkcionalni programski jezik koji je, sa jedne strane, dovoljno izražajan da se u njemu mogu pisati upotrebljivi programi, a sa druge, dovoljno jednostavan da se može implementirati na elegantan i ilustrativan način. Njegov tvorac, Piter Henderson, u svojoj knjizi [7] dao je opis njegovog dizajna i mogućeg načina implementiranja, koristeći pri tom jedan hipotetički programski jezik.

S obzirom na neverovatnu brzinu s kojom se stvari menjaju u svetu računara, od tada je, moglo bi se reći, protekla čitava jedna era. Nastajali su novi programski jezici, različitih stilova i mogućnosti. Međutim, LispKit LISP, zbog svoje jasnoće i ilustrativnosti, nije izgubio na značaju. Ali, pojava programskog jezika Java otvorila je mogućnost još većeg pojednostavljenja implementacije LispKit LISP-a, zahvaljujući brojnim lepim osobinama ovog, relativno mladog, jezika.

Prva glava ovog rada predstavlja uvod u računare, programiranje i programske jezike uopšte, sa posebnim osvrtom na funkcionalni stil programiranja. U drugoj glavi opisan je programski jezik LispKit LISP, počev od njegove osnovne strukture podataka – s-izraza, do sintakse i semantike samog jezika. Treća glava posvećena je opisu virtuelne SECD mašine na kojoj će se programi izvršavati, a četvrta prelazi na konkretnu implementaciju svega što je opisano u prethodne dve glave: s-izraza, provere sintaksne ispravnosti LispKit LISP programa, prevodioca iz LispKit LISP-a u kod SECD mašine, i interpretera tog koda. Provera sintaksne ispravnosti suštinska je novina u odnosu na originalni opis implementacije. Peta glava sadrži zaključak koji je posledica iskustva na izradi ovog rada, kao i neke od mogućnosti za njegov nastavak i unapređenje.

Implementacija je rađena sa verzijom 1.1 Java jezika na umu, a zadovoljava i kriterijume verzije 2. Od konkretnih alata korišćen je Java Development Kit, verzija 1.1.8, a program provereno radi i sa verzijom 1.2.2. Za implementaciju sintaksne analize s-izraza korišćen je kompajler generator Coco/R for Java, verzija 1.11.

Ovom prilikom zahvaljujem mentoru dr Mirjani Ivanović na iskazanoj predusretljivosti u otežanim uslovima u kojima je nastajao ovaj rad, kao

i članovima komisije na uloženom vremenu oko odbrane diplomskog rada. Zahvalio bih i svim kolegama i profesorima Prirodno-matematičkog fakulteta u Novom Sadu na konstruktivnoj i inspirativnoj atmosferi koja me je pratila u toku studiranja. Naposletku, zahvalio bih svojim roditeljima, koji su mi podarili uslove za lep život i neometano studiranje.

U Novom Sadu,
septembra 2001.

Autor

Glava 1

Uvod

1.1 Računari i programski jezici

U vreme pisanja ovog teksta, računare možemo posmatrati kao mašine koje u svakom trenutku rade na rešavanju jednog ili više zadataka. Ma koliko ono što neki računar radi izgledalo složeno, inteligentno ili, čak, nepredvidivo, to nije ništa više od zadatka, čiji je proces rešavanja računaru saopšten na određen način. Još od vremena nastanka računara, problemu saopštavanja računaru šta i kako da uradi pristupalo se sa različitih strana. Jedan od glavnih pristupa je korišćenjem *programskih jezika*, pomoću kojih se na sistematičan, jasan i precizan način opisuje niz radnji koje računar treba da izvrši rešavajući određeni zadatak.

Rešavanju zadatka pomoću računara pristupa se, uglavnom, raščlanjivanjem zadatka na jednostavnije zadatke, ovih na još jednostavnije, sve dok se ne stigne do niza pojedinačnih radnji koje računar može da izvrši. Zatim se taj niz radnji saopštava računaru pomoću nekog od programskih jezika, i time se dobija *program*. Opisani proces obično se naziva *programiranje*, a njime se bavi čovek – *programer*. Programski jezici su, u stvari, veštački jezici koji bi trebalo da su, s jedne strane, razumljivi za čoveka, a sa druge, dovoljno jednostavni i lišeni mogućnosti nejasnoća u izražavanju, da bi bili „razumljivi“ za računar. Usvojićemo dogovor da se kaže da je program *napisan u*, kao i da je zadatak *implementiran u* nekom programskom jeziku.

Iako je računarstvo relativno mlada oblast ljudskog delovanja, već je stvoreno na hiljade programskih jezika, različitih odlika, namena i načina korišćenja. Sastavljene su i razne njihove klasifikacije, a mi ćemo pomenuti neke koje su značajne radi dopune i pojašnjenja ranije izloženog, kao i radi razumevanja onoga što sledi.

Sadržaj ove glave u velikoj meri je zasnovan na materiji izloženoj u [3].

Klasifikacija 1 Ako se kao kriterijum za klasifikaciju programskih jezika uzme stepen njihove zavisnosti od računara, razlikujemo dve klase programskih jezika:

- Mašinski zavisni
- Mašinski nezavisni.

Kod **mašinski zavisnih** jezika je karakteristično da su vezani za konkretan tip računara – programi pisani za jedan tip računara teško da će moći da se izvršavaju na računaru drugog tipa. U ovu klasu programskih jezika spadaju: mašinski, asemblerski i makro-jezici. U **mašinski nezavisne** programske jezike spadaju jezici koji se mogu koristiti na različitim tipovima računara. Nazivaju se i *viši programski jezici*, zbog toga što se udaljavaju, odnosno idu „iznad“ elementarnih radnji koje konkretni računari izvršavaju. Programer koji piše programe u nekom od jezika ove klase ne mora da poznaje ni jednu od elementarnih radnji koje može da izvrši računar za kojim radi. Neki od predstavnika ove klase programskih jezika jesu: FORTRAN, Pascal, C, Java, Common LISP, PROLOG itd.

Sada sledi ispravka, odnosno dopuna ranije rečenog: *mašinski jezik* je *jedini* jezik čiji se programi mogu direktno izvršavati na računaru. Preciznije, niz radnji saopšten računaru pomoću mašinskog jezika računar stvarno može da izvrši. Kod svih ostalih programskih jezika potrebna je dodatna obrada (od strane računara) da bi se programi pisani u njima izvršili, što je posledica činjenice da su ti programi pisani pomoću simbola, odnosno predstavljaju *tekst* koji je čitljiv za čoveka. Razlog za to je velika razlika između načina razmišljanja čoveka i konkretne realizacije današnjih računara, odnosno elementarnih radnji koje oni mogu da izvrše. Uz to, elementarne radnje se na današnjim računarima označavaju brojkama, što sigurno ne doprinosi razumljivosti mašinskih programa za čoveka.

Osvrnimo se na pomenutu *dodatnu obradu*, a koja je neophodna da bi se na računaru izvršio program koji nije pisan u mašinskom jeziku. U tu svrhu koriste se specijalni programi, i to od dve vrste:

- Kompajleri
- Interpreteri.

Kompajleri, poznati i kao *prevodioci*, uzimaju ulazni podatak – program pisan u nekom ne-mašinskom programskom jeziku, a kao izlaz daju taj isti program preveden u neki programski jezik nižeg nivoa, često baš mašinski. **Interpreteri**, odnosno *interpretatori*, takođe uzimaju program pisan u nemašinskom programskom jeziku, s tim što potom *izvršavaju* radnje saopštene u programu.

Kompajleri i interpreteri, budući da su programi, moraju biti napisani u nekom programskom jeziku. Ovaj rad će ilustrovati jedan konkretan kompajler, i jedan konkretan interpreter, koji će biti napisani u programskom jeziku *Java*. Kompajler će prevoditi programe pisane na programskom jeziku *LispKit LISP* u programski jezik koji nazivamo *kôd SECD mašine*, a interpreter će taj kod izvršavati.

Pređimo sada na još jednu ilustrativnu klasifikaciju programskih jezika.

Klasifikacija 2 Ako se kao kriterijum za klasifikaciju programskih jezika uzme *način* rešavanja problema, možemo razlikovati sledeće klase programskih jezika:

- Proceduralni (imperativni)
- Deklarativni (deskriptivni).

Ovde je možda bolje govoriti o proceduralnom i deklarativnom **stilu** programiranja, odnosno o *načinu razmišljanja* programera prilikom rešavanja problema. Za neki programski jezik se tada kaže da, manje ili više, *naginje* jednom ili drugom stilu, odnosno da pruža veće mogućnosti da se zadatak reši u jednom od dva stila.

Proceduralni, odnosno *imperativni* stil programiranja karakteriše način razmišljanja koji bi se približno mogao opisati kao iznalaženje odgovora na pitanje KAKO – kojim postupkom rešiti problem, dok je kod **deklarativnog** (*deskriptivnog*) stila dominantno pitanje ŠTA – iz čega se problem sastoji. Ilustrujmo ovo na primeru.

Primer 1.1 *Program za gradnju kolibe napisan u proceduralnom stilu mogao bi raščlaniti taj zadatak na sledeće podzadatke:*

1. *Postaviti temelje*
2. *Sazidati zidove*
3. *Postaviti pod*
4. *Postaviti krov,*

gde je navedeni redosled od ključne važnosti za konačni uspeh gradnje. S druge strane, program u deklarativnom stilu sadržavao bi opis sastavnih delova kolibe:

- *Zidovi se oslanjaju na temelj*
- *Pod se oslanja na temelj*

- *Krov se oslanja na zidove,*

pri čemu redosled deklaracija nije od važnosti, a redosled gradnje kolibe je implicitno sadržan u njenom opisu.

U klasu programskih jezika koji podržavaju proceduralni stil ubrajamo Pascal, C, FORTRAN, COBOL i druge. Deklarativni stil se može podeliti na dve važne podvrste – na *logički* i *funkcionalni* stil. Podklasu jezika koji podržavaju funkcionalni stil čine: (porodica jezika) LISP, PL, Haskell i drugi. Poznatiji predstavnici programskih jezika pretežno logičkog stila jesu: PROLOG, KLO, Mandala itd. Postoje i mnogi drugi stilovi programiranja, na primer *objektno-orijentisani*, koji ima osobine i proceduralnog i deklarativnog stila, a predstavljaju ga programski jezici Java, C++, SmallTalk, Delphi itd., zatim *distribuirani*, *relacioni* ...

Retki su primeri programskih jezika koji podržavaju isključivo jedan stil programiranja. U takve jezike, moglo bi se reći, spada LispKit LISP, koji (praktično) podržava isključivo funkcionalni stil programiranja.

1.2 Funkcionalni stil programiranja

Funkcionalni stil programiranja jeste vrsta deklarativnog stila programiranja kod kojeg je izvršavanje programa zasnovano na pojmu funkcije. (S druge strane, logički stil programiranja zasnovan je na pojmu relacije.) Funkcionalni stil programiranja karakterišu samo tri aktivnosti na kojima se zasniva ceo proces programiranja:

- Definisanje funkcije, tj. pridruživanje imenu funkcije izraza kojim se izračunava vrednost funkcije. U definiciji funkcije mogu se pojavljivati i druge (takođe definisane) funkcije.
- Primena funkcije na argument, tj. poziv funkcije.
- Kompozicija funkcija.

Program funkcionalnog stila je niz definicija i poziva funkcija. Izvršavanje programa je tada, u stvari, proces izračunavanja vrednosti funkcije (ili funkcija), primenjene na određene argumente. Pogledajmo primer.

Primer 1.2 *Posmatrajmo nekoliko definicija matematičkih funkcija:*

$$\begin{aligned} f(x) &= x^2 + 1 \\ g(x, y) &= f(x) + 2y \\ z(x) &= g(x, 2) - 1 \end{aligned}$$

Na ovaj niz definicija funkcija možemo gledati kao na program funkcionalnog stila. Ako želimo da odredimo vrednost funkcije $z(x)$ za $x = 1$, tada proces izračunavanja vrednosti te funkcije predstavlja izvršenje funkcionalnog programa. Tok izvršavanja mogli bismo predstaviti ovako:

$$z(1) = g(1,2) - 1 = (f(1) + 4) - 1 = ((1 + 1) + 4) - 1 = 5$$

sa napomenom da programer, u principu, prilikom pisanja programa funkcionalnog stila o toku njegovog izvršavanja uopšte ne mora da razmišlja.

Na osnovu do sada rečenog moglo bi se zaključiti da su izražajne mogućnosti funkcionalnog stila donekle ograničene. To jeste tačno, ali ako je za rešavanje konkretnog zadatka funkcionalni stil dobar izbor, dobiće se program sa sledećim osobinama:

- jasan je i koncizan,
- po pravilu je kraći od proceduralnog ekvivalenta,
- teže je napraviti grešku, i lakše ju je otkriti ako je napravljena,
- može se lako realizovati na paralelnim računarima,
- ispravnost je moguće formalno dokazati.

U nastavku će biti opisane neke od glavnih osobina funkcionalnog stila programiranja. Objašnjenja će biti potkrepljena poređenjem sa odgovarajućim osobinama proceduralnog stila programiranja.

Nepostojanje eksplicitnog zadavanja redosleda izvršavanja

U funkcionalnom stilu programiranja je nepoznat koncept *eksplicitnog* zadavanja redosleda izračunavanja vrednosti funkcija. Dok program pisan u proceduralnom stilu više liči na *spisak naredbi* koje treba da se izvrše u određenom redosledu, program funkcionalnog stila *definiše izraz* koji predstavlja rešenje zadatka. Primer 1.1 na strani 7 ilustruje ovu razliku.

Pogledajmo još jedan, malo konkretniji, primer.

Primer 1.3 *Faktorijel prirodnog broja n, u oznaci $n!$, definiše se kao proizvod svih prirodnih brojeva manjih ili jednakih sa n, tj. $n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1$. Ekvivalentna definicija bila bi da je $n! = 1$ u slučaju da je $n = 1$, a inače $n! = n \cdot (n-1)!$. Programi za izračunavanje faktorijela proceduralnog i funkcionalnog stila mogli bi se napisati na sledeći način:*

```
rez = 1;                fac(n) =
while (n > 0) {           if n == 0 then 1
  rez = rez * n;          else n*fac(n-1)
  n = n - 1;
}
```

gde se vidi da program proceduralnog stila (levo) implementira prvu datu definiciju faktorijela, i to tako da je bitan redosled u kojem se naredbe izvršavaju, dok program funkcionalnog stila (desno) predstavlja jednostavno preformulaciju druge definicije. Bitno je istaći da `if ... then ... else` *iskaz u funkcionalnom stilu ne predstavlja kontrolu toka izvršavanja programa, kao kod proceduralnog stila, nego predstavlja* uslovni izraz*; što u ovom slučaju znači da taj izraz ima* vrednost `1`*, ako je prosleđen argument* `n` *jednak nuli, dok inače ima vrednost koja se dobija izračunavanjem izraza* `n*fac(n-1)`*.*

Direktna posledica ove osobine funkcionalnog stila programiranja jeste veliki stepen deklarativnosti programa, što za sobom povlači smanjenu mogućnost pravljenja grešaka, naročito onih vezanih za pogrešno zadavanje redosleda izvršavanja delova programa.

Nepostojanje naredbi

Za programe proceduralnog stila tesno je vezan pojam *stanja*, kao nečega što je prisutno i što se može menjati u toku izvršavanja programa. Stanja su uglavnom, manje ili više skriveno od očiju programera, predstavljena vrednostima koje se čuvaju u memoriji računara, i mogu se menjati korišćenjem različitih konstrukcija programskog jezika. Jedna od takvih konstrukcija jeste *naredba dodele*, a pored nje postoje i naredbe za izvršavanje petlji (npr. naredbe `for` i `while` u Javi), i uslovne naredbe (`if`, `switch`), koje se izvršavaju određenim redosledom na koji stanja u velikoj meri utiču. Rezultat rada programa obično se smešta u neku memorijsku lokaciju, i potrebno ga je iz nje preuzeti po završetku obrade. Tako u primeru 1.3 od vrednosti promenljive (memorijske lokacije obeležene simbolom) `n` zavisi tok izvršavanja `while` petlje, a rezultat izračunavanja faktorijela smešta se u promenljivu `rez`.

S druge strane, funkcionalni stil uopšte ne poznaje naredbe, već samo *izraze* – zapise načina izračunavanja vrednosti, slično matematičkom pojmu izraza. Naredbe petlji iz proceduralnog stila zamenjuju se rekurzivnim pozivima funkcija, a uslovne naredbe uslovnim *izrazima*, kao u primeru 1.3. Ceo program funkcionalnog stila jeste, u stvari, jedan izraz, čija vrednost se izračunava za konkretne vrednosti ulaznih argumenata (u primeru imamo jedan argument označen simbolom `n`).

Ako rešenje problema prirodno nalaže korišćenje neke vrste stanja, u funkcionalnom stilu se to postiže eksplicitnim prenošenjem stanja u sve izraze u kojima se to stanje koristi, kao u sledećem primeru.

Primer 1.4 *Računanje faktorijela može se implementirati i prosleđivanjem delimično izračunate vrednosti u naredni poziv funkcije:*

```
fac(n) = f(n,1)
f(n,a) =
  if n > 0 then f(n-1,a*n)
  else a
```

Ova prosleđena vrednost, odnosno parametar `a` *funkcije* `f` *inače se naziva i* akumulirajući parametar*. Primeri korišćenja akumulirajućeg parametra često se mogu sresti u programima funkcionalnog stila.*

Nepostojanje naredbi i eksplicitno prenošenje stanja doprinose deklarativnosti i jasnoći programa funkcionalnog stila, a smanjuju mogućnost pravljenja grešaka pri programiranju, jer direktno utiču na sledeću osobinu funkcionalnog stila programiranja – nepostojanje sporednih efekata.

Nepostojanje sporednih efekata

U funkcionalnom stilu programiranja važi pravilo da isti izraz u istom okruženju uvek ima istu vrednost. Na primer, izraz $x^2 + 1$ uvek ima istu vrednost za istu vrednost x, tj. ako je u nekom okruženju $x = 3$, onda je vrednost izraza uvek 10. Ovo takođe znači i da funkcija pozvana sa istim argumentima uvek daje istu vrednost. Na primer, funkcija $\ln(x)$ za $x = \mathrm{e}$ uvek ima vrednost 1.

Sporedni efekti mogu direktno uticati da ovo pravilo ne važi. Oni su česta pojava u proceduralnom stilu programiranja, dok u funkcionalnom ne bi trebalo ni da postoje. Greške koje mogu nastati kao posledica sporednih efekata po pravilu se teško otkrivaju. Naredba dodele je čest uzrok sporednih efekata, kao u sledećem primeru.

Primer 1.5 *Posmatrajmo fragment programa pisan u programskom jeziku Java:*

```
.....
int a, x;
boolean fun(int i) {
  a++;
  return i == a;
}
.....
```

```
a = 0; x = 1;
if (fun(x) == fun(x))
  System.out.println("Jednako");
else
  System.out.println("Nije jednako");
.....
```

Zbog promene vrednosti globalne promenljive `a`*, program će prikazati poruku* `Nije jednako`*, što znači da je sporedni efekat doveo do toga da ne važi relacija* `fun(x) == fun(x)`*.*

Posledica nepostojanja sporednih efekata je da se izrazi koji imaju istu vrednost mogu slobodno zamenjivati jedni drugima. Npr. ako je $y = x^2 + 1$, tada se izraz $y + y$ može pisati i kao $x^2 + 1 + y$, ili $y + x^2 + 1$, ili $x^2 + 1 + x^2 + 1$. Ova osobina je u matematičkoj logici poznata kao *referentnost pominjanja*[1], i predstavlja moćan alat u matematici.

Funkcije kao ravnopravni objekti

U funkcionalnom stilu programiranja, funkcije su koncept ravnopravan sa svim ostalim konceptima konkretnog programskog jezika. To, recimo, može značiti da su funkcije tip podataka ravopravan sa ostalim tipovima podataka u programskom jeziku, da se mogu smeštati u strukture podataka, da se mogu prosleđivati drugim funkcijama kao parametri, pa čak i da mogu biti vrednost (rezultat izvršavanja) drugih funkcija. Funkcije koje manipulišu drugim funkcijama često se nazivaju *funkcije višeg reda*[2], i možemo ih uporediti sa nekim funkcijama u matematici. Na primer, *izvod* je, u stvari, funkcija višeg reda, čiji su i parametar i rezultat funkcija:

$$\frac{\partial}{\partial x} f(x) = \frac{\partial}{\partial x}(x^3 + x^2 + x) = 3x^2 + 2x + 1 = g(x)$$

gde je $f(x)$ parametar izvoda, a $g(x)$ rezultat.

