UNIVERZITET U NOVOM SADU
PRIRODNO-MATEMATIČKI FAKULTET
INSTITUT ZA MATEMATIKU

mr Zoran Budimac

Prilog teoriji funkcionalnih programskih jezika i implementaciji njihovih procesora

- doktorska disertacija -

Novi Sad
1994

SADRŽAJ

SLIKE

Uvod

Izučavanje programskih jezika i njihovih implementacija je jedna od fundamentalnih oblasti računarskih nauka, bez koje bi razvoj i primena računarstva u bilo kojoj oblasti bilo krajnje otežano. Medju mnogobrojnim klasama programskih jezika, funkcionalni programski jezici od 60-ih godina imaju sve značajniju ulogu u rešavanju tzv. "softverske krize"*, iz sledećih razloga [Turner, 1982]:

- osobine i korektnost funkcionalnih programa se mogu dokazivati formalno, matematičkim metodama,
- funkcionalni programi su deklarativni, kratki, koncizni i laki za čitanje i održavanje,
- funkcionalni programi se na prirodan način mogu izvršavati na paralelenim računarskim arhitekturama, bez uvodjenja dodatnih jezičkih konstrukcija.

Pored toga, funkcionalni programski jezici imaju veliku primenu u istraživanjima u oblasti veštačke inteligencije.

Funkcionalno programiranje se kao stil programiranja pojavilo sa programskim jezikom LISP, krajem 50-ih godina [McCarthy, 1960]. LISP je bio jednim delom inspirisan λ računom [Church, 1941], ali je sadržavao i mnogo karakteristika imperativnih programskih jezika - pogotovu njegove kasnije verzije.

Prvim čistim funkcionalnim programskim jezikom se danas smatra ISWIM [Landin, 1966], koji je u potpunosti bio zasnovan na λ računu. Zajedno sa ISWIM-om, Landin je definisao i apstraktnu mašinu nazvanu SECD [Landin, 1964] pogodnu za "mehaničko izračunavanje λ izraza" - odnosno ISWIM programa.

Sledeći veliki prodor u oblasti funkcionalnog programiranja i funkcionalnih programskih jezika je bio jezik FP [Backus, 1978], koji je bio prvi čisto-funkcionalni jezik koji je privukao širu pažnju istraživača i stručnjaka iz oblasti programskih jezika. Programski jezik SASL [Turner, 1976] je bio prvi čisto-funkcionalni programski jezik nestriktne semantike, prvi jezik koji je omogućavao

---

* Objektno-orijentisano projektovanje i programiranje je nastalo sa istim motivima, ali i sa nešto drugačijim pristupom kreiranju kvalitetnog softvera.

pisanje programa u obliku jednačina i prvi jezik koji je realizovan prevodjenjem u izraze kombinatorske logike [Schönfinkel, 1924; Curry, Feys, 1958].

Početkom osamdesetih godina se intenziviraju istraživanja funkcionalnih programskih jezika i u oblasti definisanja novih jezika i oblasti njihove efikasne implementacije. Tako na primer jezik ML [Gordon, Milner, Morris, Newey, Wadsworth, 1978] prvi uvodi tipove podataka u funkcionalne programske jezike, Hope [Burstall, MacQueen, Sanella 1980], KRC [Turner, 1981a] i Miranda [Turner, 1985] uvode nove sintaksne konstrukcije, a kao deo istraživanja drugačijih računarskih arhitektura, javljaju se i tzv. mašine i jezici sa protokom podataka (engl. *data flow machines and languages*), medju kojima su danas najpoznatiji Id [Arvind, Gostelow, 1982] i SISAL [McGraw, Allan, Glauert, Dobes, 1983]. U poslednjem definisanom funkcionalnom programskom jeziku Haskell [Hudak, Peyton Jones, Wadler, 1992] su sakupljena najvažnija iskustva iz dosadašnjih istraživanja funkcionalnih programskih jezika, a u pokušaju da se stvori de-fakto standard u ovoj oblasti.

Posle otkrića da je kombinatorska logika pogodan način za implementacije funkcionalnih programskih jezika [Turner, 1979, 1981b], dalje se istražuju primene različitih vrsta logika u ovoj oblasti. Tako je na primer na kombinatorskoj logici zasnovana još i SKIM mašina [Stoye, 1983; 1985], na superkombinatorima [Hughes, 1982; 1984] su zasnovane G mašina [Augustsson, 1984], [Johnson, 1984], PAM mašina [Fairbairn, Wray, 1986] i TIM mašina [Fairbairn, Wray, 1987], na teoriji kategorija [Lambek, 1980; Curien, 1986] je zasnovana CAM mašina [Cousineau, Curien, Mauny, 1987] a i SECD mašina se dalje optimizuje i poboljšava.

Od početka osamdesetih godina se istražuje primenljivost funkcionalnog programiranja u rešavanju raznih vrsta problema.

Istraživanje u oblasti funkcionalnog programiranja i funkcionalnih programskih jezika se početkom devedesetih godina karakteriše postojanjem velikog broja funkcionalnih programskih jezika i velikog broja načina za njihovu implementaciju, što je karakteristika mladih i "vitalnih" oblasti. Medjutim, velike medjusobne razlike izmedju jezika i izmedju njihovih implementacija značajno usporavaju dalji razvoj cele oblasti, te se ubrzo javlja potreba za uspostavljanjem standarda. Krajem osamdesetih godina se izdvajaju istraživanja opštih (funkcionalnih) medjujezika kao moguće rešenje problema.

Postojanjem opšteg medjujezika se implementacija funkcionalnih programskih jezika obavlja u dve faze:

a) prevodjenje programa pisanog u izvornom funkcionalnom programskom jeziku u opšti medjujezik i

b) implementacija medjujezika na različitim apstraktnim ili konkretnim računarima.

Medjujezik omogućava jednostavnije razmenjivanje rezultata i programa medju različitim istraživačkim timovima i veću specijalizaciju u istraživanjima. Izučavanja tehnika implementacije medjujezika su tako potpuno odvojene od izučavanja osobina koje funkcionalni jezik treba da poseduje.

Postojeći medjujezici ne zadovoljavaju neke kriterijume koje bi opšti jezik trebao da ispunjava [Peyton Jones, 1988; Peyton Jones, Joy, 1990] i uglavnom su posledica izvornih jezika za čiju implementaciju su originalno poslužili. Tako se na primer, LIF-om [Young, 1989] ne mogu predstaviti jezici striktne semantike, T [Kranz, 1986] omogućava predstavljanje svih jezika ali je predstavljanje nestriktnih jezika komplikovanije, IF1 [Gurd, Kirkham, Watson, 1985] podržava isključivo jezike sa protokom podataka, DACTL [Glauert, Kennaway, Sleep, Somner, 1988] i CLEAN [Brus, van Eekelen, van Leer, Plasmeijer, 1987; van Eekelen, Nöcker, Plasmeijer, Smetsers, 1990; Nöcker, Smetsers, van Eekelen, Plasmeijer, 1991] se mogu implementirati samo na arhitekturama zasnovanim na transformacijama grafa, a FLIC-om [Peyton Jones, 1988] se ne mogu predstaviti netipizirani funkcionalni jezici.

U ovoj doktorskoj tezi je definisan novi opšti medjujezik (nazvan LL) pogodan za implementaciju velikog broja funkcionalnih programskih jezika. Novi medjujezik je definisan kao opšti medjujezik za jedan podskup čisto-funkcionalnih programskih jezika i sadržava dovoljan broj jezičkih konstrukcija i ugradjenih funkcija za jednostavno predstavljanje čisto-funkcionalnih programskih jezika različitih osobina, a da se pri tome ne naruši čitljivost LL programa. Na taj način je medjujezik LL bolji od analiziranih medjujezika, jer:

a) omogućava (podjednako jednostavno) predstavljanje i striktnih i nestriktnih jezika, kao i tipiziranih i netipiziranih jezika,

b) nezavisan je od načina implementacije.

Pored ovih osnovnih prednosti nad analiziranim medjujezicima, LL sadržava i konstrukcije koje podržavaju nove koncepte (funkcionalnih) programskih jezika, kao što je rad sa nizovima, pozivanje "stranih" funkcija, modularnost itd.

Medjujezik LL je definisan na osnovu:

- analize važnijih predstavnika funkcionalnih programskih jezika od LISP-a i ISWIM-a (1966) do Haskell-a (1990-92), i prepoznavanja/klasifikovanja osobina koje ih odredjuju i razlikuju od drugih jezika;

- analize postojećih medjujezika (FLIC, LIF, CLEAN, Dactl itd...) i uočavanja njihovih prednosti i nedostataka;

- višegodišnjeg iskustva autora u implementaciji funkcionalnih programskih jezika posredstvom različitih apstraktnih mašina i analizom performansi izvodjenja funkcionalnih programa na apstraktnim i konkretnim računarskim arhitekturama.

U tezi su dati svi aspekti novog medjujezika: definicija, pravila prevodjenja iz nekoliko izvornih funkcionalnih programskih jezika u medjujezik, implementacija jezika na nekoliko apstraktnih mašina, opisi apstraktnih mašina za implementaciju medjujezika i njihovih proširenja, strukture podataka za implementaciju medjujezika i neki rezultati ispitivanja performansi medjujezika na apstraktnim i konkretnim računarskim arhitekturama.

Pored osnovnog rezultata, u tezi su izloženi i izvedeni rezultati: originalna realizacija "nestandardnih" mogućnosti LL-a na poznatim apstraktnim mašinama (operatori **_apply**, **_foreign**, **_append** itd.), poboljšanja postojećih algoritama za translaciju medjujezika u kombinatorski račun i za prevodjenje jednačina izvornog jezika u **case** izraz ciljnog jezika, definicije 4 funkcionalna programska jezika posredstvom medjujezika i eksperimentalni rezultati koji doprinose razrešenju nekih dilema iz ove oblasti.

Rezultati teze su izloženi u 6 glava:

U prvoj glavi su opisani i analizirani glavni predstavnici funkcionalnih programskih jezika i načina za njihovu implementaciju. Identifikovano je pet osnovnih karakteristika funkcionalnih programskih jezika, na osnovu kojih se mogu podeliti u klase. Načini implementacije jezika su po sopstvenim osobinama i po osobinama jezika za čiju implementaciju su pogodni, podeljeni u tri podskupa, nasuprot uobičajenom gledištu poznatom iz literature [Field, Harrison, 1988; Hankin, Glaser, Till, 1985]. Na osnovu analize postojećih medjujezika [Budimac, 1993], utvrdjeno je da nijedan od njih ne uspeva da predstavi sve osobine postojećih funkcionalnih medjujezika. Osobine koje se najteže podjednako jednostavno prikazuju medjujezikom su striktna / nestriktna semantika, odnosno tipiziranost / netipiziranost izvornog funkcionalnog programskog jezika.

Obrazložena je potreba za novim medjujezikom koji bi mogao da predstavi programe izvornih jezika, nezavisno od toga koju od pomenutih pet osobina (ne) poseduju.

U drugoj glavi je dat opis sintakse i semantike medjujezika LL. Uz opis svake konstrukcije jezika je data motivacija za njeno uvodjenje u medjujezik. Tokom opisa jezika se njegove osobine porede sa osobinama i mogućnostima ostalih medjujezika, a naročito sa FLIC-om [Peytom Jones, 1988] koji je najrašireniji.

Nezavisnost LL-a od (ne)striktne semantike i (ne)tipiziranosti izvornog funkcionalnog jezika je postignuta odvajanjem formalne semantike jezika od tehnika implementacije te semantike, odnosno proglašavanjem nekih konstrukcija

medjujezika notacijom, a ne konstrukcijama programskog jezika strogo definisane semantike. Pojedine ugradjene funkcije jezika striktnu ili nestriktnu semantiku dobijaju tek prevodjenjem u jezik apstraktne mašine.

U trećoj glavi je data translacija tipičnih predstavnika raznih klasa čisto-funkcionalnih programskih jezika u medjujezik LL. Data su pravila prevodjenja za LispKit LISP [Henderson, 1980], ISWIM [Landin, 1966], SASL [Turner, 1978] i Haskell [Hudak, Peyton Jones, Wadler, 1992]. Opisi sintakse jezika ISWIM i SASL su dati u obliku LL(1) gramatike i direktno su primenljivi u bilo kom poznatom "kompajler-kompajleru", a definicija sintakse SASL-a je ispravljena [Budimac, Nikolajević, 1993] u odnosu na gramatiku datu u originalnom radu [Turner, 1976]. Pravila za translaciju Haskell-a u LL predstavljaju jednu od retkih publikovanih kompletnih implementacija Haskell-a [Hudak, Fasel, 1992; Hudak, Peyton Jones, Wadler, 1992].

U četvrtoj glavi su prikazani prevodioci LL-a i definisani su formalnim pravilima prevodjenja medjujezika u jezike karakterističnih apstraktnih mašina za implementaciju funkcionalnih programskih jezika. LL je implementiran prevodjenjem u jezike sledećih apstraktnih mašina: SECD [Henderson, 1980], "lenja" SECD mašina [Henderson, Jones, Jones, 1983], SK redukciona mašina [Turner, 1976], superkombinatorska mašina [Hughes, 1982] i G mašina [Augustsson, 1984; Johnson, 1984]. Mašinski jezici svih navedenih mašina su prošireni, da bi podržali implementaciju pojedinih, nestandardnih, operatora LL-a: stranih funkcija [Ivanović, Budimac, 1990; Budimac, Ivanović, 1991a], funkcije **_apply** [Budimac, Ivanović, 1991b], "lenjih" ugradjenih funkcija [Budimac, Maćoš, 1993] i bibliotečkih funkcija (inspirisanih rešenjima iz [Ivanović, Budimac, Putnik, 1991; Budimac, Ivanović, 1991c].