Nestriktna semantika

Za semantiku nekog jezika, ili stila programiranja, kaže se da je striktna ako je svaki izraz striktan (uključujući tu i pozive funkcija). Izraz je striktan ako i samo ako nema vrednost kada bar jedan od njegovih operanada (odnosno podizraza) nema vrednost. Suprotno, izraz je nestriktan ako ima vrednost čak i kada neki od njegovih operanada nemaju vrednost.

[1] Engl. *referential transparency.*
[2] Engl. *higher order functions.*

Na primer, logički izraz $a \vee b$ ima vrednost *tačno* ako bar jedan od operanada a i b ima vrednost tačno, što znači da ako znamo da a ima vrednost tačno, vrednost b ni ne utiče na konačnu vrednost izraza, i može ostati nedefinisana.

Slično, neka je `(a.b)` oznaka za listu čiji je prvi element `a`, a `b` je ostatak liste. Neka je `car(x)` funkcija koja vraća prvi element liste `x`. Tada, ako je poznata vrednost izraza `a`, biće poznata i vrednost izraza `car((a.b))`, bez obzira da li je vrednost `b` poznata ili ne.

Striktna semantika jezika se realizuje tako što se prvo izračunaju svi operandi izraza (ili svi argumenti funkcije), pa se tek onda izračunava vrednost izraza. Tehnika prosleđivanja parametara kojom se postiže ovo ponašanje naziva se *prenošenje argumenata po vrednosti*[3]. Nestriktna semantika se u praksi realizuje tako što se izračunavanje operanada u izrazu i argumenata u pozivu funkcije odlaže sve dok te vrednosti nisu zaista potrebne. Ta je strategija poznata pod nazivom zadržano (ili lenjo) izračunavanje[4], odnosno princip odlaganja. Odgovarajuća tehnika prenošenja parametara obično se naziva *prenošenje argumenata po potrebi*[5].

Programi se, u principu, brže izvršavaju ako se vrednost izraza može odrediti bez izračunavanja svih operanada. Pored toga, značaj nestriktne semantike ogleda se i u sledeće dve osobine.

1. Programer je oslobođen brige o redosledu izračunavanja, a time i brige o efikasnosti i mogućim greškama u toku izračunavanja vrednosti izraza. Na primer, logički izraz $x = 0 \vee 1/x = 10$ će za $x = 0$ u nestriktnoj semantici imati vrednost *tačno*, dok će se u striktnoj semantici računati vrednosti oba podizraza, pa će izračunavanje vrednosti $1/x = 10$ prouzrokovati grešku u toku izvršavanja programa.

2. Omogućeno je stvaranje (naizgled) beskonačnih struktura podataka. Na primer, neka je izrazom `a = (1.a)` definisana beskonačna struktura – lista jedinica. Izraz `car(a)`, koji bi trebalo da ima vrednost prvog elementa liste `a`, odnosno `1`, u striktnoj semantici ne bi mogao biti izračunat, jer bi došlo do beskonačnog izračunavanja: `car(a)` = `car((1.a))` = `car((1.(1.a)))` = `...` U nestriktnoj semantici argumenti se računaju samo po potrebi, pa je izraz `car(a)` izračunljiv: `car(a)` = `car((1.a))` = `1`.

[3]Engl. *call by value*.

[4]Engl. *lazy evaluation*.

[5]Engl. *call by need*. Pomenimo još jednu tehniku prenošenja parametara – *po imenu* (engl. *call by name*), koja je pretežno odlika proceduralnih jezika i znači da se, manje ili više vidljivo, prenosi informacija o *identitetu* promenljive koja je prosleđena kao parametar pri pozivu funkcije, tako da argument u telu funkcije označava baš tu promenljivu.

Statičko vezivanje

U funkcionalniom stilu programiranja, simbolima koji označavaju izraze, odnosno „promenljivama“, vrednost se dodeljuje u trenutku njihovog definisanja (tj. statički), a ne u toku izvršavanja programa (tj. dinamički), kao što je to slučaj u proceduralnom stilu. Ovde moramo naglasiti razliku – „promenljiva“ u funkcionalnom stilu nije ništa drugo do simbol (ime) kojem se pridružuje neki duži izraz, pa se zahvaljujući spomenutoj ekvivalentnosti pominjanja duži izraz u programu može zamenjvati tim simbolom. U proceduralnom stilu, „promenljiva“ predstavlja *stanje*, odnosno vrednost neke lokacije u memoriji koja se može menjati u toku izvršavanja programa, a obeležava se imenom. Analizirajmo sledeći primer.

Primer 1.6 *Neka je dat program u nekom hipotetičkom programskom jeziku:*

```
neka_je n = 1 i
  neka_je f(z) = z+n i
    neka_je n = 2 u_izrazu f(3)
```

U ovom programu „promenljivoj“ `n` *pridružuje se „duži izraz“* `1`*, i zatim se ime* `n` *koristi umesto tog izraza u ostatku programa. Međutim, kasnije se „promenljivoj“* `n` *pridružuje izraz* `2`*. Kolika je onda vrednost izraza* `f(3)`*?*

Pri statičkom vezivanju, „promenljive“ i vrednosti pridružuju se u trenutku definisanja*, što znači da je funkcija definisana kao* `f(z) = z+1`*, i da naknadne izmene značenja imena* `n` *u drugim izrazima nemaju uticaja. Dakle, vrednost izraza* `f(3)` *je* `4`*.*

Pri dinamičkom vezivanju, „promenljive“ i vrednosti pridružuju se u trenutku izračunavanja*, što znači da je prilikom izračunavanja vrednosti izraza* `f(3)` *„promenljivoj“* `n` *pridružena vrednost* `2` *(takvo je njeno trenutno stanje), te je tada funkcija definisana kao* `f(z) = z+2`*, i vrednost izraza* `f(3)` *je* `5`*.*

O funkcionalnim programskim jezicima

Sve prethodno pomenute osobine određuju funkcionalni stil programiranja. Za programske jezike koji imaju većinu ovih osobina kažemo da su *funkcionalni programski jezici*. Neki od njih imaju karakteristike po kojima bi se mogli svrstati i u *proceduralne programske jezike* (npr. Common LISP), a neki dozvoljavaju i upotrebu sporednih efekata (npr. Common LISP, ML). Naravno, ovakve osobine programskih jezika koje svrstavamo u funkcionalne, ne moraju se koristiti. Ako programski jezik uopšte nema takvih osobina, obično se za njega kaže da je *čisto funkcionalan programski jezik*. U takve programske jezike spada i LispKit LISP.

Glava 2

Programski jezik LispKit LISP

Programski jezik LispKit LISP jedan je od mnogobrojnih dijalekata programskog jezika LISP. Pre bismo mogli reći da je LISP *porodica jezika*, koja sadrži pravo mnoštvo funkcionalnih programskih jezika, od kojih možemo pomenuti (danas aktuelne) Common LISP, Scheme, Frantz LISP i druge. Svi ovi jezici imaju velikih sličnosti, ali i razlika. Jedna od njihovih glavnih zajedničkih karakteristika jeste *s-izraz* – osnovna struktura podataka i gradivni blok svih LISP programa. Svaki od programskih jezika iz porodice LISP definiše neku svoju varijaciju s-izraza, ali osnova svih tih varijacija je u suštini ista.

U prvom odeljku daćemo definiciju s-izraza, onakvog kakav se koristi u programskom jeziku LispKit LISP. Sledeći odeljak posvetićemo opisu samog programskog jezika, a u poslednjem ćemo dati nekoliko primera programa koji ilustruju osobine i mogućnosti LispKit LISP-a. Detaljan opis znatno proširene verzije LispKit LISP-a može se naći u [2].

2.1 S-izraz

S-zraz je *tekst*, odnosno konačan niz *znakova*, koji zadovoljava određenu *sintaksu*, odnosno pravila koja ćemo u ovom odeljku formulisati.

Znakovi moraju pripadati skupu dozvoljenih znakova, tj. *azbuci*, u koju ulaze slova engleske abecede (od „`A`“ do „`Z`“), cifre (od „`0`“ do „`9`“) i specijalni znaci: „`-`“, „`(`“, „`)`“ i „`.`“. Znaci koji ne spadaju u azbuku, a koji se, neprimetno, pojavljuju u s-izrazima jesu tzv. *beline*, u koje spadaju razmaci i prelasci u novi red. Beline služe da razdvajaju različite konstrukcije u s-izrazima, kao što razmaci i novi redovi razdvajaju reči u tekstovima govornih jezika. Nećemo praviti razliku između velikih i malih slova, pa, na primer, prema znacima „`A`“ i „`a`“ odnosićemo se kao prema istom slovu. U ostatku teksta trudićemo se da sva slova u s-izrazima pišemo kao velika.

S-izrazi se grade od *atoma*, pa ćemo prvo njih definisati.

Atom u LispKit LISP-u može biti *simbolički* i *numerički*.

Simbolički atom jeste niz slova i cifara koji obavezno počinje slovom. Na primer, `A`, `ABC`, `KRUG9`, `G1F3` predstavljaju simboličke atome, dok `13`, `7NEBO`, `A.B` to nisu.

Numerički atom, u našoj verziji LispKit LISP-a, može biti samo **celobrojni atom**, a ovaj ćemo definisati kao niz cifara koji može (a i ne mora) počinjati znakom „`-`“. U cele atome spadaju `1`, `-1`, `13`, `-12321`, ali ne i `3.14`, `7NEBO`, `3A`. U drugim dijalektima LISP-a, pored celobrojnog, postoje i druge vrste numeričkih atoma (realni, kompleksni i dr.), ali mi se ovde njima nećemo baviti. Uvođenje drugih vrsta numeričkih atoma ostaje kao mogućnost proširenja ove verzije LispKit LISP jezika.

Sada već možemo preći na definiciju s-izraza.

S-izraz je

1. Atom, *ili*
2. *Tačkasti izraz*, oblika `(`s_1`.`s_2`)`, gde su s_1 i s_2 s-izrazi.

Kod tačkastog izraza za s_1 se često kaže da je *glava*, a s_2 *rep* izraza.

Ovo znači da su primeri simboličkih i numeričkih atoma koje smo dali ujedno i primeri s-izraza. Takođe, `(A.1)` je s-izraz, zato što je `(A.1)` tačkasti izraz kojem je simbolički atom `A` glava (a on je s-izraz), a numerički atom `1` (takođe s-izraz) rep. Slično, `(3.14)` jeste s-izraz jer predstavlja tačkasti izraz kojem su numerički atomi `3` i `14` glava i rep.

Pogledajmo jedan malo složeniji primer.

Primer 2.1 *Izraz* `(A.(B.(C.D)))` *je s-izraz. Zašto? Zato što on predstavlja tačkasti izraz čija je glava simbolički atom* `A`*, a rep tačkasti izraz* `(B.(C.D))` *čija je glava simbolički atom* `B`*, a rep tačkasti izraz* `(C.D)` *čija je glava simbolički atom* `C`*, a rep simbolički atom* `D`*. Time je definicija s-izraza zadovoljena.*

Tri simbolička atoma imaće specijalnu namenu u s-izrazima što će u ovom radu biti korišćeni. To su: `T`, koji će služiti za označavanje logičke vrednosti *tačno*, `F`, koji će obeležavati logičku vrednost *netačno*, i `NIL`, koji će označavati „prazan izraz“, o čemu će biti reči kasnije.

Kao što se da primetiti iz prethodnog primera, pravljenje iole većih struktura u vidu s-izraza dovodi do priličnog gomilanja tačaka i zagrada, što baš i ne ide u prilog njihovoj lepoti i preglednosti. Zato ćemo uvesti **dogovor o brisanju zagrada** (tačnije: tačaka, zagrada i atoma `NIL`), po kojem se niz znakova „`.(`“ može izostaviti (odnosno zameniti belinom), uz istovremeno izostavljanje odgovarajućeg znaka „`)`“. Niz znakova „`.NIL`“ se takođe može izostaviti.

Tako se, na primer, s-izraz `(A.(B.C))` može pisati kao `(A B.C)`. S-izraz iz primera 2.1 sada ima još *nekoliko* mogućih zapisa: `(A B.(C.D))`, `(A.(B C.D))` i `(A B C.D)`. Mi ćemo uglavnom težiti da s-izraze zapisujemo u obliku sa najmanjim brojem suvišnih znakova.

Po dogovoru o brisanju zagrada, s-izraz `(A.(B.(C.NIL)))` možemo zapisati i kao `(A B C)`. S-izraze ovog oblika zvaćemo *liste*. Definišimo ih malo preciznije.

Lista je

1. Simbolički atom `NIL`, *ili*

2. Tačkasti izraz oblika $(s.l)$, gde je s s-izraz, a l lista.

Da pojasnimo ranije rečeno, atom `NIL` u stvari predstavlja „praznu *listu*“, odnosno listu bez elemenata.

Pogledajmo još neke primere lista, zapisane u izvornom obliku i uz dogovor o brisanju zagrada:

`(A.NIL)`	`(A)`
`(1.(2.NIL))`	`(1 2)`
`(A.((B.C).NIL)`	`(A (B.C))`
`((A.(B.NIL)).((C.(D.NIL)).NIL))`	`((A B)(C D))`

Za kraj, daćemo i definiciju s-izraza, odnosno gramatiku koja definiše njegovu *sintaksu*, u proširenoj Bekus-Naur notaciji (EBNF). Ta gramatika predstavlja formalni zapis onog što smo do sad rekli o s-izrazima. Opis EBNF može se naći npr. u [2], a ova konkretna gramatika adaptirana je iz [7]. Imena metapromenljivih data su na engleskom jeziku, radi lakšeg dovođenja u vezu sa implementacijom u programskom jeziku Java, koja će biti opisana u glavi 4.

```
<letter>          ::=  'A' | 'B' | ... | 'Z'
<digit>           ::=  '0' | '1' | ... | '9'

<atom>            ::=  <symbolic>|<numeric>
<symbolic>        ::=  <letter> { <letter>|<digit> }
<numeric>         ::=  <integer>
<integer>         ::=  ['-'] digit { <digit> }

<SExpression>     ::=  <atom> | '(' <SExpressionList> ')'
<SExpressionList> ::=  <SExpression> |
                       <SExpression> '.' <SExpression> |
                       <SExpression> <SExpressionList>
```

2.2 Sintaksa i semantika LispKit LISP-a

Programi pisani u programskom jeziku LispKit LISP imaju formu s-izraza, koji mora da zadovoljava sintaksu koju ćemo opisati u ovom odeljku. Pravila sintakse formulisaćemo polazeći od gotovih *s-izraza*, a ne od *znakova* kao što smo to činili definišući s-izraz.

LispKit LISP *program* može biti jedan od nekoliko dozvoljenih vrsta LispKit LISP *izraza*. U nastavku odeljka LispKit LISP *izraz* označavaćemo, jednostavno, *izraz*, radi kratkoće zapisa. Takođe, umesto LispKit LISP *program* pisaćemo samo *program*. U LispKit LISP-u, izraza ima više vrsta, od kojih svaka ima svoju sintaksu, ali i *semantiku*, koja se ogleda u vrednosti koju izraz može da ima. Definišimo, dakle, sintaksu i semantiku izraza, preko opisa i primera svih vrsta izraza. U primerima ćemo dati ispravne izraze i, ukoliko je to moguće, njihove vrednosti, označene strelicom (⟶). Ideja da se programski jezik LispKit LISP opiše na ovakav način potekla je iz opisa jezika datog u [2].

Izraz u LispKit LISP-u, može biti:

Simbolički atom. Simbolički atomi u programima mogu označavati, odnosno imati vrednost, parametara funkcija, samih funkcija i, uopšte, bilo kojeg *izraza* koji im se pridruži. O ovom pridruživanju biće više reči kasnije.

Primeri: `A`, `JEDAN`, `JASAMIME`.

Izrazi koji slede predstavljaju pozive *elementarnih funkcija* LispKit LISP jezika, i imaju formu liste čiji je prvi element simbolički atom koji označava ime funkcije, a naredni element(i) predstavlja(ju) argument(e) poziva funkcije.

QUOTE izraz. QUOTE izraz ima oblik

(QUOTE <s-izraz>)

gde na mestu argumenta <s-izraz> može da stoji *bilo koji s-izraz*. Vrednost QUOTE izraza je baš <s-izraz>. Napomenimo da je QUOTE izraz *jedini* izraz, *jedino* mesto u programu gde se može pojaviti bilo koji s-izraz. Na ostalim se može pojavljivati isključivo *izraz*.

Primeri:

```
(QUOTE 1)                ⟶  1
(QUOTE ABC)              ⟶  ABC
(QUOTE (JA SAM LISTA))   ⟶  (JA SAM LISTA)
```

EQ izraz. EQ izraz ima oblik

(EQ <izraz> <izraz>)

gde na mestu argumenata mogu da se pojave bilo koji *izrazi*. Vrednost EQ izraza je simbolički atom `T`, ako važi da su *vrednosti* argumentata *jednaki atomi*, inače je `F`.

Primeri:

```
(EQ (QUOTE 1) (QUOTE 1))       ⟶  T
(EQ (QUOTE A) (QUOTE B))       ⟶  F
(EQ (QUOTE (A)) (QUOTE (B)))   ⟶  F
```

ADD izraz. ADD izraz ima oblik

(ADD <izraz> <izraz>)

gde su argumenti izrazi čije vrednosti treba da budu numerički atomi. Vrednost ADD izraza je numerički atom koji predstavlja *zbir* vrednosti argumenata.

Primeri:

```
(ADD (QUOTE 1) (QUOTE 1))    ⟶  2
(ADD (QUOTE -1) (QUOTE 1))   ⟶  0
```

SUB izraz, **MUL izraz**, **DIV izraz** i **REM izraz** definišu se analogno ADD izrazu, s tim što su im vrednosti redom *razlika*, *proizvod*, *celobrojni količnik* i *ostatak pri deljenju* vrednosti argumenata.

Primeri:

```
(SUB (QUOTE 1) (QUOTE 5))                    ⟶  -4
(MUL (QUOTE 2) (ADD (QUOTE 1) (QUOTE 1)))    ⟶  4
(DIV (QUOTE 7) (QUOTE 3))                    ⟶  2
(REM (QUOTE 7) (QUOTE 3))                    ⟶  1
```

LEQ izraz. LEQ izraz ima oblik

(LEQ <izraz> <izraz>)

gde su argumenti izrazi čije vrednosti treba da budu numerički atomi. Vrednost LEQ izraza je atom `T`, ako je vrednost prvog argumenta *manja ili jednaka* vrednosti drugog, inače je `F`.

Primeri:

```
(LEQ (QUOTE 1) (QUOTE 1))    ⟶  T
(LEQ (QUOTE 1) (QUOTE -1))   ⟶  F
```

CAR izraz. CAR izraz ima oblik

(CAR <izraz>)

gde je argument izraz čija vrednost treba da bude *tačkasti izraz*. Vrednost CAR izraza je *glava* vrednosti argumenta.

Primeri:

```
(CAR (QUOTE (A.B)))  ⟶  A
(CAR (QUOTE (1)))    ⟶  1
```

CDR izraz. CDR izraz ima oblik

(CDR <izraz>)

gde je argument izraz čija vrednost treba da bude *tačkasti izraz*. Vrednost CDR izraza je *rep* vrednosti argumenta.

Primeri:

```
(CDR (QUOTE (A.B)))  ⟶  B
(CDR (QUOTE (1)))    ⟶  NIL
```

ATOM izraz. ATOM izraz ima oblik

(ATOM <izraz>)

gde je argument bilo koji izraz. Vrednost ATOM izraza je simbolički atom `T` ako je vrednost argumenta atom, inače je `F`.

Primeri:

```
(ATOM (QUOTE (A.B)))  ⟶  F
(ATOM (QUOTE 1))      ⟶  T
```

CONS izraz. CONS izraz ima oblik

(CONS <izraz> <izraz>)

gde su argumenti bilo koji izrazi. Vrednost CONS izraza je *tačkasti izraz* čija je *glava* vrednost prvog argumenta, a *rep* vrednost drugog.