Pravila za prevodjenje LL-a u jezik lenje SECD mašine su popravljena u odnosu na originalne [Henderson, Jones, Jones, 1983] i daju korektniji mašinski kôd. Popravljen je Diller-ov algoritam [Diller, 1988] za transformaciju proizvoljnog λ izraza u kombinatorski term, koji sada daje 3-4 puta efikasniji mašinski kôd SK redukcione mašine.

U petoj glavi su razmatrane mogućnosti implementacije LL-a, u kome su analizirani praktični aspekti implementacije LL-a na (mikro) računarima sa ograničenim memorijskim prostorom (na kojima je vrlo malo implementacija funkcionalnih programskih jezika). Predložena je uniformna struktura podataka (koje je odgovarajuće proširenje one date u [Budimac, Ivanović, 1989]) koja je pogodna za predstavljanje svih struktura medjujezika: unutrašnje predstavljanje LL programa, osnovnih struktura prevodilaca i simulatora apstraktnih mašina. Iskorišćenost memorijskog prostora je velika, nije na uštrb brzine izvršavanja bilo kog od programa, a većina procedura za upravljanje strukturom podataka je zajednička za sve programe. Prikazane su pojedine procedure za implementaciju čitavog sistema.

Na kraju glave su prikazani eksperimentalni rezultati dobijeni ispitivanjem performansi različitih implementacija LL-a, testiranih na skupu programa koji je reprezentativniji od mnogih testnih primera poznatih iz literature [Koopman, Lee, Siewiorek, 1992; King, 1991]). Ispitivan je uticaj različitih strategija za dodeljivanje novih memorijskih ćelija na performanse sistema i izvedena formula kojom se može aproksimirati gubitak vremena u slučaju izbora neke strategije. Eksperimentalno je utvrdjeno da je implementacija SK mašine sa B* kombinatorom umesto sa B' kombinatorom [Peyton Jones, 1987] značajno efikasnija, iako se u literaturi mogu naći oprečna mišljenja [Peyton Jones, 1987; Diller, 1988]. Utvrdjene su razlike u performansama SK redukcione mašine u zavisnosti od različitih implementacija pomoćnog steka za redukciju kombinatorskog izraza.

U dodatku je navedena potpuna implementacija pravila za translaciju programa pisanih u jeziku ISWIM u medjujezik LL, navedenih u trećoj glavi. Pravila su implementirana definisanjem odgovarajućih semantičkih akcija u definiciji sintakse ISWIM-a i realizovana pomoću "kompajler-kompajler"-a COCO-2 [Dobler, Pirklhauer, 1990].

Za izradu ove disertacije su posredno i neposredno zaslužni sledeći ljudi kojima se na ovaj način zahvaljujem. Pri tome sam naravno, za sve eventualne greške u tekstu disertacije, odgovoran isključivo ja.

Bez delovanja i ostvarenih rezultata "Grupe za funkcionalno programiranje", koja je radila pri Institutu za matematiku u Novom Sadu sredinom osamdesetih godina, ne bi bilo ni kasnijih rezultata iz oblasti funkcionalnog programiranja na ovim prostorima, a verovatno niti ove disertacije. Rukovodilac grupe je bio dr Vojislav Stojković (sada *associate professor* na Morgan State University, SAD), a članovi: dr Ivan Stojmenović (sada *associate professor* na Ottawa University, Canada), Julijana Mirčevski, mr Ljubomir Jerinić (sada asistent na PMF, Novi Sad) i Marija Kulaš (sada asistent na Technische Universität, Dortmund).

Zahvaljujem se svim članovima tzv. "Lige za funkcionalno programiranje", koja početkom devedesetih okuplja zainteresovane za implementaciju funkcionalnih programskih jezika i funkcionalno programiranje uopšte, na inspirativnim sastancima i diskusijama. Posebno sam zahvalan: dr Mirjani Ivanović (docent, PMF, Novi Sad) na primedbama na ranije verzije teksta ove disertacije, kao i na pomoći u implementaciji SECD mašine, lenje SECD mašine i LispKit LISP-a; mr Nenadu Mitiću (asistent, Matematički fakultet, Beograd) na plodnim diskusijama o implementaciji lenje SECD mašine; Branislavu Nikolajeviću (šef Univerzitetskog računskog centra, Univerzitet u Novom Sadu) na realizaciji SASL-a (iako ne na način predložen u ovoj disertaciji), Zoranu Putniku (stručni saradnik, PMF, Novi Sad) na svesrdnom testiranju i korišćenju implementiranih jezika, kao i na implementaciji sintaksnog analizatora i analize zavisnosti za LL, Draganu Maćošu (diplomirani matematičar, PMF, Novi Sad) na realizaciji SK redukcione mašine i delova G mašine i Saši Živkovu (diplomirani matematičar, PMF, Novi Sad) na

pomoći u implementaciji ISWIM-a i realizaciji nove verzije YACC-a, potrebne za implementaciju Haskell-a.

Zahvaljujem se članovima komisije dr Živku Tošiću (redovni profesor, Elektronski fakultet, Niš) i dr Draganu Acketi (vanredni profesor, Prirodno-matematički fakultet, Novi Sad) na korisnim savetima i komentarima pri čitanju rukopisa, te na strpljenju i istrajnosti. Posebno se zahvaljujem mentoru dr Djuri Pauniću (vanredni profesor, PMF, Novi Sad) i članu komisije dr Dušanu Tošiću (docent, Matematički fakultet, Beograd) na pomoći u izboru teme doktorske disertacije, kao i na korisnim sugestijama i savetima u toku i posle njene izrade.

Na kraju (ali ne i najmanje) zahvaljujem se supruzi Mirjani Ivanović, roditeljima i prijateljima na moralnoj podršci i razumevanju koje su ispoljili u toku izrade ove disertacije.

U Novom Sadu
april, 1994.

kandidat
mr Zoran Budimac

Glava 1

Pregled funkcionalnih programskih jezika

U ovoj glavi je dat kratak pregled najpoznatijih funkcionalnih programskih jezika, načina njihove implementacije i postojećih medjujezika. Na osnovu analize funkcionalnih programskih jezika, njihove implementacije i postojećih medjujezika istaknuta je potreba za novim medjujezikom, čija definicija je tema ove doktorske teze.

1.1 Osnovni pojmovi

U ovom odeljku se uvode osnovni pojmovi koji će se koristiti u daljem tekstu.

Izraz programskog jezika je pravilo izračunavanja zapisano po (sintaksnim) pravilima programskog jezika. Rezultat računanja izraza je vrednost. Izraz se gradi od operatora i operanada: operatori pripadaju skupu svih operatora programskog jezika, a operandi mogu biti konstante programskog jezika, identifikatori kojima je pridružena neka vrednost, pozivi ugradjenih funkcija programskog jezika, pozivi korisničkih funkcija i izrazi.

"Ekvivalentnost pominjanja" (engl. *referential transparency*) je osobina (programskog jezika) po kojoj se u svakom izrazu (programskog jezika) svaki podizraz može zameniti sopstvenom vrednošću unutar dosega (engl. *scope*) identifikatora koji se javljaju u izrazu, tako da vrednost izraza ostane nepromenjena. Ako je ispunjena ova osobina, tada se svaki izraz može zameniti drugim izrazom ako oba imaju istu vrednost.

Funkcionalno programiranje (engl. *functional programming*) je stil (način) programiranja u kome je program izraz, a izvršavanje takvog programa je izračunavanje vrednosti izraza. Ukoliko u izrazu učestvuju korisničke funkcije, njihove definicije su takodje deo programa funkcionalnog stila programiranja. U formiranju izraza se može koristiti kompozicija funkcija, a definicije korisničkih

funkcija mogu biti rekurzivne. Funkcionalni program se sastoji od izraza čiju vrednost je potrebno izračunati i definicija identifikatora koji se javljaju u tom izrazu. Funkcionalni stil programiranja je najbolje podržan posebnom klasom programskih jezika - funkcionalnim programskim jezicima.

Čisto-funkcionalni programski jezici (engl. *purely functional programming languages*) su programski jezici koji podržavaju funkcionalno programiranje i u kojima za svaki mogući izraz programskog jezika važi osobina ekvivalentnosti pominjanja. Čisto-funkcionalni programski jezici imaju sledeće osobine [Hudak, 1989]:

- nepostojanje naredbi
  Naredbom se utiče na vremenski ustrojenu promenu stanja rešavanja problema. Od naročitog značaja je nepostojanje naredbe dodeljivanja.
- nepostojanje eksplicitnog zadavanja redosleda izvršavanja
  Čisto-funkcionalni program je "statički" i "opisuje" neku vrednost, koja ne zavisi od redosleda izračunavanja. Zbog toga ne postoji pojam redosleda, niti kontrolnih struktura, petlji itd.
- nepostojanje sporednih efekata (engl. *side effects*)
  Vrednost primene funkcije na argumente zavisi samo od vrednosti argumenata.
- tretiranje funkcija kao ravnopravnih objekata jezika
  Funkcije su ravnopravne sa ostalim objektima jezika, što znači da mogu da budu argument druge funkcije, rezultat druge funkcije, mogu da budu smeštane u strukture podataka itd. Funkcije čiji argumenti i/ili rezultati su funkcije se često nazivaju funkcijama višeg reda (engl. *higher order functions*)
- statičko vezivanje identifikatora
  Identifikatori se za svoje vrednosti vezuju tokom prevodjenja programa, a ne tokom izvršavanja programa.

Funkcionalni programski jezici (engl. *functional programming languages*) su programski jezici koji podržavaju funkcionalno programiranje, ali u kojima osobina ekvivalentnosti pominjanja ne važi za svaki mogući izraz programskog jezika. Za takve funkcionalne programske jezike se često kaže da sadrže elemente imperativnog (proceduralnog) stila programiranja.

Čisto-funkcionalni podskup funkcionalnog programskog jezika p je deo definicije jezika p kojim se generišu samo izrazi za koje važi ekvivalentnost pominjanja.

Do kraja teze će se oznakom F označavati skup čisto-funkcionalnih programskih jezika i čisto-funkcionalni podskupovi funkcionalnih programskih jezika. U daljem tekstu će se takodje pri razmatranju funkcionalnih programskih jezika podrazumevati njihovi čisto-funkcionalni podskupovi.

U daljem tekstu se skup F deli na klase, u zavisnosti od osobina izraza čisto-funkcionalnih programskih jezika. Svi izrazi programskih jezika koji se budu pominjali do kraja ove glave će biti zapisani u apstraktnoj sintaksi.

Izraz $\mathbf{f}\ \mathbf{e}_1 \ldots \mathbf{e}_n$, pri čemu je **f** operator, identifikator ugradjene funkcije, identifikator korisničke funkcije, ili izraz čija je vrednost funkcije a $\mathbf{e}_i$, $i=1,\ldots,n$

operandi, je striktne semantike, ako za svako e_i, $i=1,\ldots,n$, važi da ako je e_i nedefinisano, tada je nedefinisan i čitav izraz. Izraz je nestriktne semantike ako nije striktne semantike. Skup F se u odnosu na striktnost semantike može podeliti u dve klase: F^S_- (nestriktni čisto-funkcionalni programski jezici) čiji su elementi oni programski jezici iz skupa F čiji je svaki izraz nestriktne semantike; i F^S_+ (striktni čisto-funkcionalni programski jezici) čiji su elementi programski jezici iz F čiji je bar jedan izraz striktne semantike. Po definiciji važi: $F=F^S_- \cup F^S_+$, $F^S_- \cap F^S_+ = \varnothing$.

"Lenjo" izračunavanje (engl. *lazy evaluation*) je način izračunavanja vrednosti izraza u kome se vrednost svakog argumenta u pozivu ugradjene ili korisničke funkcije izračunava najviše jedanput. Potpuno "lenjo" izračunavanje (engl. *fully lazy evaluation*) je način izračunavanja vrednosti izraza u kome se vrednost svakog operanda proizvoljnog izraza programskog jezika izračunava najviše jedanput, pošto su identifikatori koji u njemu učestvuju vezani za svoje vrednosti. (Potpuno) "lenjo" izračunavanje je čest način za relizaciju nestriktne semantike*. "Vredno" izračunavanje (engl. *eager evaluation*) je način izračunavanja vrednosti izraza u kome se vrednost svakog operanda proizvoljnog izraza programskog jezika izračunava tačno jedanput.

Izraz (čisto-funkcionalnog) programskog jezika je tipiziran (engl. *typed, typeful*) ako je svakom identifikatoru koji učestvuje u izrazu pridružen tip podataka. Izraz (čisto-funkcionalnog) programskog jezika je netipiziran (engl. *untyped*) ako nije tipiziran. Skup F se u odnosu na tipiziranost izraza može podeliti u dve klase: F^T_- (netipizirani čisto-funkcionalni programski jezici) čiji su elementi oni programski jezici iz skupa F čiji je bar jedan izraz netipiziran; i F^T_+ (tipizirani čisto-funkcionalni programski jezici) čiji su elementi programski jezici iz F čiji je svaki izraz tipiziran. Zbog toga što je u netipiziranim čisto-funkcionalnim programskim jezicima tip podataka pridružen identifikatorima, uskladjenost tipova podataka je moguće kontrolisati tokom prevodjenja programa ("statički"). U netipiziranim (čisto-funkcionalnim) programskim jezicima je tip podataka pridružen vrednostima te se uskladjenost tipova podataka može kontrolisati samo tokom izvršavanja programa ("dinamički"). Po definiciji važi $F=F^T_- \cup F^T_+$, $F^T_- \cap F^T_+ = \varnothing$.