Primeri:

```
(CONS (QUOTE A) (QUOTE B))    ⟶  (A.B)
(CONS (QUOTE 1) (QUOTE NIL))  ⟶  (1)
```

IF izraz. IF izraz ima oblik

(IF <izraz> <izraz> <izraz>)

gde je prvi argument izraz čija je vrednost atom `T` ili `F`, a ostali argumenti su bilo koji izrazi. Vrednost IF izraza je jednaka vrednosti drugog argumenta ako je vrednost prvog argumenta `T`, u protivnom je jednaka vrednosti trećeg.

Primeri:

```
(IF (EQ (QUOTE A) (QUOTE A))
    (QUOTE (JEDNAKO))
    (QUOTE (NIJE JEDNAKO)))    ⟶  (JEDNAKO)
(IF (EQ (QUOTE 1) (QUOTE -1))
    (QUOTE (JEDNAKO))
    (QUOTE (NIJE JEDNAKO)))    ⟶  (NIJE JEDNAKO)
```

Sada slede izrazi koji predstavljaju složenije, ali i ključne koncepte programskog jezika LispKit LISP.

LAMBDA izraz. LAMBDA izraz ima oblik

(LAMBDA <lista> <izraz>)

gde argument <lista> treba da je *lista simboličkih atoma*, a <izraz> može biti bilo koji izraz, uz dodatnu mogućnost da je u njemu dozvoljeno pojavljivanje simboličkih atoma iz liste. Vrednost LAMBDA izraza je *funkcija* čiji su argumenti označeni simboličkim atomima iz liste, a vrednost funkcije računa se kao vrednost izraza <izraz>, u kojem mogu da se pojavljuju argumenti funkcije, odnosno simbolički atomi iz liste. Dobijena *funkcija* može se označiti nekim imenom, odnosno simboličkim atomom, kao u izrazima čiji opisi slede, a može se i direktno *pozvati*, kao u sledećim primerima.

Primeri:

```
( (LAMBDA (X) X)
  (QUOTE 13) )                    ⟶  13
( (LAMBDA (X) (ADD X (QUOTE 1)))
  (QUOTE 13) )                    ⟶  14
```

LET izraz. LET izraz ima oblik

(LET <izraz> { (<simbolički> . <izraz>) })

gde je prvi argument bilo koji *izraz*, a nakon njega sledi proizvoljan broj (nijedan ili više) *tačkastih izraza* čije su *glave* simbolični atomi, a *repovi* bilo koji izrazi. Vrednost LET izraza je vrednost prvog argumenta u kojem su sve pojave simboličkih atoma iz glava narednih tačkastih izraza zamenjene odgovarajućim izrazima iz repova. LET izraz se može koristiti da se imenuje npr. funkcija, kao uostalom i bilo koji drugi *izraz* – na taj način *simbolički atom* dobija značenje kao *izraz*.

Primeri:

```
(LET (ADD TWO TWO)
  (TWO . (QUOTE 2)))      ⟶  4
(LET (INC (QUOTE 13))
  (INC LAMBDA (X)
    (ADD X (QUOTE 1))))   ⟶  14
```

Primetimo da je u drugom primeru primenjen dogovor o brisanju zagrada u tačkastom izrazu, dok u prvom nije. Usvojićemo konvenciju da ćemo eliminisati suvišne zagrade u ovakvim izrazima samo u slučaju da je rep tačkastog izraza *LAMBDA izraz*. Ovo će važiti i u *LETREC izrazu*, koji sledi. *LETREC izraz* uvodimo zbog jednog važnog ograničenja LET izraza – a to je nemogućnost imenovanja, odnosno definisanja rekurzivnih funkcija, tj. funkcija u čijim izrazima koji određuju njihovu vrednost figurišu (pojavljuju se)

pozivi istih tih funkcija (neposredno ili preko drugih izraza i poziva drugih funkcija).

LETREC izraz. LETREC izraz ima oblik

(LETREC <izraz> { (<simbolički> . <LAMBDA izraz>) })

gde je prvi argument bilo koji *izraz*, a nakon njega sledi proizvoljan broj (nijedan ili više) *tačkastih izraza* čije su glave simbolički atomi, a repovi *LAMBDA izrazi*. Kao i kod LET izraza, vrednost LETREC izraza jeste vrednost prvog argumenta u kojem su sve pojave simboličkih atoma iz glava narednih tačkastih izraza zamenjene odgovarajućim izrazima iz repova. Ključna razlika u odnosu na LET izraz je ta što su imenovanja izraza iz repova sada vidljiva i u samim tim izrazima, a ne samo u izrazu koji predstavlja prvi argument LETREC izraza. Ova osobina omogućuje definisanje *rekurzivnih funkcija*, praktično glavnog oružja programskog jezika LispKit LISP.

Ograničenje da se u repovima tačkastih izraza smeju pojavljivati samo LAMBDA izrazi posledica je načina na koji ćemo implementirati jezik, i ne predstavlja veliku prepreku u programiranju, s obzirom na postojanje LET izraza. Primere LETREC izraza daćemo u sledećem odeljku.

Poziv funkcije. Poziv funkcije ima oblik (<funkcija> { <izraz> }), tj. predstavlja listu čiji je prvi element izraz čija je vrednost *funkcija*, a ostali elementi su argumenti poziva funkcije. <funkcija> je često samo *simbolički atom*, kojem je pomoću LET ili LETREC izraza pridružena neka funkcija, najčešće LAMBDA izraz. Pored simboličkog atoma, <funkcija> može biti i LAMBDA, LET ili LETREC izraz, odnosno sama definicija funkcije, ali se takvi direktni pozivi funkcije ređe sreću. Poziv funkcije ima puno sličnosti sa elementarnim izrazima, jedino je *programer* dužan da pozvane funkcije definiše, odnosno imenuje. Primeri poziva funkcija dati su među primerima LAMBDA i LET izraza, a biće ih i u sledećem odeljku.

Sada nam je lako dati definiciju *programa*.

Program je LETREC izraz, ili LET izraz, ili LAMBDA izraz. Program u LispKit LISP-u mora biti *funkcija*, a jedino ovi izrazi imaju osobinu da im vrednost može biti funkcija. U praktičnoj implementaciji LispKit LISP-a program se, na ovaj ili onaj način, „poziva“, kao funkcija, pri čemu mu se prosleđuju odgovarajući argumenti, i time počinje njegovo izvršavanje.

Na kraju, slično kao pri definisanju s-izraza, daćemo sumiranu formalnu definiciju sintakse LispKit LISP *programa*. Koristićemo notaciju koja liči na EBNF, a čije smo primere već videli u ovom odeljku. Razlika u odnosu na EBNF je intuitivna – notacija se odnosi na *s-izraze*, pa znakovi „(“, „)“ i „.“ označavaju *tačkaste izraze*, odnosno *liste*, a imena koja ne obeležavaju

metapromenljive (imena van „<“ i „>“) *simboličke atome*. Kao i kod s-izraza, imena metapromenljivih navodćemo na engleskom jeziku.

```
<symbolicList>  ::=  NIL | (<symbolic> {<symbolic>})
<exp>           ::=  <symbolic>|<quoteExp>|<eqExp>|
                     <addExp>|<subExp>|<mulExp>|<divExp>|
                     <remExp>|<leqExp>|<carExp>|<cdrExp>|
                     <atomExp>|<consExp>|<ifExp>|<lambdaExp>|
                     <letExp>|<letrecExp>|<callExp>
<quoteExp>      ::=  (QUOTE <SExpression>)
<eqExp>         ::=  (EQ <exp>)
<addExp>        ::=  (ADD <exp> <exp>)
<subExp>        ::=  (SUB <exp> <exp>)
<mulExp>        ::=  (MUL <exp> <exp>)
<divExp>        ::=  (DIV <exp> <exp>)
<remExp>        ::=  (REM <exp> <exp>)
<leqExp>        ::=  (LEQ <exp> <exp>)
<carExp>        ::=  (CAR <exp>)
<cdrExp>        ::=  (CDR <exp>)
<atomExp>       ::=  (ATOM <exp>)
<consExp>       ::=  (CONS <exp> <exp>)
<ifExp>         ::=  (IF <exp> <exp> <exp>)
<lambdaExp>     ::=  (LAMBDA <symbolicList> <exp>)
<letExp>        ::=  (LET <exp> { (<symbolic> . <exp>) })
<letrecExp>     ::=  (LETREC <exp>
                     { (<symbolic> . <lambdaExp>) })
<callExp>       ::=  (<function> { <exp> })
<function>      ::=  <symbolic>|<lambdaExp>|<letExp>|<letrecExp>
<program>       ::=  <letrecExp>|<letExp>|<lambdaExp>
```

2.3 Primeri programa

U ovom odeljku obradićemo nekoliko jednostavnijih primera LispKit LISP programa. Mnogo drugih, mahom složenijih, primera može se naći u [2].

Primer 2.2 *Neka želimo napisati program koji uvećava uneti broj za* 1. *Jedno rešenje u LispKit LISP-u moglo bi biti:*

```
(LAMBDA (X)
  (ADD X (QUOTE 1))
)
```

Ovaj program, dakle, predstavlja funkciju jednog argumenta `X`*, koja vraća numerički atom dobijen povećanjem* `X` *za* 1. *Ako, na primer, „pozovemo“*

ovaj program i kao argument mu prosledimo broj 7*, dobićemo posle izvršenja programa rezultat* 8*.*

Pogledajmo još jedno rešenje:

```
(LET INC
  (INC LAMBDA (X)
    (ADD X (QUOTE 1))
  )
)
```

Sada smo u programu definisali ime `INC`*, dodelivši mu kao vrednost funkciju. Ovaj program predstavlja vrednost izraza* `INC`*, a to je funkcija, što znači da i ovaj program možemo pozvati.*

Primer 2.3 *Program koji računa faktorijel unetog broja napisaćemo na sledeći način:*

```
(LETREC FAC
  (FAC LAMBDA (X)
    (IF (EQ X (QUOTE 0))
        (QUOTE 1)
        (MUL X (FAC (SUB X (QUOTE 1))))
    )
  )
)
```

Ovde nam je program funkcija `FAC` *koja vraća broj* 1 *ako je argument* `X` *jednak* 0*, inače vraća broj dobijen množenjem* `X`*-a sa faktorijelom broja* $\texttt{X} - 1$*. Time smo zadovoljili definiciju faktorijela iz primera 1.3 na strani 9. Ako pozovemo ovaj program sa argumentom, na primer,* 4*, trebalo bi da dobijemo rezultat* 24*. Tok ovog izračunavanja mogli bismo prikazati nizom jednakosti:*

$$\begin{aligned}
\texttt{(FAC 4)} &= 4 \cdot \texttt{(FAC 3)} \\
&= 4 \cdot 3 \cdot \texttt{(FAC 2)} \\
&= 4 \cdot 3 \cdot 2 \cdot \texttt{(FAC 1)} \\
&= 4 \cdot 3 \cdot 2 \cdot 1 \cdot \texttt{(FAC 0)} \\
&= 4 \cdot 3 \cdot 2 \cdot 1 \cdot 1 \\
&= 24
\end{aligned}$$

Primer 2.4 *Program koji računa dužinu liste u LispKit LISP-u možemo formulisati ovako:*

```
(LETREC LENGTH
  (LENGTH LAMBDA (L)
    (IF (EQ L (QUOTE NIL))
        (QUOTE 0)
        (ADD (QUOTE 1) (LENGTH (CDR L)))
    )
  )
)
```

Ako je lista prazna, odnosno `NIL`*, funkcija* `LENGTH` *vraća* 0*, inače vraća dužinu liste bez prvog elementa (repa liste), uvećanu za* 1*. Npr. ako ovom programu prosledimo argument* `(A B C)`*, dobićemo rezultat* 3*. Prikažimo tok ovog računanja:*

$$\begin{aligned} \texttt{(LENGTH (A B C))} &= 1 + \texttt{(LENGTH (B C))} \\ &= 1 + 1 + \texttt{(LENGTH (C))} \\ &= 1 + 1 + 1 + \texttt{(LENGTH NIL)} \\ &= 1 + 1 + 1 + 0 \\ &= 3 \end{aligned}$$

Glava 3

SECD mašina

SECD mašinu možemo posmatrati kao virtuelnu (hipotetičku, zamišljenu) mašinu čija je namena da izvršava programe pisane na jeziku koji smo nazvali kôd SECD mašine. Taj programski jezik je, po svojim osobinama, dosta niskog nivoa, tako da ako zamislimo da SECD mašina stvarno postoji[1], kod SECD mašine bi bio njen *mašinski jezik*. Zbog takve tesne veze opisaćemo istovremeno i arhitekturu SECD mašine i njen mašinski jezik, onakve kakvi su prikazani u [1, 7].

SECD mašina dobila je ime po četiri njena *registra*:

s	stek[2]	koristi se za smeštanje međurezultata prilikom izračunavanja vrednosti izraza
e	okruženje[3]	koristi se za čuvanje vrednosti *promenljivih* prilikom izračunavanja
c	kontrolna lista[4]	sadrži mašinski program koji se izvršava
d	skladište[5]	služi za čuvanje vrednosti ostalih registara prilikom poziva funkcija.

Svaki od registara SECD mašine sadrži jedan s-izraz. Reći čemo da je *stanje* SECD mašine jedinstveno određeno sadržajem njena četiri registra. *Promene stanja* u tom slučaju možemo označavati

$$s\ \ e\ \ c\ \ d\ \rightarrow\ s'\ \ e'\ \ c'\ \ d'$$

gde s-izrazi sa leve strane označavaju stanje *pre*, a s-izrazi sa desne strane stanje *posle* promene.

Elementrane naredbe, odnosno *instrukcije* SECD mašine iniciraju promene stanja SECD mašine, tako da izvršavanje mašinskog programa SECD

[1] A mi je zamišljamo, dakle, ona postoji.
[2] Engl. *stack*.
[3] Engl. *environment*.
[4] Engl. *control list*.
[5] Engl. *dump*.

mašine nije ništa drugo do konačan niz *promena stanja*. *Instrukcije* ćemo opisati pomoću promena stanja koje one iniciraju pri svom izvršavanju.

Za početak, navedimo instrukcije SECD mašine, koje ćemo označiti simboličkim atomima, kao i njihove kratke opise:

1. LD punjenje
2. LDC punjenje konstante
3. LDF punjenje funkcije
4. AP primena funkcije
5. RTN povratak
6. DUM kreiranje fiktivne ćelije tj. formalnog okruženja
7. RAP rekurzivna primena funkcije
8. SEL selekcija podliste za dalje izvršavanje
9. JOIN vraćanje glavnom toku izvršavanja
10. CAR *glava* elementa s vrha steka
11. CDR *rep* elementa s vrha steka
12. ATOM ispitivanje da li je element s vrha steka atom
13. CONS formiranje tačkastog izraza od dva elementa sa steka
14. EQ ispitivanje da li su dva elementa sa steka jednaki atomi
15. ADD sabiranje dva elementa s vrha steka
16. SUB oduzimanje dva elementa s vrha steka
17. MUL množenje dva elementa s vrha steka
18. DIV deljenje dva elementa s vrha steka
19. REM ostatak pri deljenju dva elementa s vrha steka
20. LEQ relacija „manje ili jednako“ dva elementa sa steka
21. STOP zaustavljanje izvršavanja programa.

Ovde je važno napomenuti da ova imena označavaju *instrukcije* SECD mašine, i ne treba ih mešati sa imenima elementarnih funkcija LispKit LISP-a. SECD mašina je specijalno dizajnirana tako da se LispKit LISP programi prevode u njen mašinski jezik, što ima za posledicu da važi direktna korespondencija između konstrukcija LispKit LISP-a i instrukcija SECD mašine. Da bi se ovo istaklo, korišćena su identična imena gde god je to moguće. U praksi, odnosno u našoj *implementaciji* SECD mašine, instrukcije ćemo prosto označavati njihovim rednim brojevima. Ali, o tome više u glavi 4.

Opišimo sada svaku instrukciju.

ADD instrukcija inicira promenu stanja oblika

$$(a\ b.s)\ \ e\ \ (\texttt{ADD}.c)\ \ d\ \rightarrow\ (b+a.s)\ \ e\ \ c\ \ d$$

što znači da ako je *kontrolna lista* oblika $(\texttt{ADD}.c)$, gde je c s-izraz, *stek* oblika $(a\ b.s)$, gde je s s-izraz, a a i b numerički atomi, tada je u stanju posle izvršenja ADD instrukcije stek oblika $(b+a.s)$, gde je $b+a$ numerički atom koji se dobija kao zbir b i a, a kontrolna lista je oblika c.

Primer:

$$(5\ 8.s)\ \ e\ \ (\texttt{ADD}.c)\ \ d\ \rightarrow\ (13.s)\ \ e\ \ c\ \ d$$

SUB, **MUL**, **DIV** i **REM** instrukcije izvršavaju se analogno ADD instrukciji:

$$(a\ \ b.s)\ \ e\ \ (\texttt{SUB}.c)\ \ d\ \rightarrow\ (b-a.s)\ \ e\ \ c\ \ d$$

$$(a\ \ b.s)\ \ e\ \ (\texttt{MUL}.c)\ \ d\ \rightarrow\ (b\cdot a.s)\ \ e\ \ c\ \ d$$

$$(a\ \ b.s)\ \ e\ \ (\texttt{DIV}.c)\ \ d\ \rightarrow\ (b/a.s)\ \ e\ \ c\ \ d$$

$$(a\ \ b.s)\ \ e\ \ (\texttt{REM}.c)\ \ d\ \rightarrow\ (b\%a.s)\ \ e\ \ c\ \ d$$

kao i **EQ** i **LEQ** instrukcije:

$$(a\ \ b.s)\ \ e\ \ (\texttt{EQ}.c)\ \ d\ \rightarrow\ (b=a.s)\ \ e\ \ c\ \ d$$

$$(a\ \ b.s)\ \ e\ \ (\texttt{LEQ}.c)\ \ d\ \rightarrow\ (b\leq a.s)\ \ e\ \ c\ \ d$$

gde je izraz $b = a$ logička vrednost `T` ako su a i b *jednaki atomi*, inače je `F`, a izraz $b \leq a$ je logička vrednost `T` ako je numerički atom b manji ili jednak numeričkom atomu a, inače je `F`.

LDC instrukcija se izvršava u obliku:

$$s\ \ e\ \ (\texttt{LDC}\ a.c)\ \ d\ \rightarrow\ (a.s)\ \ e\ \ c\ \ d$$

gde je a s-izraz. Ova instrukcija jednostavno prebacuje konstantu a na vrh steka.

Sada već možemo ilustrovati rad SECD mašine na konkretnom primeru.

Primer 3.1 *Logički izraz* $1-2\cdot 3 = 4$ *u LispKit LISP-u se može zapisati kao* `(EQ (SUB (QUOTE 1) (MUL (QUOTE 2)(QUOTE 3))) (QUOTE 4))`*, a ovaj se prevodi u SECD program* `(LDC 1 LDC 2 LDC 3 MUL SUB LDC 4 EQ)`*. Posmatrajmo izvršavanje ovog programa korak po korak, odnosno iz stanja u stanje.*

s	e	`(LDC 1 LDC 2 LDC 3 MUL SUB LDC 4 EQ)`	d
`(1.`s`)`	e	`(LDC 2 LDC 3 MUL SUB LDC 4 EQ)`	d
`(2 1.`s`)`	e	`(LDC 3 MUL SUB LDC 4 EQ)`	d
`(3 2 1.`s`)`	e	`(MUL SUB LDC 4 EQ)`	d
`(6 1.`s`)`	e	`(SUB LDC 4 EQ)`	d
`(-5.`s`)`	e	`(LDC 4 EQ)`	d
`(4 -5.`s`)`	e	`(EQ)`	d
`(F.`s`)`	e	`NIL`	d

Iz ovog primera se vidi kako se sledeća promena stanja, odnosno instrukcija, određuje jednostavno na osnovu prvog elementa kontrolne liste. Ovo pravilo važi u svim SECD programima.

STOP instrukcija se izvršava ovako:

$$s\ \ e\ \ \texttt{(STOP)}\ \ d\ \rightarrow\ s\ \ e\ \ \texttt{(STOP)}\ \ d$$

odakle se jasno vidi da se posle STOP instrukcije ne može izvršiti ni jedna druga instrukcija. U praktičnim implementacijama STOP instrukcija dovodi do prekida rada interpretera SECD mašine.