Funkcija **f** od **n** argumenata je definisana kao Curry-jeva funkcija (engl. *curried function*), ako je rezultat njene primene na $k<n$ argumenata $x_1,\ldots,x_k$ nova funkcija **f'** od **n-k** argumenata za koju važi $f'\ x_{k+1} \ldots x_n = f\ x_1 \ldots x_n$. Skup F se u odnosu na postojanje Curry-jevih funkcija može podeliti u dve klase: F^C_- (čisto-funkcionalni programski jezici bez Curry-jevih funkcija) čiji su elementi oni programski jezici iz skupa F u kojima bar jedna korisnička funkcija nije definisana kao Curry-jeva funkcije; i F^C_+ (čisto-funkcionalni programski jezici sa Curry-jevim funkcijama) čiji su elementi programski jezici iz F čije su sve korisničke funkcije definisane kao Curry-jeve funkcije. Po definiciji važi: $F=F^C_- \cup F^C_+$, $F^C_- \cap F^C_+ = \varnothing$.

---

* Nestriktna semantika se može realizovati i tzv. "popustljivim" izračunavanjem (engl. *lenient evaluation*) [Traub, 1991].

Definisanje funkcije kao Curry-jeve je jedan od načina za definisanje funkcije čija je vrednost funkcija. Funkcija čija je vrednost funkcija se može definisati i anonimnom funkcijom. Izraz oblika $\lambda x_1 ... x_k ; e$ se naziva anonimnom funkcijom (ili lambda izrazom čisto-funkcionalnog programskog jezika) i njegova vrednost je funkcija koja zavisi od argumenata $x_1,...,x_k$ i u kojoj je izrazom **e** definisano pravilo izračunavanja vrednosti funkcije.

Ako je **f** Curry-jeva funkcija od **n** argumenata i sa telom **e**, tada je definicija funkcije f: $\mathbf{f}\ x_1 \ ...\ x_n = \mathbf{e}$ ekvivalentna sa sledećom definicijom identifikatora f: $\mathbf{f} = \lambda x_1 ; (\lambda x_2 ; (...; (\lambda xn ; e))...)$. Dokaz ove ekvivelencije sledi direktno na osnovu definicija Curry-jeve i anonimne funkcije.

U ovoj tezi će se razmatrati implementacija čisto-funkcionalnih programskih jezika translacijom programa u odgovarajuće programe medjujezika i prevodjenjem programa medjujezika u mašinske programe apstraktnih mašina.

Translacija (engl. *translation*) je postupak transformacije programa pisanog u nekom (izvornom) programskom jeziku u odgovarajući program u drugom (ciljnom) programskom jeziku. Prevodjenje (engl. *compilation*) je translacija programa pisanog u mašinski nezavisnom programskom jeziku u program mašinski zavisnog programskog jezika*. Translatorima i prevodiocima se nazivaju izvršni programi koji vrše odgovarajuću transformaciju (translaciju, odnosno prevodjenje). Poštujući gornju definiciju, u daljem tekstu ove teze će se (opštiji) termin "translacija" upotrebljavati za transformaciju programa čisto-funkcionalnih programskih jezika u odgovarajuće programe (mašinski-nezavisnog) medjujezika; a termin "prevodjenje" za transformaciju programa (mašinski-nezavisnog) medjujezika u mašinske jezike apstraktnih mašina.

Funkcionalni programski jezici, λ račun (teoretska osnova jednog dela funkcionalnih programskih jezika) i funkcionalno programiranje su detaljnije opisani u prvobitnim referencama o funkcionalnim programskim jezicima (u kojima su definisani) i u [Barendregt, 1984; Hudak, 1989; Budimac, Ivanović, 1993a] (λ račun i njegov uticaj na funkcionalno programiranje); i u [Turner, 1982; Bird, Wadler, 1988; Hudak, 1989; Budimac, Ivanović, Putnik, Tošić, 1991] (funkcionalno programiranje).

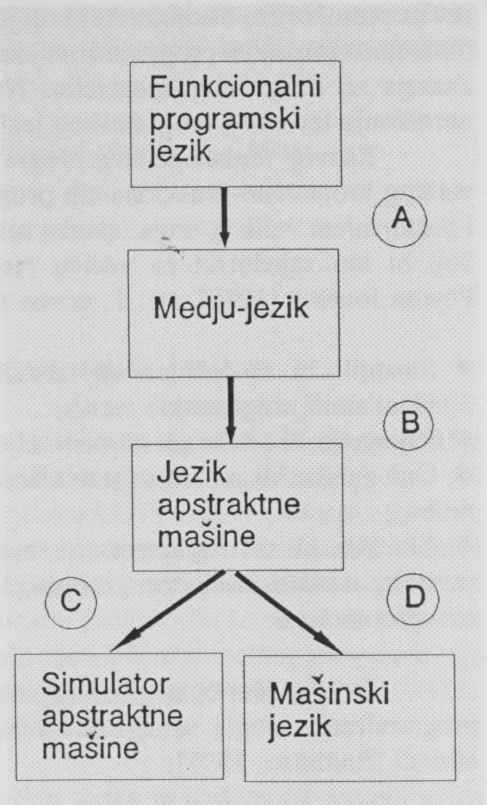
1.2 Princip implementacije čisto-funkcionalnih programskih jezika

Princip na kome se danas često zasniva implementacija čisto-funkcionalnih programskih jezika je prikazan na Sl. 1.1 i predstavlja zadovoljavajući kompromis izmedju jednostavnosti implementacije, mogućnosti optimizacija i portabilnosti implementiranog čisto-funkcionalnog programskog jezika.

---

* Drugi mogući prevod engleskih termina *translation* i *compilation* je "prevodjenje" i "kompilacija".

- Program izvornog čisto-funkcionalnog programskog jezika se prvo translira u neki medjujezik (faza A) (ponekad i u više koraka), a obično je propraćeno kontrolom tipova podataka jezika (ukoliko izvorni jezik pripada klasi F^{T}_{+}).
- Zavisno od vrste medjujezika, na medjuprogramima je moguće vršiti razne transformacije, analize i optimizacije, posle čega se vrši prevodjenje u jezik izabrane apstraktne mašine (faza B). Izbor apstraktne mašine može da zavisi od medjujezika, ali se često bira neka od standardnih apstraktnih mašina za funkcionalne programske jezike.
- Apstraktna mašina koja se koristi za izvršavanje funkcionalnog programa se može programski simulirati na konkretnom računaru (faza C) ili se mašinski jezik apstraktne mašine može prevesti u mašinski jezik konkretnog računara (faza D). U slučaju D je moguće izvršiti još neke optimizacije nad mašinskim kodom konkretnog računara.