CONS instrukcija se izvršava na sledeći način:

$$\texttt{(}a\ \ b\texttt{.}s\texttt{)}\ \ e\ \ \texttt{(CONS.}c\texttt{)}\ \ d\ \rightarrow\ \texttt{((}a\texttt{.}b\texttt{).}s\texttt{)}\ \ e\ \ c\ \ d$$

a **CAR** i **CDR** instrukcije ovako:

$$\texttt{((}a\texttt{.}b\texttt{).}s\texttt{)}\ \ e\ \ \texttt{(CAR.}c\texttt{)}\ \ d\ \rightarrow\ \texttt{(}a\texttt{.}s\texttt{)}\ \ e\ \ c\ \ d$$

$$\texttt{((}a\texttt{.}b\texttt{).}s\texttt{)}\ \ e\ \ \texttt{(CDR.}c\texttt{)}\ \ d\ \rightarrow\ \texttt{(}b\texttt{.}s\texttt{)}\ \ e\ \ c\ \ d$$

gde su a i b s-izrazi, a $\texttt{(}a\texttt{.}b\texttt{)}$ je tačkasti izraz čija je glava a, a rep b.

ATOM instrukcija se izvršava u obliku:

$$\texttt{(}a\texttt{.}s\texttt{)}\ \ e\ \ \texttt{(ATOM.}c\texttt{)}\ \ d\ \rightarrow\ \texttt{(}l\texttt{.}s\texttt{)}\ \ e\ \ c\ \ d$$

gde je a s-izraz, a l je simbolički atom `T` ako je a atom, inače je `F`.

Sada su na redu nešto složenije instrukcije, čiji rad zahteva malo dodatnih objašnjenja.

LD instrukcija služi za prebacivanje veličina, odnosno s-izraza iz *okruženja* na vrh *steka*. Jedan primer okruženja mogao bi biti

```
((3 17)(A B))
```

Ovde je okruženje lista koja sadrži dve podliste, od kojih svaka sadži dva elementa. Svaki element je odeđen sa dva broja: rednim brojem podliste u kojoj se nalazi, u odnosu na okruženje, i rednim brojem samog elementa u okviru podliste. Ova dva broja možemo staviti u tačkasti izraz, pa tako `(0.0)` određuje element `3`, `(0.1)` određuje element `17`, `(1.0)` određuje `A` i `(1.1)` određuje `B`. Okruženje može imati proizvoljan broj podlisti, a svaka podlista proizvoljan broj elemenata, koji su s-izrazi (pod „proizvoljnim brojem“ podrazumevamo ceo broj ≥ 0).

Označimo sada sa $locate(i, e)$ funkciju koja za tačkasti izraz i čiji su glava i rep celi brojevi (≥ 0) i okruženje e vraća element okruženja određen sa i. Tada izvršavanje instrukcije LD možemo definisati sa:

$$s\ \ e\ \ \texttt{(LD}\ \ i\texttt{.}c\texttt{)}\ \ d\ \rightarrow\ \texttt{(}a\texttt{.}s\texttt{)}\ \ e\ \ c\ \ d$$

gde je $a = locate(i, e)$.

Primer:

$$s\ \texttt{((3 17)(A B))}\ \texttt{(LD (0.1).}c\texttt{)}\ d \rightarrow \texttt{(17.}s\texttt{)}\ \texttt{((3 17)(A B))}\ c\ d$$

SEL i **JOIN** instrukcije rade ruku pod ruku prilikom uslovnog izvršavanja delova programa. Njihovo izvršavanje ima oblik:

$$\texttt{(}a\texttt{.}s\texttt{)}\ e\ \texttt{(SEL}\ c_T\ c_F\texttt{.}c\texttt{)}\ d \rightarrow s\ e\ c_a\ \texttt{(}c\texttt{.}d\texttt{)}$$

$$s\ e\ \texttt{(JOIN)}\ \texttt{(}c\texttt{.}d\texttt{)} \rightarrow s\ e\ c\ d$$

SEL instrukcija očekuje na vrhu steka simbolički atom `T` ili `F`, i na osnovu njega učitava kontrolnu listu c_T ili c_F, a prethodnu kontrolnu listu c čuva na skladištu. Od kontrolnih listi c_T i c_F se očekuje da se završavaju instrukcijom JOIN, koja vraća tok izvršavanja na sačuvanu kontrolnu listu.

Primer: Fragment programa, odnosno kontrolna lista

```
(LD (0.0) SEL (LD (0.1) JOIN) (LD (0.2) JOIN) STOP)
```

će na osnovu elementa okruženja na poziciji `(0.0)` učitati na stek element okruženja sa pozicije `(0.1)` ili `(0.2)`.

Priča oko instrukcija koje operišu sa funkcijama nešto je složenija. Za početak, prisetimo se prethodnog poglavlja, gde smo pri opisu LispKit LISP-a rekli da vrednost nekih LispKit LISP izraza, kao što je LAMBDA izraz, može biti *funkcija* (vidi str. 21). Sada ćemo objasniti šta to znači u ovoj našoj verziji LispKit LISP sistema. Spomenuta *funkcija* je u SECD mašini predstavljena *zatvorenjem*, odnosno strukturom koju čine *telo* funkcije (tj. njen mašinski program – kontrolna lista) i *okruženje* u kojem je ta funkcija definisana. Kao strukturu uzećemo, jednostavno, tačkasti izraz.

LDF instrukcija je zadužena za kreiranje zatvorenja. Izvršava se ovako:

$$s\ e\ \texttt{(LDF}\ c\texttt{'.}c\texttt{)}\ d \rightarrow \texttt{((}c\texttt{'.}e\texttt{).}s\texttt{)}\ e\ c\ d$$

gde je *c'* kontrolna lista koja predstavlja telo funkcije. Zatvorenje se smešta na vrh steka i čeka dalju obradu, pri kojoj obično dospe u okruženje iz kojeg kasnije biva kopirano na stek instrukcijom LD i nakon toga pozvano kao funkcija.

Primer: Stanje

```
(0) ((3 7)(A)) (LDF (LD (1.1) RTN) LD (0.1)) NIL
```

preći će u stanje

```
(((LD (1.1) RTN).((3 7)(A))) 0) ((3 7)(A)) (LD (0.1)) NIL
```

gde je `((LD (1.1) RTN).((3 7)(A)))` zatvorenje koje je formirano na vrku steka.

AP instrukcija poziva funkciju, odnosno zatvorenje, na sledeći način:

$$((c'.e')\ v.s)\ e\ (\texttt{AP}.c)\ d \rightarrow \texttt{NIL}\ (v.e)\ c'\ (s\ e\ c.d)$$

pri čemu se na steku očekuju zatvorenje $(c'.e')$ i lista v konkretnih argumenata funkcije. Nakon što se sadržaj svih registara SECD mašine sačuva na skladištu, stek se prazni, a lista argumenata funkcije dodaje se kao prva podlista u okruženje, odakle će argumentima biti pristupano tokom izvršavanja tela funkcije. Očekuje se da se nova kontrolna lista c', odnosno telo funkcije, završava instrukcijom RTN.

Primer: Stanje

$$(((\texttt{LD(1.1)LD(0.0)ADD RTN}).((3\ 7)(A)))(6)0)\ ((2\ B))\ (\texttt{AP STOP})\ d$$

gde je `((LD(1.1)LD(0.0)ADD RTN).((3 7)(A)))` zatvorenje na vrhu steka, `(6)` lista argumenata (sa samo jednim elementom), a `0` ostatak steka, prelazi u:

$$\texttt{NIL}\ ((6)(3\ 7)(A))\ (\texttt{LD(1.1)LD(0.0)ADD RTN})\ ((0)((2\ B))(\texttt{STOP}).d)$$

RTN instrukcija upotpunjuje instrukciju AP tako što omogućuje povratak iz funkcije u glavni tok programa, sačuvan na skladištu. Izvršavanje RTN instrukcije ima oblik:

$$(a)\ e'\ (\texttt{RTN})\ (s\ e\ c.d) \rightarrow (a.s)\ e\ c\ d$$

Odavde se vidi da RTN instrukcija očekuje da nađe tačno jedan element na steku, kojeg postavlja na vrh sačuvanog steka. Taj element predstavlja *provratnu vrednost* izvršene funkcije.

Primer: Nakon izvršenja LD-ova i ADD-a iz prethodnog primera, dolazimo u stanje

$$(13)\ ((6)(3\ 7)(A))\ (\texttt{RTN})\ ((0)((2\ B))(\texttt{STOP}).d)$$

koje nakon izvršenja instrukcije RTN postaje

$$(13\ 0)\ ((2\ B))\ (\texttt{STOP})\ d$$

i povratna vrednost funkcije, na vrhu steka, je `13`.

DUM i **RAP** instrukcije sarađuju pri realizaciji rekurzivnih poziva funkcija. DUM instrukcija kreira fiktivni element na vrhu okruženja:

$$s\ e\ (\texttt{DUM}.c)\ d \rightarrow s\ (\Omega.e)\ c\ d$$

gde je sa Ω obeležen taj fiktivni element, koji instrukcija RAP kasnije koristi u svoje svrhe. Definišimo sada (pseudo)funkciju $rplaca(x, y)$, koja kao argumente prima s-izraze x i y, od kojih x mora biti tačkasti izraz, a rezultat joj je tačkasti izraz identičan sa x, s tim što mu je glava zamenjena y-om. Međutim, od ove funkcije zahtevaćemo da ima još jednu osobinu, koja joj opravdano donosi prefiks „pseudo“. Ta osobina je da se u toku njenog izvršavanja argument x *menja*, tako što mu se glava zameni y-om. To jest, proceduralno govoreći, promenljiva prosleđena funkciji *rplaca* kao argument x biće žrtva sporednog efekta, i biće izmenjena na opisan način[6]. Sada izvršavanje instrukcije RAP teče vrlo slično instrukciji AP:

$$\texttt{((}c\texttt{'.}e\texttt{')}\ v\texttt{.}s\texttt{)}\ \texttt{(}\Omega\texttt{.}e\texttt{)}\ \texttt{(RAP.}c\texttt{)}\ d \rightarrow \texttt{NIL}\ rplaca(e', v)\ c'\ \texttt{(}s\ e\ c\texttt{.}d\texttt{)}$$

uz napomenu da će, kada RAP instrukcija bude pozivana, uvek važiti relacija e'=$\texttt{(}\Omega\texttt{.}e\texttt{)}$. Izuzimajući tu osobinu, RAP instrukcija, ako je okruženje prethodno prošireno instrukcijom DUM, praktično se ponaša kao i AP. Ključnu razliku pravi opisani sporedni efekat pseudofunkcije *rplaca*. Ovo će biti pojašnjeno u narednom odeljku.

3.1 Prevođenje LispKit LISP-a u kod SECD mašine

Ostalo nam je još da damo pravila po kojima će se LispKit LISP programi, odnosno *izrazi*, prevoditi u odgovarajući kod, tj. niz instrukcija SECD mašine. Opis ovih pravila može se naći u [1, 7].

Nizovi SECD instrukcija dobijeni prevođenjem ispravnog LispKit LISP izraza treba da zadovoljavaju sledeću osobinu.

Osobina prevedenih izraza. Kod dobijen prevođenjem LispKit LISP izraza, ako se učita u kontrolnu listu SECD mašine i izvrši, ostavlja vrednost tog izraza na vrhu steka. Očekuje se da su vrednosti slobodnih promenljivih izraza učitane na odgovarajuće pozicije u okruženju. Pored vrednosti izraza na vrhu steka, stek ostaje nepromenjen nakon izvršavanja koda. Okruženje i skladište imaju iste vrednosti na početku i kraju izvršavanja.

Preciznije, ako je c kod dobijen prevođenjem nekog izraza, a e okruženje koje sadrži vrednosti promenljivih iz izraza, tada za bilo koji stek s i skladište d važi promena stanja

$$s\ e\ c\ d \Rightarrow \texttt{(}x\texttt{.}s\texttt{)}\ e\ \texttt{NIL}\ d$$

[6] Argument x prenosi se strategijom „poziv po imenu“.

gde je x vrednost koju je izračunao c. Sa $\Rightarrow$ označili smo da je došlo do jedne ili više promena stanja opisanih u prethodnom odeljku.

Operator „|“ koristićemo da označimo spoj dve liste. Na primer, `(A B)|(C D)` je sinonim za `(A B C D)`. Ovaj operator je asocijativan, tj. za liste x, y i z možemo slobodno pisati $x|y|z$, bez razmišljanja o tome da li će se operator prvo primeniti na x i y ili na y i z. Na primer, `(LDC NIL)|(LDC 1)|(CONS)` je isto što i `(LDC NIL LDC 1 CONS)`.

U toku analize LispKit LISP izraza formiraćemo listu promenljivih, odnosno imena koja se pojavljuju u LAMBDA, LET i LETREC izrazima. Ta lista imena imaće istu strukturu kao i *okruženje* koje se formira u toku izvršavanja SECD mašinskog programa, s tim što lista imena sadrži *imena*, a okruženje sadrži *vrednosti* promenljivih iz LispKit LISP izraza. Na primer,

```
((A) (X Y) (APPEND REV DUP))
```

predstavlja jednu mogućnu listu imena. Ona se može dobiti prilikom analize LispKit LISP izraza

```
(LETREC ....
  (APPEND LAMBDA (X Y)
    (LET ....
      (A ....)
    )
  )
  (REV LAMBDA .....)
  (DUP LAMBDA .....)
)
```

i to u trenutku analize ugneždenog LET izraza. Tačkicama smo označili izostavljene delove izraza, nama trenutno nebitne. Prilikom izvršavanja SECD koda dobijenog prevođenjem ovog LETREC izraza, odnosno programa, navedenoj listi imena moglo bi odgovarati okruženje

```
((13) ((1 2) (3 4 5)) (z1 z2 z3))
```

gde `13` predstavlja vrednost promenljive `A`, liste `(1 2)` i `(3 4 5)` su vrednosti promenljivih `X` i `Y`, a z_1, z_2 i z_3 su zatvorenja koja odgovaraju funkcijama pridruženim promenljivama `APPEND`, `REV` i `DUP`.

Listu imena koristićemo prilikom prevođenja LispKit LISP izraza kada u izrazu naiđemo na neko ime, odnosno simbolički atom. Tada će biti potrebno to ime naći u listi imena, i izračunati njegovu poziciju u toj listi. U tu svrhu definisaćemo funkciju $location(x, n)$, koja u listi imena n pronalazi element x i vraća tačkasti izraz čija je glava redni broj podliste od n u kojoj se x nalazi,

a rep je redni broj x-a u toj podlisti. Taj tačkasti izraz će, pri izvršavanju SECD programa, koristiti instrukcija LD.

Opišimo sada kako se svaka vrsta LispKit LISP izraza prevodi. Pri tom ćemo sa

$$e{*}n$$

označavati kod dobijen prevođenjem izraza e u odnosu na listu imena n. Ovo će biti pojašnjeno pravilom prevođenja LispKit LISP izraza koji je najprostiji po strukturi – **simbolički atom**:

$$x{*}n = \texttt{(LD } i\texttt{)}, \text{ gde je } i = location(x, n)$$

Znači, za proizvoljni LispKit LISP izraz e, $e{*}n$ predstavlja SECD kod koji se dobije prevođenjem tog izraza (po pravilima koja ćemo navesti), s tim da se sve pojave imena (tj. simboličkih atoma) u izrazu e prevode po navedenom pravilu.

QUOTE izraz se takođe jednostavno prevodi:

$$\texttt{(QUOTE } s\texttt{)}{*}n = \texttt{(LDC } s\texttt{)}$$

EQ izraz se prevodi na sledeći način:

$$\texttt{(EQ } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(EQ)}$$

što znači da će se pri izvršavanju dobijenog koda prvo izvršiti kôd $e_1{*}n$ i ostaviti rezultat izraza e_1 na steku, zatim će se izvršiti kôd $e_2{*}n$ i takođe ostaviti rezultat na steku, i na kraju će instrukcija EQ biti primenjena na ta dva elementa s vrha steka. Opet napomenimo da „`EQ`“ sa leve strane jednakosti označava *elementrnu funkciju* LispKit LISP-a, dok „`EQ`“ sa desne predstavlja *instrukciju* SECD mašine.

ADD, **SUB**, **MUL**, **DIV**, **REM** i **LEQ** izrazi prevode se analogno:

$$\texttt{(ADD } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(ADD)}$$

$$\texttt{(SUB } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(SUB)}$$

$$\texttt{(MUL } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(MUL)}$$

$$\texttt{(DIV } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(DIV)}$$

$$\texttt{(REM } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(REM)}$$

$$\texttt{(LEQ } e_1\ e_2\texttt{)}{*}n = e_1{*}n | e_2{*}n | \texttt{(LEQ)}$$

CAR, **CDR** i **ATOM** izrazi imaju jedan argument:

$$\texttt{(CAR } e\texttt{)}*n = e*n|\texttt{(CAR)}$$

$$\texttt{(CDR } e\texttt{)}*n = e*n|\texttt{(CDR)}$$

$$\texttt{(ATOM } e\texttt{)}*n = e*n|\texttt{(ATOM)}$$

CONS izraz specifičan je po tome što se kod generiše u drugačijem redosledu:

$$\texttt{(CONS } e_1\ e_2\texttt{)}*n = e_2*n|e_1*n|\texttt{(CONS)}$$

Da bismo ilustrovali do sada navedena pravila, pogledajmo primer.

Primer 3.2 *Posmatrajmo prevođenje jednog konkretnog LispKit LISP izraza, primenjujući do sada obrađena pravila prevođenja, redom, na njegovim podizrazima:*

```
    (ADD (CAR X) (QUOTE 1))*((X Y))
=   (CAR X)*((X Y)) | (QUOTE 1)*((X Y)) | (ADD)
=   X*((X Y)) | (CAR) | (LDC 1) | (ADD)
=   (LD (0.0)) | (CAR) | (LDC 1) | (ADD)
=   (LD (0.0) CAR LDC 1 ADD)
```

IF izraz realizuje se pomoću instrukcija SEL i JOIN:

$$\texttt{(IF } e_1\ e_2\ e_3\texttt{)}*n = e_1*n|\texttt{(SEL } e_2*n|\texttt{(JOIN) } e_3*n|\texttt{(JOIN))}$$

Prilikom izvršavanja koda dobijenog prevođenjem IF izraza prvo se izvrši kôd e_1*n, a zatim na osnovu rezultata koji je ovaj ostavio na steku (atom `T` ili `F`), instrukcija SEL prebacuje tok izvršavanja na e_2*n ili e_3*n, kojima je zbog povratka u glavni tok izvršavanja na kraj dodata instrukcija JOIN.

Primer:

```
(ADD Y (IF (LEQ X Y) X (QUOTE 1)))*((X Y))
```

daje sledeći kod SECD mašine:

```
(LD (0.1)
    LD (0.1) LEQ SEL (LD (0.0) JOIN) (LDC 1 JOIN)
 ADD)
```

gde je deo koda dobijen prevođenjem IF izraza posebno uvučen.

LAMBDA izraz se prirodno prevodi u kod koji sadrži instrukcije LDF i RTN:

$$\texttt{(LAMBDA (}x_1 \ldots x_k\texttt{) } e\texttt{)}*n = \texttt{(LDF } e*((x_1 \ldots x_k).n)|\texttt{(RTN))}$$

Treba obratiti pažnju na to da se izraz e, tj. telo funkcije, prevodi u odnosu na listu imena n proširenu imenima parametara funkcije. Na kraj tela funkcije stavlja se instrukcija RTN, da obezbedi povratak iz funkcije kada ona bude pozvana.

Primer:

`(LAMBDA (X) (ADD X Y))*((X Y))`

prelazi u

`(LDF (LD (0.0) LD (1.1) ADD RTN))`

Poziv funkcije prevodi se na sledeći način:

$$(e\ \ e_1 \ldots e_k)*n = \texttt{(LDC NIL)}|e_k*n|\texttt{(CONS)}|\ldots|e_1*n|\texttt{(CONS)}|e*n|\texttt{(AP)}$$

Analizirajmo izvršavanje dobijenog koda. Jasno, `(LDC NIL)` na stek stavlja `NIL`. Zatim na stek dolazi rezultat izvršavanja koda e_k*n, obeležimo ga sa v_k, a potom instrukcija CONS od oboje na vrhu steka pravi listu koja sadrži jedan element: (v_k). Zatim se na tu listu nadovezuje rezultat izvršavanja $e_{k-1}*n$ (tj. v_{k-1}), i tako redom dok se ne stigne do v_1. Dakle, nakon izvršavanja koda $\texttt{(LDC NIL)}|e_k*n|\texttt{(CONS)}|\ldots|e_1*n|\texttt{(CONS)}$ na vrhu steka imamo listu izračunatih argumenata funkcije, tj. listu oblika $(v_1\ \ v_2 \ldots v_k)$. Sada kod $e*n$ na vrh steka stavlja svoj rezultat, a to je *zatvorenje*, i sve je spremno za izvršenje instrukcije AP.

LET izraz, oblika $\texttt{(LET}\ e\ \texttt{(}x_1\texttt{.}e_1\texttt{)}\ldots\texttt{(}x_k\texttt{.}e_k\texttt{))}$, ako malo bolje pogledamo, možemo zaključiti da nije ništa drugo do *poziv funkcije*, koja ima telo e i argumente $(x_1 \ldots x_k)$, a pri pozivu su joj kao argumenti prosleđene vrednosti izraza $e_1, \ldots, e_k$. Preciznije, navedeni LET izraz možemo zapisati u obliku $\texttt{((LAMBDA (}x_1 \ldots x_k\texttt{)}\ e\texttt{)}\ e_1 \ldots e_k\texttt{)}$. Kada na ovo primenimo pravilo prevođenja za LAMBDA izraz, dobijamo da se LET izraz prevodi na sledeći način:

$\texttt{(LET}\ e\ \texttt{(}x_1\texttt{.}e_1\texttt{)}\ldots\texttt{(}x_k\texttt{.}e_k\texttt{))}*n =$
$\texttt{(LDC NIL)}|e_k*n|\texttt{(CONS)}|\ldots|e_1*n|\texttt{(CONS)}|$
$\texttt{(LDF}\ e*m|\texttt{(RTN) AP)}$, gde je $m = \texttt{((}x_1 \ldots x_k\texttt{).}n\texttt{)}$

Ovde valja napomenuti da, dok se izraz e prevodi u odnosu na listu imena n koja je proširena imenima $(x_1 \ldots x_k)$, u izrazima e_1,...,e_k koji definišu vrednosti tih promenljivih, te promenljive nisu definisane, tj. izrazi $e_1, \ldots, e_k$ se prevode u odnosu na listu imena n.

LETREC izraz se razlikuje od LET izraza po tome što su promenljive koje on uvodi vidljive i u izrazima koji te promenljive definišu. Pravilo prevođenja LETREC izraza glasi:

(LETREC e $(x_1.e_1)\ldots(x_k.e_k))*n =$
(DUM LDC NIL)$|e_k*m|$(CONS)$|\ldots|e_1*m|$(CONS)$|$
(LDF $e*m|$(RTN) RAP), gde je $m =((x_1 \ldots x_k).n)$

Vidimo da se izrazi $e_1, \ldots, e_k$ prevode u odnosu na proširenu listu imena m. Razlike u odnosu na LET izraz još uključuju stavljanje instrukcije DUM na početak generisanog koda, i korišćenje instrukcije RAP umesto AP. Pogledajmo na primeru kako funkcioniše ovako dobijen kod.

Primer 3.3 *Posmatrajmo LispKit LISP izraz*

```
(LETREC (FAC (QUOTE 6))
  (FAC LAMBDA (X) ....)
)
```

gde je funkcija FAC *definisana kao u primeru 2.3 na strani 24. Prevođenjem ovog izraza u odnosu na listu imena* NIL*, dobijamo sledeći kod:*

```
(DUM LDC NIL LDF c CONS
     LDF (LDC NIL LDC 6 CONS
         LD (0.0) AP RTN) RAP)
```

gde je c kôd tela funkcije FAC*. Ako u kontrolnu listu SECD mašine učitamo dobijeni kod, a ostale registre postavimo na* NIL*, i započnemo izvršavanje koda, dobićemo sledeći niz stanja:*

```
NIL  NIL  (DUM...RAP)      NIL
NIL  α    (LDC NIL...RAP)  NIL
```

gde je α fiktivno okruženje oblika $\alpha =$(Ω.NIL)*. Izvršavanje se nastavlja:*

```
(NIL)                          α (LDF c...RAP)              NIL
((c.α) NIL)                    α (CONS...RAP)               NIL
(((c.α)))                      α (LDF (LDC NIL...RTN) RAP)  NIL
(((LDC NIL...RTN).α)((c.α)))   α (RAP)                      NIL
```

sada se na vrhu steka nalazi zatvorenje koje predstavlja kôd LETREC-ovog „glavnog“ izraza (FAC (QUOTE 6))*. Ispod njega je lista vrednosti definisanih u LETREC izrazu, u ovom slučaju to je samo zatvorenje funkcije* FAC*. Oba zatvorenja sadrže fiktivno okruženje α. Sada, kada se izvrši instrukcija RAP, glava fiktivnog okruženja α se zamenjuje pomenutom listom definisanih vrednosti i SECD mašina prelazi u stanje*

```
NIL  α  (LDC NIL...RTN)  (NIL NIL NIL.NIL)
```

gde je, zahvaljujući sporednom efektu pseudofunkcije rplaca (vidi str. 32), α *sada oblika*

$$\alpha = (((c.\alpha)).\texttt{NIL})$$

to jest, α *je okruženje koje sadrži samo jednu podlistu i u njoj samo jedan element –* $(c.\alpha)$*. To znači da će prilikom izvršavanja koda dobijenog prevođenjem LETREC-ovog „glavnog“ izraza instrukcija* `LD (0.0)` *učitati na stek zatvorenje funkcije* `FAC`*, a pri rekurzivnim pozivima funkcije* `FAC` *tom zatvorenju će se pristupati sa pozicije* `(1.0)` *u okruženju, zato što tada lista vrednosti argumenata funkcije* `FAC` *zauzima mesto prve podliste u okruženju.*

Glava 4

Implementacija LispKit LISP-a

U ovoj glavi predstavićemo jednu moguću implementaciju LispKit LISP-a u programskom jeziku Java. Pri tom ćemo u velikoj meri koristiti rezultate prikazane u [1, 7, 8]. Prvi odeljak posvetićemo opisu tipa podataka u Javi, kojim će biti predstavljen s-izraz, a zatim učitavanju s-izraza iz ulaznog tekstualnog fajla. U drugom odeljku daćemo opis implementacije provere sintaksne ispravnosti LispKit LISP programa sadržanog u učitanom s-izrazu. Sledeći odeljak opisaće implementaciju kompajlera iz programskog jezika LispKit LISP u kod SECD mašine, a zatim sledi odeljak u kojem će biti prikazan jedan interpreter SECD mašine. Na kraju, u poslednjem odeljku, biće opisane konkretne Java aplikacije kojima se pokreće prevođenje i izvršavanje programa, i biće dat primer njihovog korišćenja.

Sve Java klase koje čine implementaciju organizovaćemo u paket `lispkit`.

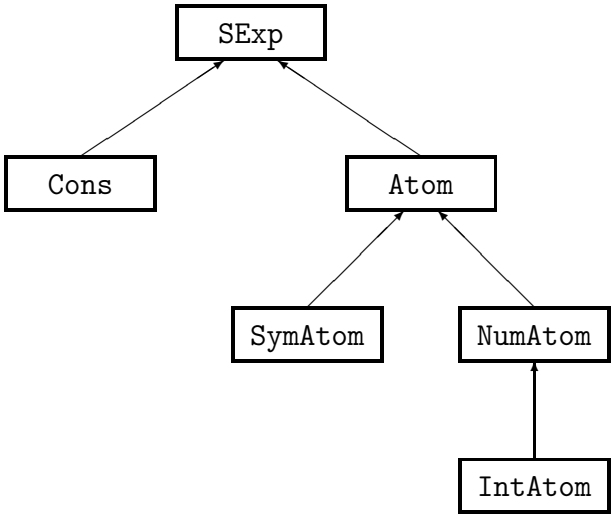
4.1 S-izraz

4.1.1 Tip podataka `SExp`

Tip podataka koji će u Javi predstavljati s-izraz, tj. strukturu podataka i operacije nad njim, implementiraćemo sledećim klasama:

`SExp`	predstavlja opšti s-izraz
`Cons`	implementira tačkasti izraz
`Atom`	prestavlja sve vrste atoma
`SymAtom`	implementira simbolički atom
`NumAtom`	predstavlja numerički atom
`IntAtom`	implementira celobrojni atom, kao specijalan slučaj numeričkog atoma

koje ćemo rasporediti u hijerarhiju prikazanu na slici:



Ovo znači da je klasa `SExp` koren hijerarhije, njene podklase, odnosno naslednici su klase `Cons` i `Atom`, i tako dalje. Primetimo da ova hijerarhija prirodno oslikava opis s-izraza koji smo dali u odeljku 2.1.

Sve klase iz ove hijerarhije, radi preglednosti, smestićemo u podpaket `sexp` paketa `lispkit`. Za signalizaciju grešaka prilikom izvođenja operacija nad s-izrazima, u ovaj paket smestićemo još jednu klasu, koja će predstavljati *korisnički izuzetak*[1], definisan na standardan način:

```
public class SExpException extends Exception {
  public SExpException() {}
  public SExpException(String msg) {
    super(msg);
  }
}
```

Prokomentarišimo sada svaku od klasa iz hijerarhije ponaosob.

• **Klasa** `SExp` je, u stvari, apstraktna klasa. To znači da se u Java programima neće pojavljivati njene „čiste“ instance, tj. svi objekti tipa `SExp` će ujedno biti i instance nekog od njenih naslednika. Međutim, klasu `SExp` nećemo implementirati koristeći Javin koncept `abstract` klase, niti pomoću interfejsa. Implementiraćemo je kao običnu klasu

```
public class SExp {
  ...
}
```

a unutar nje ćemo definisati sve javne metode koji se pojavljuju u njenim naslednicima. Na primer, metod

[1] Engl. *user-defined exception*.

```
public int intValue() { ... }
```

definisan je u klasi `IntAtom` i vraća vrednost celobrojnog atoma u kao Javin elementarni tip `int`. Ovaj metod, naravno, nije definisan ni u jednoj drugoj klasi iz hijerarhije, jer nije ni primeren ostalim vrstama s-izraza. Međutim, ipak ćemo ga definisati u klasi `SExp`:

```
public int intValue() {
  throw new SExpException("intValue() -- IntAtom expected.");
}
```

što znači da, ako kojim slučajem metod `intValue` definisan u klasi `SExp` bude pozvan, biće signalizirana greška. Napomenimo da su kvalifikatori `public` potrebni da bi se klasi `SExp` i njenim članovima pristupalo izvan paketa `lispkit.sexp`, tj. iz paketa `lispkit`. Ako, recimo, u Java programu imamo sledeće deklaracije:

```
SExp i = new IntAtom(9);
SExp s = new SymAtom("SSSS");
```

kojima smo promenljivoj `i` dodelili celobrojni atom `9`, a promenljivoj `s` simbolički atom `SSSS`, tada izraz

```
i.intValue();
```

označava ceo broj 9, jer je pozvan metod `intValue` definisan u klasi `IntAtom`, a izraz

```
s.intValue();
```

dovodi do izuzetka, jer, pošto metod `intValue` nije definisan u klasi `SymAtom`, biće pozvan `intValue` definisan u `SExp`. Takođe, da u klasi `SExp` nisu definisani metodi iz njenih podklasa, ovakvi izrazi ne bi ni bili mogući, i Java kompajler prijavljivao bi grešku kako metod `intValue` nije definisan u klasi `SExp`.

• **Klasa** `Atom`, direktan naslednik klase `SExp`, takođe je apstraktna klasa. S obzirom da `SExp` implementira sve potrebne metode, klasa `Atom` je prilično jednostavna:

```
public class Atom extends SExp {
}
```

• **Klasa** `SymAtom` jedna je od konkretizacija klase `Atom`:

```
public class SymAtom extends Atom {
  ...
}
```

Niz znakova koji predstavljaju simbolički atom implementiraćemo skrivenom promenljivom tipa `String`:

```
private String value;
```

koja će moći biti inicijalizovana konstruktorom

```
public SymAtom(String value) {
  this.value = new String(value);
}
```

a kojoj će se pristupati korišćenjem metoda

```
public String stringValue() { return this.value; }
```

Implementiraćemo i funkciju *eq*, koja proverava jednakost atoma, pomoću metoda

```
public boolean eq(String s) {
  return this.value.equals(s);
}
public boolean eq(int i) {
  return false;
}
public boolean eq(SExp se) {
  if (se instanceof SymAtom)
    return this.value.equals(((SymAtom)se).stringValue());
  else
    return false;
}
```

gde se kao zamena za simbolički i celobrojni atom kao argumenti mogu pojaviti Javini tipovi `String` i `int`. Poređenje simboličkih atoma vrši se, jednostavno, putem poređenja `String`-ova korišćenjem metode `equals` definisane u klasi `String`.

Metod `toString`, koji vraća `String` reprezentaciju objekta, ima specijalan tretman u Javi – ako se neki objekat u programu pojavi na mestu gde se očekuje `String`, automatski se poziva metod `toString` tog objekta i dobijeni `String` prosleđuje tamo gde se očekuje. Na primer, definisanjem metoda `toString` u klasi `SymAtom`, i uopšte u celoj `SExp` hijerarhiji, naredba

```
System.out.println(se);
```

gde je `se` objekat tipa `SExp`, ispisaće `String` reprezentaciju objekta `se` na ekranu. U slučaju simboličkog atoma metod `toString` krajnje je jednostavan:

```
public String toString() {
  return this.value;
}
```

Simbolički atomi `NIL`, `T` i `F`, zbog svog specijalnog značenja u programskom jeziku LispKit LISP, zaslužuju da ih prikažemo posebnim konstantama:

```
public static final SExp NIL = new SymAtom("NIL");
public static final SExp T   = new SymAtom("T");
public static final SExp F   = new SymAtom("F");
```

koje ćemo kasnije intenzivno koristiti.

• **Klasa** `NumAtom` veoma je slična klasi `Atom`:

```
public class NumAtom extends Atom {
}
```

• **Klasa** `IntAtom` jedina je konkretizacija klase `NumAtom` koju ćemo implementirati:

```
public class IntAtom extends NumAtom {
  ...
}
```

Vrednost celobrojnog atoma implementiraćemo skrivenom promenljivom tipa `int`:

```
private int value;
```

za čiju će inicijalizaciju biti zaduženi konstruktori

```
public IntAtom(int value) {
  this.value = value;
}
public IntAtom(String s) throws NumberFormatException {
  this.value = Integer.parseInt(s);
}
```

a kojoj će se pristupati pomoću metoda

```
public int intValue() { return value; }
```

Funkciju *eq* još je jednostavnije implementirati nego u klasi `SymAtom`:

```
public boolean eq(String s) {
  return false;
}
public boolean eq(int i) {
```

```
  return this.value == i;
}
public boolean eq(SExp se) throws SExpException {
  if (se instanceof IntAtom)
    return this.value == se.intValue();
  else
    return false;
}
```

`String` reprezentacija celobrojnog atoma takođe se jednostavno implementira:

```
public String toString() {
  return Integer.toString(value);
}
```

- **Klasa** `Cons` je direktan naslednik klase `SExp`:

```
public class Cons extends SExp {
  ...
}
```

Glava i rep tačkastog izraza jesu s-izrazi:

```
private SExp carSExp;
private SExp cdrSExp;
```

Inicijalizaciju tačkastog izraza obezbedićemo pomoću nekoliko prigodnih konstruktora:

```
public Cons(SExp carSExp, SExp cdrSExp) {
  this.carSExp = carSExp;
  this.cdrSExp = cdrSExp;
}
public Cons(String s, SExp cdrSExp) {
  this.carSExp = new SymAtom(s);
  this.cdrSExp = cdrSExp;
}
public Cons(int i, SExp cdrSExp) {
  this.carSExp = new IntAtom(i);
  this.cdrSExp = cdrSExp;
}
public Cons(int i, int j) {
  this.carSExp = new IntAtom(i);
  this.cdrSExp = new IntAtom(j);
}
```

koji će nam omogućiti da tačkasti izraz inicijalizujemo koristeći i Javine tipove `String` i `int`, umesto simboličkih i celobrojnih atoma. Za pristup komponentama tačkastog izraza koristićemo se metodima

```
public SExp car() { return carSExp; }
public SExp cdr() { return cdrSExp; }
```

kao i njhovim kombinacijama, na primer

```
public SExp caar() throws SExpException {
  return this.car().car();
}
public SExp cadr() throws SExpException {
  return this.cdr().car();
}
```

koje će služiti za skraćivanje zapisa izraza u Java programima što koriste ovu klasu. Moramo napomenuti da je na izbor imena „kombinovanih“ metoda uticala tradicija – da su `car` i `cdr` definisane kao *funkcije jedne promenljive*, a ne kao *metodi*, tada bi *funkcija* `cadr` bila definisana kao

```
SExp cadr(SExp se) { return car(cdr(se)); }
```

gde redosled slova `a` i `d` u imenu funkcije `cadr` označava redosled navođenja poziva *funkcija* `car` i `cdr` u njenom telu. Kod *metoda* je taj redosled obrnut, ali mi ćemo ipak zadržati tradicionalni način imenovanja, kao da su u pitanju funkcije.

Funkcija, tačnije, metod `eq` za tačkaste izraze uvek ima vrednost `false`:

```
public boolean eq(SExp se)  { return false; }
public boolean eq(String s) { return false; }
public boolean eq(int i)    { return false; }
```

gde, opet, `String` i `int` mogu da zamene simbolički, odnosno celobrojni atom.

Ovde ćemo implementirati i funkciju *rplaca*, koju smo opisali na strani 32, pomoću jednostavnog metoda

```
public SExp rplaca(SExp carSExp) {
  this.carSExp = carSExp;
  return this;
}
```

Na kraju, reprezentacija tačkastog izraza u vidu `String`-a nešto je složenija nego što je to slučaj kod atoma:

```
public String toString() {
  String s = this.car().toString();
  SExp tmpSExp = this.cdr();

  while (tmpSExp instanceof Cons) {
    s = s + " " + ((Cons)tmpSExp).car().toString();
    tmpSExp = ((Cons)tmpSExp).cdr();
  }
  if (tmpSExp != SymAtom.NIL)
    s = s + " . " + tmpSExp.toString();
  return "(" + s + ")";
}
```

pri čemu se dobija reprezentacija izraza sa najmanjim brojem tačaka i zagrada, u skladu sa dogovorom o njihovom brisanju. Algoritam je sledeći. Ukoliko je rep tačkastog izraza tačkasti izraz, na `String` reprezentaciju glave tačkastog izraza nadovezuje se, odvojena razmakom, reprezentacija glave od repa tačkastog izraza, i tako redom sve dok je rep tačkasti izraz. Ako je rep simbolički atom `NIL`, znači da je u pitanju lista, inače je potrebno navesti tačku i reprezentaciju tog atoma. Naravno, reprezentacija celog tačkastog izraza uokvirena je parom zagrada.

4.1.2 Učitavanje s-izraza

Čitanje s-izraza iz ulaznog tekstualnog fajla i, da tako kažemo, punjenje strukture podataka koja će ga sadržati implementiraćemo korišćenjem kompajler generatora *Coco/R for Java*. Ovaj program na osnovu ulaznog tekstualnog fajla koji sadrži opis gramatike ciljnog programskog jezika upotpunjen *semantičkim akcijama*, tj. Java naredbama koje programer umeće sam, generiše Java klase koje implementiraju sintaksnu analizu programa pisanih u ciljnom jeziku. Sintaksna analiza se realizuje metodom *rekurzivnog spusta*. Postoje još dva ulazna fajla – *Scanner.frame* i *Parser.frame*, koji predstavljaju „šablone“ u koje se „umeće“ generisani kod, da bi se dobile izlazne klase, od kojih su bitne `Scanner`, `Parser` i `ErrorStream`. Klasa `Scanner` implementira *leksičku analizu*, odnosno prepoznavanje i izdvajanje osnovnih elemenata jezika, tzv. *tokena* – npr. ključnih reči, imena, specijalnih znakova itd. Klasa `Parser`, ruku pod ruku sa `Scanner`-om implementira samu sintaksnu analizu, dok klasa `ErrorStream` služi za obradu grešaka.