Sl. 1.1 Princip implementacije

Raspored optimizacija i provera koji su neophodni u procesu translacije izvornog programa u izvršni oblik (provera tipova, analiza zavisnosti, analiza striktnosti...) se ne može unapred utvrditi na datoj šemi. Potreba za raznim analizama i optimizacijama i mesto na kome će se one izvršiti zavise od odabranog matematičkog modela, medjujezika i apstraktne mašine. Na primer, jedan broj optimizacija u medjujeziku zasnovanom na transformaciji grafa nije potreban, jer su optimizacije već "ugradjene" u definiciju medjujezika.

Prevodjenje izvornog programa u program mašinskog jezika nekog apstraktnog računara je čest postupak i u implementaciji programskih jezika drugih stilova programiranja (Pascal, C, Modula-2, PROLOG itd.), a pogotovo u implementacijama koje treba da budu lako prenosive izmedju različitih platformi. Prethodna translacija u medjujezik je u implementaciji programskih jezika drugih stilova programiranja manje prisutna.

1.2.1 Opšti medjujezici - problemi i značaj

Trenutno stanje istraživanja u oblasti implementacije čisto-funkcionalnih programskih jezika je da mnogi istraživački timovi imaju sopstveni čisto-funkcionalni programski jezik (ili sopstvenu verziju "tudjeg" jezika). Slična situacija

je i sa postojanjem medjujezika, koji je u najvećem broju slučajeva vrlo prilagodjen čisto-funkcionalnom programskom jeziku koji se implementira i zbog toga od malog značaja za druge implementacije. Najčešće je medjujezik samo malo sintaksno uprošćenje izvornog programskog jezika.

Razvoj funkcionalnog programiranja je umnogome usporen postojanjem velikog broja čisto-funkcionalnih programskih jezika koji su danas u upotrebi kao i postojanjem velikog broja tehnika njihove implementacije. Postojanje medjujezika koji bi bio zajednički za većinu čisto-funkcionalnih programskih jezika, bi po Peyton Jones-u [1988], str. 1, doneo sledeće prednosti:

- Smanjili bi se loši uticaji nastali velikim brojem trenutno aktuelnih čisto-funkcionalnih programskih jezika;
- Omogućio bi se širi pristup istraživača najbržim implementacijama;
- Omogućila bi se bolja prohodnost programa izmedju različitih istraživačkih timova;
- Olakšala bi se implementacija nekih aktivnosti prevodioca koje bi se mogle ostvariti transformacijama izvornog na izvorni jezik (engl. *source to source transformation*).

Postojanjem opšteg medjujezika bi se istraživanja u oblasti funkcionalnog programiranja mogla usmeriti u dva potpuno nezavisna pravca i tako značajno ubrzati [Budimac, 1993]:

- Istraživanje osobina i mogućnosti koje čisto-funkcionalni programski jezik treba da poseduje i definisanje novih, moćnijih čisto-funkcionalnih programskih jezika. Teoretska razmatranja bi se završavala definisanjem translatora takvog jezika u opšti medjujezik. Izvorni čisto-funkcionalni programski jezik bi bio potpuno nezavisan od načina implementacije medjujezika.
- Istraživanje tehnika implementacije medjujezika, izborom odgovarajućeg matematičkog modela, apstraktne mašine, struktura podataka i algoritama. Tehnika implementacije bi bila potpuno nezavisna od izvornog čisto-funkcionalnog programskog jezika koji je implementiran posredstvom medjujezika.

Takav opšti medjujezik bi morao da poseduje sledeće (ponekad protivrečne) osobine [Peyton Jones, 1988], str. 2:

1. Da bude zajednički za što veći broj čisto-funkcionalnih programskih jezika.
2. Translacija programa izvornog programskog jezika u medjujezik treba da bude jednostavna.
3. Sintaksa medjujezika treba da bude jednostavna i da sadrži malo konstrukcija, kako bi bila lako mašinski čitljiva.
4. Semantika medjujezika treba da bude jednostavna.

5. Prenošenje izvornog čisto-funkcionalnog programa sa jedne na drugu implementaciju medjujezika ne sme da utiče na efikasnost izvršavanja izvornog programa.

Pored toga, medjujezik bi morao da bude i [Budimac, 1993]:

6. Visokog nivoa, tj. lako razumljiv čoveku.
7. Nezavisan od načina implementacije.

1.3 Analiza čisto-funkcionalnih programskih jezika

Razvoj funkcionalnog programiranja je vezan za definicije i implementacije (čisto) funkcionalnih programskih jezika. Funkcionalno programiranje je kao stil začeto pojavom funkcionalnog programskog jezika LISP, krajem pedesetih godina. Iako je LISP bio u osnovi dobro zamišljen, naredne verzije jezika su uvele naredbe dodeljivanja, a time i "sporedne efekte". Funkcionalni programski jezik ISWIM, koga je kreirao Landin [1966], je bio prvi funkcionalni programski jezik jasno zasnovan na matematičkoj osnovi (λ računu). Na kasniji razvoj funkcionalnog programiranja su uticali naredni funkcionalni programski jezici kao što su FP, Hope, ML, SASL, KRC, Ponder, Orwell, ALFL, Miranda, Haskell itd.

Analiza čisto-funkcionalnih programskih jezika je izvšena na osnovu uvida u osobine sledećih jezika (pored naziva jezika su navedene reference u kojima je jezik definisan, opisan, unapredjen ili korišćen):

- čisto-funkcionalni podskupovi raznih verzija **LISP**-a, [McCarthy, 1960], [Winston, Horn, 1981];
- **Scheme**, [Sussman, Steele, 1975], [Abelson, Sussman, Sussman, 1985], [Smith, 1988];
- **LispKit LISP**, [Henderson, 1980], [Henderson, Jones, Jones, 1983], [Stojković i dr., 1984], [Budimac, Ivanović, Putnik, Tošić, 1991], [Budimac, Ivanović, 1993b].
- **ISWIM**, [Landin, 1966], [MacLennan, 1990], [Glaser, Hankin, Till, 1984], [Ivanović, Budimac, 1993], [Živkov, Budimac, Ivanović, 1993];
- **FP**, **FFP** i **FL**, [Backus, 1978], [Williams, 1982], [Field, Harrison, 1988], [Mitić, 1989];
- **SASL**, [Turner, 1976];
- **ML**, [Gordon, Milner, Morris, Newey, Wadsworth, 1978];
- **SML**, [Milner, 1984], [Sokolowski, 1991];
- **LML**, [Augustsson, 1984];
- **ALFL**, [Hudak, 1984];
- **Hope**, [Burstall, MacQueen, Sanella, 1980], [Field, Harrison, 1988];
- **KRC**, [Turner, 1981a], [Glaser, Hankin, Till, 1984], [Diller, 1988];