S-izraz, kakav smo opisali u odeljku 2.1, nije baš pravi „programski jezik“, ali bez obzira na to vrlo je pogodan da se njegova sintaksna analiza implementira uz pomoć Coco/R-a na kratak i elegantan način. Opišimo strukturu jednog mogućeg (kratkog) Coco/R ulaznog fajla sa tom svrhom – *lispkit.atg*. Nejgov sadržaj imaće puno sličnosti sa gramatikom s-izraza koju smo dali na

strani 17.

Posle obavezne početne direktive

```
COMPILER lispkit
```

koja deklariše ime kompajlera, a ovo mora biti identično imenu paketa u kojem se sve odvija, deklarisaćemo promenljivu

```
private static SExp se;
```

u koju će biti smešten pročitani s-izraz, i definisati proceduru

```
static SExp getSExp() { return se; }
```

kojom će se toj promenljivoj pristupati.

Neće se praviti razlika između velikih i malih slova, što se obeležava direktivom

```
IGNORE CASE
```

Ovo znači da će prilikom sintaksne analize mala slova na koja se naiđe biti konvertovana u velika.

Od dozvoljenih znakova, *slova* i *cifre* implementiraju se na standardan način:

```
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  digit  = "0123456789".
```

Omogućićemo i pisanje komentara u ulaznim s-izrazima, između znakova „/*“ i „*/“, sa mogućnošću ugneždenja:

```
COMMENTS
  FROM "/*" TO "*/" NESTED
```

Simboličke i celobrojne atome implementiraćemo kao *tokene*:

```
TOKENS
  symbolic = letter { letter | digit }.
  integer  = ["-"] digit { digit }.
```

Pre nego što pređemo na implementaciju ostatka gramatike s-izraza, važno je reći da gramatika koju Coco/R prihvata mora da zadovoljava LL(1) ograničenje. To znači da se na osnovu sledećeg tokena sa ulaza mora u svakom trenutku moći odrediti koja od alternativa u gramatici razdvojenih znakom „|“ opisuje izraz pisan na ciljnom jeziku. Gramatika s-izraza koju smo dali ne zadovoljava ovo ograničenje jer u produkciji

```
<SExpressionList>  ::=  <SExpression> |
                        <SExpression> '.' <SExpression> |
                        <SExpression> <SExpressionList>
```

imamo tri alternative koje započinju na isti način. Srećom, ovu produkciju možemo izmeniti tako da gramatika zadovoljava LL(1) ograničenje, a da se ništa od njenog značenja ne gubi:

```
<SExpressionList>  ::=  <SExpression>
                        [ '.' <SExpression>
                        | <SExpressionList> ]
```

Sada možemo da pređemo na produkcije u *lispkit.atg*:

```
PRODUCTIONS
  lispkit = SExpression<^se> .
```

U ovoj produkciji leva strana je ista kao ime koje smo naveli uz direktivu `COMPILER`. Jedna takva produkcija je obavezna u listi produkcija, a ova naša je jednostavna – `lispkit` se izjednačava sa metapromenljivom `SExpression`, kojoj kao parametar prosleđujemo promenljivu `se` deklarisanu na početku. Znak „^“ ispred imena promenljive označava da će njen sadržaj biti izmenjen u toku sintaksne analize, odnosno da je u pitanju „poziv po imenu“. Ovo prosleđivanje parametara ima smisla jer za svaku metapromenljivu sa leve strane, iz liste produkcija, Coco/R generiše istoimenu Java proceduru koja prima parametre navedene uz metapromenljivu i čije se telo generiše na osnovu desne strane produkcije i umetnutih semantičkih akcija, po poznatim pravilima rekurzivnog spusta. Sledeća produkcija je

```
SExpression<^SExp se>
=                               (. se = null; .)
( Atom<^se>
| "(" SExpressionList<^se> ")"
) .
```

koja ima jedan promenljivi parametar tipa `SExp`, što ga semantička akcija na početku, navedena između tačkastih zagrada, tj. znakova „`(.`“ i „`.)`“, inicijalizuje na `null`. Kasnije se taj parametar prosleđuje dalje metapromenljivama `Atom` ili `SExpExpressionList`.

```
SExpressionList<^SExp se>       (. SExp cdrSExp; .)
=
  SExpression<^se>              (. if (LookAheadName().equals(")"))
                                       se = new Cons(se, SymAtom.NIL);
                                 .)
[ "." SExpression<^cdrSExp>     (. se = new Cons(se, cdrSExp); .)
| SExpressionList<^cdrSExp>     (. se = new Cons(se, cdrSExp); .)
] .
```

Ovde se na osnovu sledećeg tokena, u kojeg možemo da „zavirimo“ pomoću funkcije `LookAheadName`, zaključuje da li smo pri analizi stigli do kraja *liste*. Produkcija `Atom` takođe je jednostavna:

```
Atom<^SExp se>
=                          (. se = null; .)
( "NIL"                    (. se = SymAtom.NIL; .)
| "T"                      (. se = SymAtom.T; .)
| "F"                      (. se = SymAtom.F; .)
| symbolic                 (. se = new SymAtom(LexName()); .)
| integer                  (. try { se = new IntAtom(LexName()); }
                             catch (Exception ex) {
                               SemError(1);
                             } .)
) .
```

gde funkcijom `LexName` pristupamo tek pročitanom tokenu, u formi `String`-a. Ako pri konverziji celobrojnog tokena iz `String`-a u `IntAtom` dođe do izuzetka, a to će se desiti ako se premaše granice tipa `int`, procedurom `SemError` biće prijavljena greška.

I na kraju, Coco/R ulazni fajl završavamo direktivom

```
END lispkit.
```

Po preporukama autora Coco/R-a, da bi se pri sintaksnoj analizi obrađivale još neke greške, a ne samo one koje pronalazi generisani sintaksni analizator, klasu `ErrorStream` treba proširiti tako što se definiše nova klasa koja je nasleđuje, i sva se proširenja implementiraju u toj novoj klasi. Tako ćemo i postupiti. Definisaćemo klasu

```
class LispErrorStream extends ErrorStream {
  ...
}
```

i u njoj proceduru `SemError`, koja je identična istoimenoj proceduri iz klase `ErrorStream`, s tim što uvodi nove vrste grešaka:

```
void SemErr(int n, int line, int col) {
  String s;
  count++;
  switch (n) {
    case -1: {s = "invalid character"; break;}
    case  1: {s = "invalid integer"; break;}
    default: {s = "Semantic error " + n; break;}
  }
  StoreError(n, line, col, s);
}
```

Procedura `StoreError` ispisuje poruku o grešci, i informaciju o redu i koloni ulaznog fajla u kojem je greška, pri sintaksnoj analizi, pronađena.

Na kraju, opišimo našin korišćenja novostvorenih klasa `Scanner`, `Parser` i `ErrorStream`, odnosno `LispErrorStream`. U klasi `Scanner`, procedura koja služi za pripremu, odnosno inicijalizaciju sintaksnog analizatora je

```
static void Init(String, ErrorStream)
```

koja prima kao argumente ime fajla što sadrži program koji treba analizirati, i instancu klase `ErrorStream` koja će služiti za obradu grešaka. Naravno, na mestu `ErrorStream`-a može se koristiti i `LispErrorStream`. Nakon inicijalizacije, procedurom klase `Parser`:

```
static void Parse()
```

pokreće se sama sintaksna analiza. U klasi `Parser` postoji još jedna korisna procedura:

```
static boolean Successful()
```

kojom se proverava da li je analiza završena uspešno, tj. bez pronađenih grešaka, ili ne. Procedurom klase `ErrorStream`, odnosno `LispErrorStream`:

```
void Summarize(String)
```

ispisuje se prikaz uspešnosti analize.

Iskoristimo priliku da se zapitamo: LispKit LISP programi mogu da sadrže mnogo simboličkih atoma, odnosno imena elementarnih funkcija, izraza, i promenljivih, koja se ponavljaju. Da li to znači da je naša implementacija s-izraza neefikasna u smislu uštede prostora, zato što se odgovarajući `String`-ovi u strukturi podataka koja implementira simbolički atom ponavljaju?

Odgovor je negativan, a daje nam ga sama Java, koja interno vodi računa da nikad ne dođe do dupliranja `String`-ova u memoriji u toku izvršavanja programa. Sve su instance klase `SymAtom` (sem onih koje predstavljaju simboličke atome `NIL`, `T` i `F`) u toku izvršavanja programa različite, čak i one čiji su `String`-ovi isti. No, ovo ne predstavlja veliki nedostatak, s obzirom da same instance, pošto se `String`-ovi ne dupliraju, zauzimaju vrlo malo memorije.

Ovo je jedno od pojednostavljenja implementacije LispKit LISP-a koje nam omogućuje programski jezik Java, jer bismo inače bili primorani sami da implemntiramo sprečavanje dupliranja `String`-ova u memoriji. Ako to ne bismo uradili, naša implementacija bi bila manje efikasna.

4.2 Provera sintaksne ispravnosti

Ako pretpostavimo da učitani s-izraz sadrži *ispravan* LispKit LISP program, verovatno smo pogrešili – po pravilu kompajleri provode veći deo vremena obrađujući programe u kojima je programer napravio grešku, nego one u kojima nije. Bez obzira da li te greške spadaju u banalne sintaksne greške, ili u teško otklonjive suštinkse greške, potrebno ih je u što većem broju otkriti, i to u što je moguće ranijoj fazi programiranja.

Iz tog razloga ćemo, pre nego što se upustimo u prevođenje učitanog LispKit LISP programa u kod SECD mašine, učiniti izvesne napore da proverimo njegovu korektnost, tj. da li zadovoljava sintaksu i sematiku LispKit LISP-a, koju smo opisali u odeljku 2.2. U stvari, implementiraćemo samo proveru *sintaksne* ispravnosti učitanog programa. Pod tim konkretno mislimo na proveru da li program zadovoljava sintaksu datu gramatikom na str. 23. Provera *semantičke* ispravnosti prilično je složeno pitanje koje bi moglo da bude oblast posebnog istraživanja. Na primer, provera semantičke ispravnosti izraza

```
(ADD (FUN X) (QUOTE 7))
```

tj. da li su vrednosti argumenata poziva elementarne funkcije ADD numerički atomi, morala bi u obzir da uzme okruženje u kojem se taj izraz nalazi. To znači da bi trebalo analizirati šta simbolički atom `X` može da predstavlja, i kakve vrednosti funkcija `FUN` može da vrati, a da nam sve to ne mora garantovati da neće doći do semantičke greške. Provera da li je vrednost izraza `(QUOTE 7)` numerički atom mnogo je jednostavnija, ali mi se ni u takve provere nećemo upuštati, s obzirom da se banalne greške navođenja konstante pogrešnog tipa najređe i prave, a lako se i otkrivaju. Najveća pretnja programerima su suptilne semantičke greške skrivene u više slojeva poziva funkcija i izračunavanja izraza. Sve semantičke greške, bilo banalne ili suptilne, u imlementaciji LispKit LISP-a koju opisujemo biće ispoljene tek u toku izvršavanja programa.

Sintaksnu proveru implementiraćemo pomoću procedura koje direktno „prate“ gramatiku datu na str. 23, to jest metodom *rekurzivnog spusta*, s tim što se obrada neće vršiti nad nizom znakova, kao što je bio slučaj kod sintaksne analize s-izraza, nego nad *s-izrazom*, odnosno njegovom internom reprezentacijom u formi klase `SExp` i njenih naslednika. Ovakav pristup je mnogo jednostavniji od pokušaja da sintaksnu proveru obavimo prilikom učitavanja znakova iz ulaznog fajla jer u tom slučaju

- pogrešna struktura samog tekstualnog s-izraza čini sintaksnu proveru jako otežanom, ako ne i besmislenom, *i*
- dogovor o brisanju zagrada umnogome komplikuje analizu, jer postoje mnogobrojni ekvivalentni zapisi programa.

Za početak, proširićemo klasu `LispErrorStream` dvema novim procedurama, koje će nam služiti za obradu sintaksnih grešaka u LispKit LISP programima. Jedna od njih je

```
void StoreLispError(int n, SExp e, String s) {
  System.out.println("-- lisp syntax error: " + s + ": " + e);
}
```

napisana po ugledu na `StoreError` iz originalnog `ErrorStream`-a, a koja jednostavno ispisuje opis greške, sadržan u parametru `s`, i izraz `e` u kojem je ta greška pronađena. Sledeća, u stvari bitna, procedura je

```
void LispSynErr(int n, SExp e) {
  String s;
  count++;
  switch (n) {
    case  0: {s = "unknown error"; break;}
    case  1: {s = "symbolic atom expected"; break;}
      .........
    case 26: {s = "multiple occurences of symbol"; break;}
    default: {s = "Syntax error " + n; break;}
  }
  StoreLispError(n, e, s);
}
```

analogna proceduri `SemErr` iz `ErrorSream`-a, opisanoj u prethodnom odeljku, s tom razlikom da kao argument prima i grešni izraz `e`, i koristi proceduru `StoreLispError` za ispis poruka o greškama.

Klasom

```
class LispSyntaxChecker {
  .....
}
```

implementiraćemo samu proveru sintaksne ispravnosti. Unutar nje, biće nam portebna promenljiva

```
private static LispErrorStream err;
```

kao i procedura

```
private static void LispSynError(int n, SExp e) {
  err.LispSynErr(n, e);
}
```

koju će procedure što implementiraju sintaksnu proveru direktno pozivati da bi signalizirale grešku.

U vidljive članove klase `LispSyntaxChecker` spadaju samo tri procedure:

```
static void syntaxCheck(SExp e) {
  ...
}
```

kojom se pokreće sintaksna provera izraza `e`, a koju ćemo kasnije definisati u celosti, zatim

```
static void Init(LispErrorStream err) {
  LispSyntaxChecker.err = err;
}
```

koju je potrebno pozvati pre `syntaxCheck` da bi se inicijalizovala promenljiva `err`, i

```
static boolean Successful() {
  return err.count == 0;
}
```

kojom se posle izvršene provere ispituje da li je bilo grešaka ili ne.

Pređimo sada na implementaciju same sintaksne provere. Provera da li je s-izraz simbolički atom vrši se jednostavno:

```
private static void symbolic(SExp e) {
  if (!(e instanceof SymAtom))
    LispSynError(1, e); // symbolic atom expected
}
```

Dakle, ako `e` nije tipa `SymAtom`, signalizira se greška. Kao komentar pored poziva `LispSynError` naveli smo poruku koja bi bila prikazana u slučaju greške. Provera da li je izraz lista simboličkih atoma takođe je jednostavna:

```
private static void symbolicList(SExp e) {
  SExp d;
  try {
    d = e;
    while (!d.eq(SymAtom.NIL)) {
      symbolic(d.car());
      d = d.cdr();
    }
  }
  catch (SExpException ex) {
    LispSynError(2, e); // invalid symbolic atom list
  }
}
```

gde se `while` petljom prolazi kroz s-izraz `e` kao kroz listu i pomoću procedure `symbolic` proverava da li je svaki element simbolički atom. Ovde je

ilustrovana zgodna primena mehanizma izuzetaka, koja će se ponoviti i u svim ostalim procedurama koje implementiraju rekurzivni spust. Naime, u slučaju da je u `try` bloku došlo do izuzetka `SExpException`, odnosno ako je prilikom prolaska kroz `e` metod `car()` ili `cdr()` primenjen na pogrešnu vrstu s-izraza, biće signalizirana greška, jer to znači da s-izraz `e` uopšte nema očekivanu strukturu liste.

Proveru da li je s-izraz validan LispKit LISP izraz vrši sledeća procedura:

```
private static void exp(SExp e) {
  try {
    if (e instanceof Atom)
      symbolic(e);
    else if (e.car().eq("QUOTE"))
      quoteExp(e);
    else if (e.car().eq("EQ"))
      eqExp(e);
        .........
    else if (e.car().eq("LETREC"))
      letrecExp(e);
    else
      callExp(e);
  }
  catch (SExpException ex) {
    LispSynError(3, e);
  }
}
```

gde smo skratili nabrajanje svih vrsta izraza. Algoritam je jednostavan – ako je `e` atom proveravamo da li je simbolički, inače na osnovu glave od `e` utvrđujemo koja bi vrsta izraza mogla biti u pitanju. Opet, mehanizmom izuzetaka štitimo se od nepredviđenosti u strukturi s-izraza `e`.

Za proveru sintaksne ispravnosti **QUOTE** izraza zadužena je procedura

```
private static void quoteExp(SExp e) {
  ......
    if (e.car().eq("QUOTE")) {
      if (!e.cddr().eq(SymAtom.NIL))
        LispSynError(4, e); // invalid QUOTE expression
    }
    else
      LispSynError(4, e);
  ......
}
```

koja *neće* signalizirati grešku ukoliko je `e` lista od dva elementa od kojih je prvi simbolički atom `QUOTE`. Ovde smo izostavili `try` ... `catch` blok, pot-

puno analogan kao u prethodno navedenim procedurama. Isti je slučaj i u procedurama koje slede, pa ni u njima taj blok nećemo navoditi.

Ilustrujmo još proveru sintaksne ispravnosti **EQ** izraza:

```
private static void eqExp(SExp e) {
  ......
    if (e.car().eq("EQ")) {
      exp(e.cadr());
      exp(e.caddr());
      if (!e.cdddr().eq(SymAtom.NIL))
        LispSynError(5, e); // invalid EQ expression
    }
    else
      LispSynError(5, e);
  ......
}
```

Ova procedura zahteva od s-izraza `e` da je lista čiji je prvi element (`e.car()`) simbolički atom `EQ`, drugi i treći (`e.cadr()` i `e.caddr()`) izrazi, i da se tu lista završava.

Provera sintaksne ispravnosti **ADD**, **SUB**, **MUL**, **DIV**, **REM**, **LEQ** i **CONS** izraza analogna je EQ izrazu. Što se tiče **CAR**, **CDR** i **ATOM** izraza, kod provere njihove ispravnosti potrebno je proveravati jedan argument manje. Ni **IF** izraz se ne razlikuje puno – potrebno je voditi računa o njegova *tri* argumenta koji treba da su izrazi.

Ovim smo stigli do sledeće suštinski različite vrste izraza – **LAMBDA** izraza. Proveru njegove sintaksne ispravnosti vrši sledeća procedura:

```
private static void lambdaExp(SExp e) {
  ......
    if (e.car().eq("LAMBDA")) {
      symbolicList(e.cadr());
      exp(e.caddr());
      if (!e.cdddr().eq(SymAtom.NIL))
        LispSynError(17, e); // invalid LAMBDA expression
    }
    else
      LispSynError(17, e);
  ......
}
```

Prvi element tročlane liste `e` treba da je simbolički atom `LAMBDA`, drugi lista simboličkih atoma, a treći izraz.

LET izraz se proverava na sledeći način:

```
private static void letExp(SExp e) {
  SExp d;
  ......
    if (e.car().eq("LET")) {
      exp(e.cadr());
      d = e.cddr();
      while (!d.eq(SymAtom.NIL)) {
        symbolic(d.caar());
        exp(d.cdar());
        d = d.cdr();
      }
    }
    else
      LispSynError(18, e); // invalid LET expression
  ......
}
```

gde nakon što se utvrdi da je prvi element liste `e` atom `LET`, i drugi da je izraz, `while` petljom se prolazi kroz listu do njenog kraja i utvrđuje da li su preostali njeni elementi tačkasti izrazi čije su glave simbolički atomi, a repovi izrazi. Analogno se realizuje provera ispravnosti **LETREC** izraza, s napomenom da repovi tačkastih izraza moraju biti LAMBDA izrazi.

Poziv funkcije se može proveriti ovako:

```
private static void callExp(SExp e) {
  SExp d;
  ......
    function(e.car());
    d = e.cdr();
    while (!d.eq(SymAtom.NIL)) {
      exp(d.car());
      d = d.cdr();
    }
  ......
}
```

gde je procedura `function` analogna proceduri `exp`, samo je ograničena na simbolički atom, LAMBDA, LET i LETREC izraze. Argumenti poziva funkcije mogu biti bilo koji izrazi.