- **Id**, [Arvind, Gostelow, 1982], [Nikhil, 1988], [Ekanadham, 1991];
- **SISAL**, [McGraw, Allan, Glauert, Dobes, 1983], [Skedzielewski, 1991];
- **Val**, [Ackerman, Dennis, 1979], [McGraw, 1982];
- **FEL**, [Keller, 1982];
- **Lucid**, [Wadge, Ashcroft, 1985];
- **Ponder**, [Fairbairn, 1985];
- **Miranda**, [Turner, 1985], [Bird, Wadler, 1988], [Peyton Jones, 1987];
- **Tui**, [Boutel, 1988];
- **Orwell**, [Wadler, Miller, 1988];
- **Haskell**, [Hudak, Peyton Jones, Wadler, 1991; 1992], [Hudak, 1991], [Hudak, Fasel, 1992];

Na osnovu analize ovih čisto-funkcionalnih programskih jezika uočavamo dve karakteristike na osnovu kojih se skup F može podeliti u nove klase: mogućnost uvodjenja korisničkih tipova podataka i postojanje specijalnih sintaksnih konstrukcija. U specijalne sintaksne konstrukcije čisto-funkcionalnih programskih jezika spadaju: "čuvari" (engl. *guards*), ZF-izrazi (engl. *ZF-expressions*, *list-comprehensions*) i jednačine.

Ako je definicija funkcije **f** od **n** argumenata sledećeg oblika: $\mathbf{f}\ x_1 \ldots x_n = e_1\ s_1; \ldots; e_k\ s_k$, tada su čuvari logički izrazi s_i kojima se kontroliše izbor jednog od izraza e_i, $1 \le i \le k$. Vrednost primene funkcije **f** na argumente je jednaka vrednosti izraza e_i, ako je vrednost izraza s_i, $i \in \{1,\ldots,k\}$ logička vrednost "tačno", a vrednosti svih izraza s_j, $1 \le j < i$ "netačno". Vrednost ZF izraza oblika $e;\ q_1,\ldots,q_n$, je lista čiji su elementi jednaki vrednosti izraza **e**, pri čemu se vrednosti identifikatora od kojih se sastoji izraz **e** uzimaju redom iz "generatora" vrednosti q_i, $i \in \{1,\ldots,n\}$ ukoliko zadovoljavaju logičke vrednosti ("filtere") definisane izrazima q_j, $j \in \{1,\ldots,n\}$, $j \neq i$. "Generator" vrednosti je izraz čija vrednost je lista.

Ako je definicija funkcije **f** od **n** argumenata sledećeg oblika:

$$\mathbf{f}\ x_1^1\ x_1^2 \ldots x_1^n = e_1\ ;\ \mathbf{f}\ x_2^1\ x_2^2 \ldots x_2^n = e_2\ ;\ \ldots\ ;\ \mathbf{f}\ x_k^1\ x_k^2 \ldots x_k^n = e_k$$

tada se $\mathbf{f}\ x_i^1\ x_i^2 \ldots x_i^n = e_i$, naziva jednačinom, pri čemu su e_i izrazi, a x_i^j oblici (engl. *pattern*) programskog jezika, $i \in \{1,\ldots,k\}$, $j \in \{1,\ldots,n\}$. Oblik je posebna sintaksna konstrukcija (čisto-funkcionalnog) programskog jezika, na osnovu koga se vrši izbor jedne od jednačina tokom izvršavanja programa. Vrednost primene funkcije **f** na argumente $a_1,\ldots,a_n$ je jednaka vrednosti izraza e_i, $i \in \{1,\ldots,k\}$ ako se svi oblici x_i^j, $1 \le j \le n$ "podudaraju" (engl. *pattern matching*) redom sa vrednostima argumenata a_j i ako se bar jedan od oblika x_m^j, $1 \le j \le n$, $1 \le m < i$ ne podudara sa vrednošću odgovarajućeg argumenta. Za detalje o oblicima i podudaranju oblika u SASL-u videti odeljak 3.3, a za detalje o oblicima i podudaranju oblika u Haskell-u videti odeljak 3.4.

Skup F se u odnosu na mogućnost uvodjenja korisničkih tipova podataka može podeliti u dve klase: F^U_- čiji su elementi oni programski jezici iz skupa F u kojima nije moguće uvoditi korisničke tipove podataka; i F^U_+ čiji su elementi

programski jezici iz F koji omogućavaju uvodjenje korisničkih tipova podataka. Skup F se u odnosu na postojanje specijalnih sintaksnih konstrukcija može podeliti u dve klase: F^X_- čiji su elementi programski jezici iz F koji ne poseduju ni jednu specijalnu sintaksnu konstrukciju; i F^X_+ čiji su elementi oni programski jezici iz skupa F koji poseduju bar jednu specijalnu sintaksnu konstrukciju. Po definiciji važi $F=F^U_- \cup F^U_+$, $F^U_- \cap F^U_+ = \varnothing$ i $F=F^X_- \cup F^X_+$, $F^X_- \cap F^X_+ = \varnothing$.

Na Sl. 1.2 je prikazana tabela sa važnijim predstavnicima čisto-funkcionalnih programskih jezika u kojoj je njihova pripadnost klasi F^I_+ označena simbolom +, a pripadanost klasi F^I_- simbolom -, $I \in \{S, T, C, U, X\}$.

Na osnovu analiziranih čisto-funkcionalnih programskih jezika mogu se uočiti i sledeće zakonitosti:

- Većina jezika (osim KRC-a) sadrži mogućnost lokalnih definicija;
- Većina jezika iz klase F^T_+ (osim FL-a) uskladjenost tipova proveravaju statički;
- Većina jezika iz klase F^T_+ (osim SISAL-a i Id-a) pripada i klasi F^U_+;
- Jezici iz klase F^T_+ dozvoljavaju definisanje funkcija primenljivih na argumente različitih tipova podataka;
- Većina jezika iz klase F^U_+ (osim ML-a) pored algebarskih tipova podataka dopušta uvodjenje i apstraktnih tipova podataka.
- Većina jezika iz klase F^C_- (osim FP-a i FL-a) sadrže anonimne funkcije.
- Jedino Haskell iz klase F^C_+ sadrži anonimne funkcije.

Jezik	F^S	F^T	F^U	F^C	F^X
Scheme	+	-	-	-	-
LispKit LISP	+	-	-	-	-
ISWIM	+	-	-	-[4]	-
FP	+	-	-	-	-
FFP	+	-	-	-	-
FL	-	+[3]	+	-	-
SASL	-	-	-	+	+
ML	+	+	+	-	-
SML	+	+	+	-	+
LML	-	+	+	-	+
Hope	+[1]	+	+	-	+
KRC	-	-	-	+	+
Id	+[2]	+	-	+	+
SISAL	+	+	-	-	-
Miranda	-	+	+	+	+
Haskell	-	+	+	+	+

Komentari:

1 - Semantika konstruktora podataka je nestriktna
2 - Dodavanjem posebnih oznaka moguće je postići i lenjo izračunavanje
3 - Uskladjenost tipova se proverava dinamički
4 - Funkcija treba da bude **eksplicitno** definisana kao Curryjeva

Sl. 1.2 Osnovne karakteristike pojedinih čisto-funkcionalnih programskih jezika

1.4 Implementacija čisto-funkcionalnih programskih jezika

Čisto-funkcionalni programski jezici po svojoj ideji i karakteristikama zahtevaju i poseban način implementacije. Kako se radi o "mladoj" istraživačkoj oblasti, još uvek ne postoji opšta usaglašenost o tome koja tehnika je najpodesnija za implementaciju čisto-funkcionalnih programskih jezika.