Ovim smo završili opis sintaksne provere svih vrsta LispKit LISP izraza. Ostaje nam još da proverimo LispKit LISP **program**, i mada je odgovarajuća procedura slična proceduri `function`, navešćemo je radi kompletnosti:

```
private static void program(SExp e) {
  ......
    if (e.car().eq("LETREC"))
```

```
      letrecExp(e);
    else if (e.car().eq("LET"))
      letExp(e);
    else if (e.car().eq("LAMBDA"))
      lambdaExp(e);
    else
      LispSynError(22, e); // LETREC, LET or LAMBDA ... expected
  ......
}
```

Ostalo nam je još da dovršimo definiciju funkcije `syntaxCheck`:

```
static void syntaxCheck(SExp e) {
  program(e);
}
```

4.3 Kompajler

Prevodilac LispKit LISP programa u kod SECD mašine implementiraćemo pomoću klase `Compiler`. Slično kao kod provere sintaksne ispravnosti, za obradu grešaka koristićemo se klasom `LispErrorStream`, pa će nam unutar klase `Compiler` biti potrebni promenljiva `err` i procedura `LispSynError` identični onima iz klase `LispSyntaxChecker`. Greške koje se mogu pojaviti u toku prevođenja odnose se opet na sintaksu ulaznog programa, i uključuju pojavu nedefinisanog imena, kao i dupliranje imena u LAMBDA, LET i LETREC izrazima. Razlog za proveru ovih vrsta grešaka tek u fazi kompajliranja leži u jednostavnosti – u klasi `LispSyntaxChecker` bile bi potrebne velike izmene da bi se implementiralo pronalaženje ovih grešaka, dok se u klasi `Compiler`, zahvaljujući njenoj strukturi, te provere implementiraju lako i prirodno.

U dobijenom kodu instrukcije nećemo označavati simbolima, kao u glavi 3, nego numeričkim atomima, odnosno njihovim rednim brojevima, datim u tabeli na strani 27. Tako, na primer, fragment koda

```
(LDC NIL LD (0.0) CONS RTN)
```

postaje

```
(2 NIL 1 (0.0) 13 5)
```

Kao i klasa `LispSyntaxChecker`, klasa `Compiler` imaće samo tri vidljiva člana: procedure `Init` i `Successful`, koje su identične istoimenim procedurama iz klase `LispSyntaxChecker`, i funkciju

```
static SExp compile(SExp e) {
  ...
}
```

koja za LispKit LISP program `e` vraća kd̂ SECD mašine dobijen njegovim prevođenjem. U slučaju pronalaženja jedne ili više sintaksnih grešaka vraćeni program neće biti ispravan, pa je važno koristiti funkciju `Successful` za proveru da li je kompilacija uspešno obavljena.

Kao što smo videli u odeljku 3.1, prevođenje LispKit LISP izraza vrši se u odnosu na listu imena, odnosno u odnosu na okruženje u kojem se taj izraz nalazi. Zato ćemo za prevođenje proizvoljnog izraza `e` u odnosu na listu imena `n` koristiti funkciju

```
private static SExp comp(SExp e, SExp n, SExp c) {
  ...
}
```

gde će nam `c` služiti kao *akumulirajući parametar*, u kojem će biti smešten već prevedeni kôd, na koji će se kôd dobijen prevođenjem izraza `e` nadovezati. Preciznije, rezultat funkcije `comp` biće kôd oblika `e`$*$`n`|`c`.

Prilikom prevođenja nekih vrsta izraza (LET, LETREC, poziv funkcije) mogu se pojaviti čitave liste izraza koje treba prevesti. U tu svrhu definisaćemo funkciju

```
private static SExp complis(SExp e, SExp n, SExp c) {
  ...
}
```

koja za listu izraza `e` oblika `e`=$(e_1 \ldots e_k)$ vraća kod

$$\texttt{(LDC NIL)}|e_k{*}\texttt{n}|\texttt{(CONS)}|\ldots|e_1{*}n|\texttt{(CONS)}|\texttt{c}$$

Funkciju `complis` možemo implementirati pomoću funkcije `comp`, rekurzivnim prolaskom kroz listu `e`, na sledeći način:

```
try {
  if (e.eq(SymAtom.NIL))
    return new Cons(2, new Cons(SymAtom.NIL, c));
  else
    return complis(e.cdr(), n, comp(e.car(), n, new Cons(13,c)));
}
catch (SExpException ex) {
  LispSynError(0, e); // unknown error
  return c;
}
```

gde smo „za svaki slučaj“ uokvirili telo funkcije u `try` ... `catch` blok, iako ovakva greška ne bi smela da se desi ako je LispKit LISP program prethodno uspešno prošao sintaksnu proveru.

Takođe, prilikom prevođenja LET i LETREC izraza dobro će nam doći funkcije koje iz njih izdvajaju promenljive i izraze što te promenljive definišu. Tačnije, definisaćemo funkcije `vars` i `exprs`, koje iz liste oblika

$$((x_1 . e_1) \ldots (x_k . e_k))$$

izdvajaju

$$(x_1 \ldots x_k)$$

odnosno

$$(e_1 \ldots e_k)$$

Funkciju `vars` možemo implementirati na sledeći način:

```
private static SExp vars(SExp d) {
  SExp var, list;
  try {
    if (d.eq(SymAtom.NIL))
      return SymAtom.NIL;
    else
      var = d.caar();
      list = vars(d.cdr());
      return new Cons(var, list);
    }
  catch (SExpException ex) {
    LispSynError(23, d); // invalid variable list
    return SymAtom.NIL;
  }
}
```

gde smo telo funkcije, radi sigurnosti, opet obavili u `try` ... `catch` blok. Promenljiva `var` ovde označava simbolički atom izdvojen iz prvog elementa liste `d`, a promenljiva `list` listu simboličkih atoma izdvojenih iz ostatka `d`. Ako bi se `var` pojavio u `list`, to bi značilo da je došlo do dupliranja imena promenljivih u LET ili LETREC izrazu, što predstavlja grešku. Provera da li je do ove greške došlo mogla bi se implementirati ubacivanjem naredbe

```
if (member(var, list))
  LispSynError(26, var); // multiple occurences of symbol
```

u telo funkcije odmah nakon dodeljivanja vrednosti promenljivama `var` i `list`. Funkcija

```
private static boolean member(SExp x, SExp a) { ... }
```

jednostavno proverava da li je atom `x` element liste `a`, i shodno tome vraća logičku vrednost `true` ili `false`.

Funkciju `exprs` implementiraćemo analogno funkciji `vars`, naravno, bez provere dupliranja.

Za uspešno „prevođenje“ promenljivih, odnosno simboličkih atoma kao LispKit LISP izraza, biće nam potrebno da implementiramo funkciju *location*, koju smo definisali na strani 33. To ćemo učiniti pomoću sledećih funkcija:

```
private static SExp location(SExp x, SExp n) {
  try {
    if (member(x, n.car()))
      return new Cons(0, position(x, n.car()));
    else {
      SExp z = location(x, n.cdr());
      return new Cons(z.car().intValue() + 1, z.cdr());
    }
  }
  catch (SExpException ex) {
    LispSynError(25, x); // undefined symbol
    return new Cons(-1, -1);
  }
}

private static int position(SExp x, SExp a) {
  try {
    if (x.eq(a.car()))
      return 0;
    else
      return 1 + position(x, a.cdr());
  }
  catch (SExpException ex) {
    LispSynError(25, x); // undefined symbol
    return -1;
  }
}
```

gde funkcija `position` vraća redni broj atoma `x` u listi `a`. Ovog puta `try` ... `catch` blok u obe funkcije nije tu „za svaki slučaj“ – izuzetak označava da se ime `x` ne nalazi u listi imena `n`, što predstavlja grešku.

Sada smo spremni da pređemo na implementaciju funkcije `comp`. Neizbežni blok

```
try {
  ......
catch (SExpException ex) {
  LispSynError(0, e); // unknown error
  return c;
}
```

i ovog puta će obavijati telo funkcije i „hvatati“ sve greške koje su nam, eventualno (mada malo verovatno), na drugim mestima promakle. Telo funkcije imaće sledeću strukturu:

```
if (e instanceof Atom)
  ...
else if (e.car().eq("QUOTE"))
  ...
else if (e.car().eq("EQ"))
  ...
       .........

else if (e.car().eq("LETREC"))
  ...
else
  ...
```

tj. `e` će biti obrađen u zavisnosti od toga koja je vrsta LispKit LISP izraza. Ono što sledi je, prema tome, direktna implementacija pravila prevođenja koja smo opisali u odeljku 3.1 uz, ne zaboravimo, nadovezivanje liste `c` na rezultat prevođenja izraza `e`. Opišimo šta se dešava za svaku vrstu izraza `e`.

Ako je `e`:

Simbolički atom – prevođenje implementiramo jednostavno:

```
return new Cons(1, new Cons(location(e, n), c));
```

QUOTE izraz – ni on nije mnogo komplikovaniji:

```
return new Cons(2, new Cons(e.cadr(), c));
```

EQ izraz – ima dva parametra koji su izrazi koje takođe treba prevesti:

```
return comp(e.cadr(), n, comp(e.caddr(), n, new Cons(14, c)));
```

ADD, **SUB**, **MUL**, **DIV**, **REM**, **LEQ** izrazi prevode se analogno EQ izrazu – razlika je samo u broju instrukcije.

CAR izraz – prevodi se slično kao i prethodni izrazi, samo što ima jedan parametar:

```
return comp(e.cadr(), n, new Cons(10, c));
```

CDR i **ATOM** izrazi prevode se analogno CAR izrazu.

CONS izraz – sličan je ostalim dvoparametarskim izrazima, razlika je u redosledu prevođenja parametara:

```
return comp(e.caddr(), n, comp(e.cadr(), n, new Cons(13, c)));
```

IF izraz – prvo ćemo prevesti dva izraza od kojih se bira vrednost celog IF izraza, dodajući im na kraj instrukciju JOIN (9):

```
SExp thenPt = comp(e.caddr(), n, new Cons(9, SymAtom.NIL));
SExp elsePt = comp(e.cadddr(), n, new Cons(9, SymAtom.NIL));
```

a zatim ih ukomponovati u celokupan prevod:

```
return comp(e.cadr(), n,
            new Cons(8, new Cons(thenPt, new Cons(elsePt, c))));
```

LAMBDA izraz – kada prevedemo telo funkcije i dodamo mu na kraj instrukciju RTN (5):

```
SExp body = comp(e.caddr(), new Cons(e.cadr(), n),
                 new Cons(5, SymAtom.NIL));
```

ostatak prevoda je jednostavan:

```
return new Cons(3, new Cons(body, c));
```

Ovde je prilika da obradimo i poslednju sintaksnu grešku koja nam je preostala – pojavu duplih imena u listi simboličkih atoma u LAMBDA izrazu. Ako funkcija `isSet` vraća logičku vrednost `true` ukoliko joj je argument lista atoma bez ponavljanja, dok inače vraća `false`, tada proveru ove greske možemo implementirati ubacivanjem sledeće naredbe pre početka prevođenja LAMBDA izraza:

```
if (!isSet(e.cadr()))
  LispSynError(26, e.cadr()); // multiple occurences of symbol
```

LET izraz – ako ga posmatramo kao poziv funkcije, onda se, nakon što izrazimo proširenu listu imena, listu argumenata pri pozivu, i prevedemo telo funkcije (u odnosu na proširenu listu imena):

```
SExp m = new Cons(vars(e.cddr()), n);
SExp args = exprs(e.cddr());
SExp body = comp(e.cadr(), m, new Cons(5, SymAtom.NIL));
```

prevod LET izraza svodi na prevod poziva funkcije:

```
return complis(args, n,
               new Cons(3, new Cons(body, new Cons(4, c))));
```

LETREC izraz – vrlo je sličan LET izrazu:

```
SExp m = new Cons(vars(e.cddr()), n);
SExp args = exprs(e.cddr());
SExp body = comp(e.cadr(), m, new Cons(5, SymAtom.NIL));
return new Cons(6,
  complis(args, m, new Cons(3, new Cons(body, new Cons(7, c)))));
```

Poziv funkcije – jednostavno se implementira pomoću funkcije `complis`, kao što smo mogli videti kod LET i LETREC izraza:

```
return complis(e.cdr(), n, comp(e.car(), n, new Cons(4, c)));
```

Ovim je funkcija `comp` kompletirana. Ostalo je još da definišemo funkciju `compile`:

```
static SExp compile(SExp e) {
  return comp(e, SymAtom.NIL,
              new Cons(4, new Cons(21, SymAtom.NIL)));
}
```

Dakle, jednostavno prevodimo izraz `e` u odnosu na praznu listu imena, i na kraj dodajemo instrukcije AP i STOP. Objašnjenje ovog dodavanja sledi u sledećem odeljku.

4.4 Interpreter

Interpreter SECD mašine i njenog mašinskog jezika, koje smo opisali u glavi 3, implementiraćemo klasom

```
class Interpreter {
  ...
}
```

koja ima samo jednog vidljivog člana – funkciju

```
static SExp exec(SExp code, SExp argList) {
  ...
}
```

Ova funkcija kao argumente prima s-izraze `code` i `argList` koji redom predstavljaju kôd SECD mašine koji treba da se izvrši i listu argumenata koji mu se prosleđuju. Već smo rekli da je LispKit LISP program *funkcija*, što znači da je rezultat izvršavanja koda na koji je taj program preveden *zatvorenje*, smešteno na vrhu steka. Kao što smo videli u prethodnom odeljku, kodu se na kraj dodaje instrukcija AP, koja se stara da se ta funkcija, odnosno zatvorenje, odmah i pozove. Dakle, kodu treba proslediti odgovarajuće argumente, i postaviti ih na vrh steka, da bi se ovaj ispravno izvršio.

Preciznije, ako je e ispravan LispKit LISP program, e`*NIL|(AP STOP)` SECD kod dobijen primenom funkcije `compile` na e, i v lista argumenata koje želimo da prosledimo ovom programu, tada je potrebno na sledeći način napuniti registre SECD mašine:

(v) `NIL` e`*NIL|(AP STOP) NIL`

Nakon izvršavanja koda e∗`NIL`, njegov rezultat, tj. zatvorenje, postavljeno je na vrh steka:

$$((c'.e')\ v)\ \texttt{NIL}\ \texttt{(AP STOP)}\ \texttt{NIL}$$

Za ovako dosledno ponašanje treba da zahvalimo *osobini prevedenih izraza* iz odeljka 3.1, čija je posledica da nakon izvršavanja koda dobijenog prevođenjem izraza sadržaji okruženja i skladišta nisu promenjeni, a stek je drugačiji samo po zatvorenju pridodatom na vrh. Zahvaljujući toj osobini, nakon izvršenja instrukcije AP stanje SECD mašine je oblika

$$(x)\ \texttt{NIL}\ \texttt{(STOP)}\ \texttt{NIL}$$

gde je x rezultat primene funkcije, odnosno zatvorenja, na listu argumenata v.

Povratna vrednost funkcije `exec` je rezultat izvršavanja koda `code` primenjenog na argumente `argList`, tj. spomenuto x.

Prirodno, unutar klase `Interpreter` biće nam potrebne promenljive

```
private static SExp s, e, c, d;
```

koje predstavljaju *glavne registre* SECD mašine: stek, okruženje, kontrolnu listu i skladište; kao i jedan *radni registar*[2]:

```
private static SExp w;
```

koji će nam služiti za čuvanje privremenih vrednosti prilikom izračunavanja, i u druge slične svrhe.

Na početku procedure `exec` potrebno je inicijalizovati registre SECD mašine:

```
s = new Cons(argList, SymAtom.NIL);
e = SymAtom.NIL;
c = code;
d = SymAtom.NIL;
```

tj. na vrh steka staviti listu argumenata, a u kontrolnu listu smestiti kod koji treba da se izvrši. Ostatak procedure smestićemo u jedan `try` ... `catch` blok:

```
try {
  ......
}
catch (Exception ex) {
  System.out.println("-- SECD machine runtime error");
  System.out.println("s: " + s);
```

[2]Engl. *working register*.

```
    System.out.println("e: " + e);
    System.out.println("c: " + c);
    System.out.println("d: " + d);
    System.exit(1);
    return SymAtom.NIL;
  }
```

gde se unutar `try` bloka vrši interpretacija, a ako u njenom toku dođe do greške, odnosno izuzetka, biće ispisana odgovarajuća poruka, uz sadržaj svih registara, i program će prestati sa radom. U `catch` bloku se „za svaki slučaj“ hvataju sve vrste izuzetaka (odnosno koren Javine hijerarhije klasa izuzetaka – `Exception`), a ne samo izuzetak `SExpException`. Naravno, pošto je `SExpException` naslednik od `Exception`, i on će biti uhvaćen.

Sadržaj `try` bloka imaće sledeću strukturu:

```
cycle: for (;;) {
  switch (c.car().intValue()) {
    case  1:  ...
    case  2:  ...
       ......
    case 21:  ...
  }
}
return s.car();
```

Princip samog interpretiranja je jednostavan – u svakom ciklusu „beskonačne“ `for` petlje biće izvršena jedna instrukcija, koja se utvrđuje na osnovu prvog elementa registra `c`, odnosno `c.car().intValue()`. Nakon izvršenja instrukcije registar `c` postavlja se na sledeću instrukciju, koja će se izvršiti u narednom ciklusu petlje. Jedini izuzetak je, razumljivo, instrukcija STOP, koja prekida izvršavanje petlje. Ako primetimo da je `for` petlja obeležena labelom `cycle:`, tada je realizacija instrukcije **STOP** jednostavna:

```
case 21:  break cycle;
```

kojom se tok izvršavanja prebacuje na prvu naredbu posle `for` petlje, a ova, jednostavno, izlazi iz funkcije `exec`, vraćajući rezultat izvršavanja SECD programa, što se nalazi na vrhu steka `s`.

Opišimo sada interpretaciju svake instrukcije SECD mašine. Ono što sledi predstavlja, praktično, direktnu implementaciju opisa izvršavanja instrukcija SECD mašine, datog počev od strane 27.

ADD instrukcija se implementira na sledeći način:

```
case 15:  s = new Cons(s.cadr().intValue()+s.car().intValue(),
                       s.cddr());
          c = c.cdr();
          break;
```

Registar `s` postaje nova instanca tačkastog izraza, odnosno lista čiji je prvi element numerički atom dobijen sabiranjem dva elementa sa vrha starog steka, a ostatak liste je stari stek bez ta dva elementa. Kratkoću izražavanja omogućuje nam konstruktor `Cons(int, SExp)` klase `Cons`.

Ostale aritmetičke instrukcije – **SUB**, **MUL**, **DIV** i **REM** interpretiraju se analogno, a i instrukcija **LEQ** je slična:

```
case 20:  if (s.cadr().intValue() <= s.car().intValue())
            s = new Cons(SymAtom.T, s.cddr());
          else
            s = new Cons(SymAtom.F, s.cddr());
          c = c.cdr();
          break;
```

EQ instrukcija je analogna LEQ, razlika je samo u prvom redu:

```
case 14:  if (s.car().eq(s.cadr()))
            ......
```

LDC instrukciju je prilično jednostavno interpretirati:

```
case  2:  s = new Cons(c.cadr(), s);
          c = c.cddr();
          break;
```

dok je **LD** instrukcija nešto složenija:

```
case  1:  w = e;
          for (int i=1; i<=c.caadr().intValue(); i++)
            w = w.cdr();
          w = w.car();
          for (int i=1; i<=c.cdadr().intValue(); i++)
            w = w.cdr();
          w = w.car();
          s = new Cons(w, s);
          c = c.cddr();
          break;
```

Naime, registar `w` koristimo da bi se pomoću prve `for` petlje pozicionirali na podlistu okruženja `e`, čiji je redni broj dat u glavi tačkastog izraza koji je argument instrukcije LD (`c.caadr().intValue()`), a zatim ga koristimo da bismo se pozicionirali na odgovarajući element u okviru te podliste, tj. element sa rednim brojem datim u repu pomenutog tačkastog izraza (`c.cdadr().intValue()`). Na kraju taj element, sadržan u registru `w`, jednostavno prebacujemo na vrh steka `s`.

U jednostavnije instrukcije spadaju još i **CAR**, **CDR**, **ATOM** i **CONS**, čiju ćemo interpretaciju navesti redom:

```
case 10:  s = new Cons(s.caar(), s.cdr());
          c = c.cdr();
          break;

case 11:  s = new Cons(s.cdar(), s.cdr());
          c = c.cdr();
          break;

case 12:  if (s.car() instanceof Atom)
            s = new Cons(SymAtom.T, s.cdr());
          else
            s = new Cons(SymAtom.F, s.cdr());
          c = c.cdr();
          break;

case 13:  s = new Cons(new Cons(s.car(), s.cadr()), s.cddr());
          c = c.cdr();
          break;
```

SEL i **JOIN** instrukcije implementiraćemo na sledeći način:

```
case  8:  d = new Cons(c.cdddr(), d);
          if (s.car().eq(SymAtom.T))
            c = c.cadr();
          else
            c = c.caddr();
          s = s.cdr();
          break;

case  9:  c = d.car();
          d = d.cdr();
          break;
```

Što se tiče instrukcija koje manipulišu zatvorenjima, **LDF** instrukciju možemo implementirati prilično jednostavno:

```
case  3:  s = new Cons(new Cons(c.cadr(), e), s);
          c = c.cddr();
          break;
```

a **AP** instrukciju ovako:

```
case  4:  d = new Cons(s.cddr(),
                       new Cons(e, new Cons(c.cdr(), d)));
          e = new Cons(s.cadr(), s.cdar());
          c = s.caar();
          s = SymAtom.NIL;
          break;
```

Ni **RTN** instrukcija nije veliki problem:

```
case  5:  s = new Cons(s.car(), d.car());
          e = d.cadr();
          c = d.caddr();
          d = d.cdddr();
          break;
```

DUM instrukcija je, praktično, trivijalna:

```
case  6:  e = new Cons(SymAtom.NIL, e);
          c = c.cdr();
          break;
```

gde smo kao Ω jednostavno koristili simbolički atom `NIL`.

Poslednju preostalu instrukciju – **RAP**, interpretiraćemo na sledeći način:

```
case  7:  d = new Cons(s.cddr(),
                       new Cons(e.cdr(), new Cons(c.cdr(), d)));
          e = s.cdar();
          e = e.rplaca(s.cadr());
          c = s.caar();
          s = SymAtom.NIL;
          break;
```

Za kraj, prokomentarišimo ponašanje objekata tipa `SExp`, kada se sa njima manipuliše kao pri izvršavanju svake od instrukcija SECD mašine. Poznato je da su objekti u Javi interno realizovani preko pokazivača, tako da naredba dodele ne kopira *vrednost objekta*, već *internu vrednost pokazivača*. To ima za posledicu da se u toku izvršavanja programa mogu pojaviti objekti u memoriji na koje više ne „pokazuje“ ni jedan interni pokazivač. Takvi objekti se stručno nazivaju „đubre“, a za oslobađanje memorije koju oni zauzimaju zadužen je specijalan proces, deo Java sistema, tzv. *skupljač đubreta*[3].

Prilikom manipulacije registrima SECD mašine pomoću naredbe dodele kad-tad će se pojaviti izvesna količina „đubreta“. Ali, mi o tome uopšte ne moramo da brinemo – to će činiti Javin „skupljač“ umesto nas. Ovo je još jedno od pojednostavljenja implementacije LispKit LISP-a koje nam omogućuje programski jezik Java.

4.5 Interakcija sa korisnikom

Do sada smo u ovom poglavlju opisali skup klasa od kojih svaka implementira određeni deo LispKit LISP sistema. Ostalo je još da se napiše Java

[3]Engl. *garbage collector*.

aplikacija (ili *aplet*) koja koristi sve te klase, a koju će korisnik direktno pokretati i sa njom interagovati. U tu svrhu napisali smo *dve* jednostavne aplikacije, u vidu klasa `LC` i `L`, koje se pokreću sa komandne linije operativnog sistema.

Aplikacija `LC` prevodi LispKit LISP program u kd̂ SECD mašine. Očekuje jedan argument u komandnoj liniji – ime fajla koji sadrži ulazni LispKit LISP program. Posle provere sintaksne ispravnosti, ukoliko se utvrdi da je program ispravan, njegov SECD kod snima se u fajl istog imena kao i fajl sa ulaznim programom, ali sa ekstenzijom „*.secd*“.

Aplikacija `L` izvršava prethodno prevedeni SECD program. Očekuje jedan argument u komandnoj liniji – ime fajla koji sadrži ulazni SECD program. Kao drugi argument može da se navede ime fajla koji sadrži listu argumenata sa kojima SECD program treba da se izvrši. Ako se drugi argument ne navede, korisnik će biti zamoljen da listu argumenata unese sa tastature. Nakon izvršavanja SECD programa, njegov rezltat biće ispisan na ekranu.

Za početak, pretpostavimo da smo pomoću nekog editora teksta u fajl *fac.lkl* uneli program koji računa faktorijel, prikazan na strani 24. Pretpostavimo, takođe, da smo pri tom „slučajno“ načinili nekoliko grešaka, tako da program koji smo uneli izgleda ovako:

```
(LETREC FAC
  (FAC LAMBDA (X)
    (IF (EQ (QUOTE 0))
        (QUOTE 1 1)
        (MUL X (FAC (SUB (QUOTE 1))))
    )
  )
)
```

Pod operativnim sistemom *Windows*, potrebno je u komandnom promptu preći u direktorijum koji sadrži paket (direktorijum) `lispkit`, ili, još bolje, pomoću `SET` komande operativnog sistema dodati taj direktorijum u sistemsku promenljivu `CLASSPATH`. Tada komandom

```
>java lispkit.LC fac.lkl
```

pokrećemo prevođenje programa u fajlu *fac.lkl*.

Radi skraćenja zapisa komandi, napisaćemo dva komandna (engl. *batch*) fajla – *lc.bat* i *l.bat*. Sadržaj fajla *lc.bat* je sledeći:

```
@ECHO OFF
java lispkit.LC %1 %2 %3 %4 %5 %6 %7 %8 %9
```

Dakle, jednostavno se pokreće aplikacija `LC` i prosleđuju joj se svi argumenti sa komandne linije. Sadržaj fajla *l.bat* je analogan. Ako ove komandne

fajlove smestimo u direktorijum koji sadrži paket `lispkit`, i dodamo taj direktorijum u sistemsku promenljivu `PATH`, kao i u `CLASSPATH`, tada aplikacije `LC` i `L` možemo pokretati iz proizvoljnog direktorijuma na disku. Sada komandom

```
>lc fac.lkl
```

pokrećemo prevođenje programa u fajlu *fac.lkl*. Ako smo program uneli sa greškama, onako kako je naveden, dobićemo sledeću reakciju na ekranu:

```
Parsing...
No errors detected
Checking basic lisp syntax...
-- lisp syntax error: invalid EQ expression: (EQ (QUOTE 0))
-- lisp syntax error: invalid QUOTE expression: (QUOTE 1 1)
-- lisp syntax error: invalid SUB expression: (SUB (QUOTE 1))
3 errors detected
```

koja nas obaveštava o pronađenim sintaknim greškama u programu. Kada ispravimo ove greške u fajlu *fac.lkl* i ponovo pokrenemo kompilaciju, trebalo bi da nas obraduje sledeća poruka:

```
Parsing...
No errors detected
Checking basic lisp syntax...
No errors detected
Compiling...
No errors detected
Done
```

i da na disku osvane fajl *fac.secd*. Sada, kad želimo da pokrenemo program, možemo uneti komandu

```
>l fac.secd
```

i time dobiti poruku

```
Parsing SECD code...
No errors detected
** Input argument list:
```

nakon čega aplikacija očekuje da unesemo listu argumenata programa sa tastature. Ako, na primer, unesemo

```
(4)
```

dobićemo obaveštenje

```
Parsing argument list...
No errors detected
Executing...
Result: 24
```

Pređimo na jedan složeniji primer. U pitanju je program koji obrće elemente ulazne liste i svih njenih podlista. Jedno rešenje, adaptirano iz [2], je:

```
(LETREC INVALL
  (INVALL LAMBDA (L)
    (IF (EQ L (QUOTE NIL))
        (QUOTE NIL)
        (IF (ATOM (CAR L))
            (APPEND (INVALL (CDR L))
                    (CONS (CAR L) (QUOTE NIL))
            )
            (APPEND (INVALL (CDR L))
                    (CONS (INVALL (CAR L)) (QUOTE NIL))
            )
        )
    )
  )
  (APPEND LAMBDA (A B)
    (IF (EQ A (QUOTE NIL))
        B
        (CONS (CAR A) (APPEND (CDR A) B))
    )
  )
)
```

gde funkcija `APPEND` vraća listu dobijenu spajanjem lista `A` i `B`, a vrednost funkcije `INVALL`[4] je lista koja sadrži elemente liste `L` i svih njenih podlista, u obrnutom redosledu. Ako dati program snimimo u fajl *invall.lkl* i pokrenemo kompilaciju, dobićemo u fajlu *invall.secd* odgovarajući SECD program. Kada taj program pokrenemo i unesemo kao listu argumenata, na primer,

```
((1 2 (3 4) (5 6)))
```

dobićemo rezultat

```
((6 5) (4 3) 2 1)
```

Primetimo da prilikom unošenja argumenata, u stvari, unosimo *listu* argumenata čiji je jedini element lista, pa otuda potreba za jednim parom zagrada više.

[4]Engl. *invert all* – obrni sve.

Glava 5

Zaključak

Još 1980. godine Piter Henderson, tvorac LispKit LISP-a, u svojoj knjizi [7] predvideo je da će razvoj programskih jezika i, uopšte, programiranja teći putem odvajanja i sve većeg distanciranja od arhitektura konkretnih računara. Bio je mišljenja da će napredak računara usloviti da efikasnost programa, u smislu brzine i utroška memorije, bude manje značajan faktor od njihove čitljivosti, razumljivosti, prenosivosti, lakoće odžavanja i unapređivanja itd. Takođe je smatrao da će, u budućnosti, prevagu u korišćenosti imati programski jezici koji manipulišu simboličkim podacima, koji su jednostavni, uniformni i bliži ljudskom načinu razmišljanja, kao što su, na primer, funkcionalni programski jezici. U implementacijskom smislu to znači da će programi pisani u tim jezicima mahom biti interpretirani, ili prevođeni u jezik neke virtuelne mašine, kao što se LispKit LISP prevodi u kôd SECD mašine.

Možemo slobodno konstatovati da se ova predviđanja uveliko obistinjuju. Doduše, funkcionalni programski jezici nemaju vodeću ulogu u današnjim programerskim trendovima, no ipak se često mogu sresti u odeđenim primenama. Njihov razvoj još uvek je u punom jeku. Programski jezici koji dominiraju u današnjem računarskom svetu su, uglavnom, objektno-orijentisani, a od njih zapaženo mesto zasigurno zauzima Java. Implementacija Jave slično je koncipirana kao i prikazana implementacija LispKit LISP-a – programi se prevode na jezik virtuelne mašine, tzv. *bajtkod*[1], koji se potom interpretira. Ovo ima za posledicu smanjenu efikasnost Java programa, ali sve šira i masovnija upotreba Jave (i drugih slično koncipiranih jezika) u neku ruku potvrđuje Hendersonova predviđanja. Java je prilično jednostavan programski jezik, u poređenju sa nekim drugim objektno-orijentisanim jezicima, kao što je C++.

Takođe, sa svakodnevnim širenjem primene računara u najrazličitije oblasti ljudskog delovanja, sve je teže naći „univerzalni“ programski jezik, koji

[1] Engl. *bytecode*.

bi mogao da posluži u većini tih primena. Programski jezici opšteg tipa, postavši glomazni i puni detalja što, u određenim primenama, mogu više da smetaju nego da pomažu, polako ustupaju mesto usko specijalizovanim programskim jezicima, koji su jednostavni, laki za korišćenje i brzo obavljaju posao za koji su namenjeni.

U tom smislu može se nastaviti rad na ovoj implementaciji programskog jezika LispKit LISP, u koju, da podsetimo, spadaju implementacija s-izraza, njegovo učitavanje pomoću koda generisanog Coco/R-om, provera sintaksne ispravnosti, kompajler u kod SECD mašine, interpreter tog koda i aplikacije zadužene za interakciju sa korisnikom. Pored opštih poboljšanja, kao što su

- implementacija *celokupne* provere sintaksne ispravnosti u klasi koja je za to zadužena – `LispSyntaxChecker`,
- implementacija (bar delimične) semantičke analize,
- realizacija grafičkog korisničkog interfejsa,
- uvođenje uslovne naredbe COND, slično nekim drugim dijalektima LISP-a,

poboljšanja vezana za određenu primenu mogla bi biti sledeća:

- uvođenje novih vrsta numeričkih atoma – realnih, kompleksnih, sa fiksnom tačkom itd.
- jača povezanost sa Javom – mogućnost poziva Java funkcija iz LispKit LISP programa,
- realizacija objektno-orijentisanih proširenja jezika, možda dovodeći ih u direktnu spregu sa Javom,
- uvođenje novih elementarnih funkcija LispKit LISP-a i odgovarajućih instrukcija SECD mašine, u skladu sa željenom primenom jezika.

Kada se, pri dizajnu programskog jezika, zbace stege vezanosti za arhitekturu računara i insistiranja na efikasnosti, može se dobiti jednostavan, a opet moćan programski jezik kao što je LispKit LISP. U prilog njegovoj jednostavnosti govori činjenica da je u ovaj rad stao gotovo kompletan izvorni kod njegove implementacije, proširene proverom sintaksne ispravnosti LispKit LISP programa, sa detaljnim objašnjenjima i opširnim teorijskim uvodom. Ovome je u velikoj meri doprineo programski jezik Java, svojim osobinama koje su pojednostavile implementaciju. LispKit LISP, u obliku kakav je prikazan u ovom radu, bez poboljšanja teško da može da se upotrebi u nekim ozbiljnijim primenama. Međutim, svrha ovog rada nije ni bila da se napravi jezik istog časa spreman za upotrebu, već da ilustruje kako prilično jednostavan dizajn i implementacija programskog jezika mogu da stvore osnovu za mnogo više. Ili, ako ništa drugo, da prikaže kako ono što se dešava „ispod haube“ programskih jezika i programiranja ne mora da bude ni komplikovano ni zastrašujuće, i time dâ podstrek za dalji rad.

Literatura

[1] Z. Budimac, M. Ivanović, *An Implementation of Functional Language Using S-expressions*, in *Proceedings of 14th Information Technologies Conference "Sarajevo 1990"*, pp. 111.1–111.8, Sarajevo, 1990.

[2] Z. Budimac, M. Ivanović, Z. Putnik, D. Tošić, *LISP kroz primere*, Institut za matematiku PMF, Novi Sad, 1994.

[3] Z. Budimac, M. Ivanović, Z. Putnik, D. Tošić, *Programski jezik Scheme*, Institut za matematiku PMF, Novi Sad, 1998.

[4] B. Eckel, *Thinking in Java, 2nd Edition*, Prentice Hall, elektronski dokument, 2000.

[5] J. Gosling, B. Joy, G. Steele, *The Java Language Specification, Edition 1.0*, Sun Microsystems Inc., elektronski dokument, 1996.

[6] J. Gosling, B. Joy, G. Steele, G. Bracha, *The Java Language Specification, Second Edition*, Sun Microsystems Inc., elektronski dokument, 2000.

[7] P. Henderson, *Functional Programming—Application and Implementation*, Prentice Hall, London, 1980.

[8] M. Ivanović, Z. Budimac, *Usage of S-expression in Pascal*, in *Proceedings of 11th International Symposium "Computer at University"*, pp. 3.18.1–3.18.6, Cavtat, 1989.

[9] L. Lemay, R. Cadenhead, *Naučite za 21 dan Java 1.2*, Kompjuter biblioteka, Čačak, 1998.

[10] P. K. McBride, *Java na lak način*, Tehnička knjiga, Beograd, 1997.

[11] G. L. Steele, *Common Lisp the Language, 2nd Edition*, Digital Press, elektronski dokument, 1994.

Biografija

Miloš Radovanović rođen je 2. decembra 1977. godine u Novom Sadu. Godine 1992. završio je onsnovnu školu, kao i nižu muzičku školu, odsek za klavir. Peti razred osnovne škole završio je u Ithaci, NY, SAD. Gimnaziju „J. J. Zmaj“, smer matematičko-programerski saradnik, završava 1996. godine, sa odličnim uspehom. Za vreme školovanja učestvovao je na saveznom i više republičkih takmičenja iz matematike i informatike. Bio je član reprezentacije Novog Sada na međunarodnom „Turniru gradova“, na kojem je osvojeno ekipno prvo mesto. Godine 1996. upisuje Prirodno-matematički fakultet, smer „diplomirani informatičar“. Član je Mense od 1998. U decembru 1999. godine položio je test znanja engleskog jezika i stekao *Certificate of Proficiency in English*. Stipendista je Kraljevine Norveške, za 2000. godinu. Položio je sve predviđene ispite na fakultetu sa prosečnim ocenom 9.20.

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Ključna dokumentacijska informacija

Redni broj: RBR	
Identifikacioni broj: IBR	
Tip dokumentacije: TD	Monografska dokumentacija
Tip zapisa: TZ	Tekstualni štampani materijal
Vrsta rada: VR	Diplomski rad
Autor: AU	Miloš Radovanović
Mentor: MN	dr Mirjana Ivanović
Naslov rada: NR	Implementacija programskog jezika LispKit LISP u Javi
Jezik publikacije: JP	srpski (latinica)
Jezik izvoda: JI	s/en
Zemlja publikovanja: ZP	SR Jugoslavija
Uže geografsko područje: UGP	Vojvodina
Godina: GO	2001
Izdavač: IZ	autorski reprint
Mesto i adresa: MA	Novi Sad, Trg D. Obradovića 4
Fizički opis rada: (broj poglavlja/strana/lit. citata/tabela/slika/grafika/priloga) FO	5/79/11/3/1/0/0

Naučna oblast: NO	Računarske nauke
Naučna disciplina: ND	Konstrukcija kompajlera
Predmetna odrednica/Ključne reči: PO	Programski jezici, Implementacija, Kompajleri, Interpreteri
UDK	
Čuva se: ČU	
Važna napomena: VN	
Izvod: IZ	U radu je dat uvod u funkcionalni stil programiranja, i prikazan programski jezik LispKit LISP. Opisana je SECD mašina i način prevođenja LispKit LISP programa u njen mašinski kôd. Kroz implementaciju ovog prevođenja, i interpretera SECD mašine, ilustrovane su neke mogućnosti programskog jezika Java, i korišćenje generatora Coco/R. Akcenat je stavljen na jednostavnost i ilustrativnost, a ne na praktičnu primenu implementacije.
Datum prihvatanja teme od strane NN veća: DP	septembar, 2001
Datum odbrane: DO	oktobar, 2001
Članovi komisije: (Naučni stepen/ime i prezime/zvanje/fakultet) KO	
Predsednik:	dr Ratko Tošić, redovni profesor Prirodno-matematičkog fakulteta u Novom Sadu
Član:	dr Zoran Budimac, vanredni profesor Prirodno-matematičkog fakulteta u Novom Sadu
Član:	dr Miloš Racković, vanredni profesor Prirodno-matematičkog fakulteta u Novom Sadu

University of Novi Sad
Faculty of Natural Sciences & Mathematics
Key Words Documentation

Accession number: NO	
Identification number: INO	
Document type: DT	Monograph documentation
Type of record: TR	Textual printed material
Contents code: CC	BSC thesis
Author: AU	Miloš Radovanović
Mentor: MN	dr Mirjana Ivanović
Title: TI	An Implementation of the Programming Language LispKit LISP in Java
Language of text: LT	Serbian (Latin)
Language of abstract LA	en/s
Country of publication: CP	FR Yugoslavia
Locality of publication: LP	Vojvodina
Publication year: PY	2001
Publisher: PU	Author's reprint
Publ. place: PP	Novi Sad, Trg D. Obradovića 4
Physical description: (no. chapters/pages/bib. refs/tables/pictures/graphics/appendices) PO	5/79/11/3/1/0/0

Scientific field: SF	Computer science
Scientific discipline SD	Compiler Construction
Subject/Key words: SKW	Programming languages, Implementation, Compilers, Interpreters
UC	
Holding data: HD	
Note: N	
Abstract: AB	In this thesis, an introduction to the functional programming style is given, with a description of the programming language LispKit LISP. The SECD machine is presented, along with rules for translating LispKit LISP programs to SECD machine code. Through an implementation of such translation and an SECD machine interpreter, some aspects of the programming language Java were illustrated, together with the usage of the generator Coco/R. The emphasis is on simplicity and clarity, rather than the practical use of the implementation.
Accepted by Scientific Board on: AS	September, 2001
Defended: DE	October, 2001
Thesis Defend Board: (Degree/first and last name/title/faculty) DB	
President:	dr Ratko Tošić, full professor, Faculty of Science, Novi Sad
Member:	dr Zoran Budimac, associate professor, Faculty of Science, Novi Sad
Member:	dr Miloš Racković, associate professor, Faculty of Science, Novi Sad