1.4.1 Najvažnije tehnike implementacije

U sledećem odeljku su ukratko opisane najvažnije tehnike implementacije čisto-funkcionalnih programskih jezika.

1.4.1.1 Interpretatori

Svi čisto-funkcionalni programski jezici se mogu implementirati interpretatorima, što je i čest način implementacije raznih dijalekata LISP-a. Ovakav način je, medjutim, najčešće neefikasan jer se tokom interpretiranja korisničkog programa vrše različite provere i koriste velike strukture podataka potrebne za realizaciju (na primer) nestriktne semantike. Noviji jezici takodje sadrže i koncizne specijalne sintaksne konstrukcije, kojima je teško direktno manipulisati. Zbog toga u ovoj tezi interpretatori neće biti posebno razmatrani.

Interpretatori (čisto-funkcionalnih) programskih jezika implementirani u čisto-funkcionalnim programskim jezicima su pogodno sredstvo za opisivanje (operacione) semantike programskih jezika i izradu prototipova interpretatora. Posle testiranja, interpretator se obično implementira u imperativnom programskom jeziku ili se koristi potpuno nova tehnika implementacije. Neki od interpretatora pisanih u čisto-funkcionalnim programskim jezicima se mogu naći u [McCarthy, 1960], [Henderson, 1980] str. 110 i [Glaser, Hankin, Till, 1984] str. 67.

1.4.1.2 SECD mašina

SECD mašinu je definisao Landin [1964] kao specijalizovani (apstraktni) računar za izračunavanje λ izraza, odnosno programa čisto-funkcionalnih programskih jezika zasnovanih na λ računu. SECD mašina se sastoji od 4 registra:

S stek (engl. *stack*), u kome se čuvaju medjurezultati izračunavanja;
E okolina (engl. *environment*) u kome se čuvaju identifikatori λ izraza i njihove vrednosti;
C kontrola (engl. *control*) u kome se čuva λ izraz koji se izračunava;
D ostava (engl. *dump*) u kome se čuvaju sadržine ostala tri registra kada se za to ukaže potreba.

SECD mašina je u početku zamišljena kao mašina za implementaciju jezika striktne semantike. Pojavom jezika nestriktne semantike i SECD mašina je proširena tako da podrži lenjo izračunavanje [Henderson, Morris, 1976; Henderson, 1980]. Henderson je pored toga konstruisao i mašinski jezik SECD mašine, te na taj način omogućio implementaciju čisto-funkcionalnih programskih jezika prevodjenjem izvornog jezika u mašinski jezik SECD mašine.

SECD mašina je iskorišćena za implementaciju mnogih čisto-funkcionalnih programskih jezika od kojih su najpoznatije (prvobitne) implementacije ISWIM-a [Landin, 1966] i LispKit LISP-a [Henderson, 1980; Henderson, Jones, Jones, 1983]. SECD mašina je pored toga opisana i korišćena u [Stojković i dr, 1983], [Glaser, Hankin, Till, 1984] str. 82, [Field, Harrison, 1988], str. 215.

1.4.1.3 Redukcija grafa

Wadsworth [1971] je λ izraz predstavio grafom te ga izračunavao redukcijom grafa po pravilima λ računa. Ovakav način implementacije nije direktno poslužio ni za jednu konkretnu implementaciju funkcionalnih programskih jezika, ali je postao dobra osnova za izvedene tehnike zasnovane na redukciji grafa.

Predstavljanje λ izraza grafom je pogodno za realizaciju nestriktne semantike, ali nepostojanje strukture u kojoj bi se čuvali identifikatori i njihove vrednosti, značajno pogoršava performanse ove tehnike implementacije. Ova tehnika je opisana i u [Peyton Jones, 1987] str. 193 i [Field, Harrison, 1988] str. 241.

1.4.1.4 SK redukciona mašina

Na osnovu teoretskih radova Schönfinkel-a [1924] i Curry-ja i Feys-a [1958] o ekvivalentnosti λ izraza i izraza kombinatorskog računa, Turner [1979] je definisao SK redukcionu mašinu za implementaciju čisto-funkcionalnih programskih jezika. SK redukciona mašina je zasnovana na:

- prevodjenju izvornog programa u ekvivalentni izraz kombinatorske logike,
- predstavljanju kombinatorskog izraza (kombinatora) grafom i
- redukcijom grafa na osnovu pravila redukcije kombinatorskog izraza.

Osnovna prednost kombinatorskog izraza je u tome što ne sadrži identifikatore (pa ni njihove vrednosti), što potencijalno omogućava efikasnije implementacije čisto-funkcionalnih programskih jezika.

Turner [1981b] je dalje unapredio ovu tehniku koja je iskorišćena za implementaciju svih njegovih jezika: SASL [1976], KRC [1981a] i Miranda [1985]. SK redukciona mašina je opisana i u [Peyton Jones, 1987] str. 260, [Diller, 1988], [Field, Harrison, 1988] str. 273 i [Glaser, Hankin, Till, 1983] str. 93.

1.4.1.5 Superkombinatori

Uvodjenjem superkombinatora [Hughes, 1982;1984] efikasnost implementacija zasnovanih na transformaciji grafa se dalje povećava.

Superkombinatori su izrazi λ računa koji zadovoljavaju sledeće uslove: λ apstrakcija oblika $\lambda x_1.\lambda x_2.....\lambda x_n.e$ je superkombinator (u oznaci $s) arnosti **n**, ako: a) $n \geq 0$, b) **e** nije λ apstrakcija, c) $s ne sadrži slobodne identifikatore, d) bilo koja λ apstrakcija koja se nalazi unutar **e** je takodje superkombinator. Implementacija funkcionalnih jezika superkombinatorima je analogna implementaciji zasnovanoj na kombinatorima (SK redukcionom mašinom).

1.4.1.6 G mašina

G mašinu su definisali Augustsson [1984] i Johnson [1984], kao (apstraktni) računar za redukciju superkombinatorskih izraza. G mašina se sastoji od 4 registra:

S stek (engl. *stack*), u kome se čuvaju medjurezultati izračunavanja;
G graf (engl. *graph*), u kome se čuva graf koji se redukuje;
C kontrola (engl. *control*) u kome se čuva mašinski kod G mašine koji se izvršava;
D ostava (engl. *dump*) u kome se čuvaju sadržine registara S i C kada se za to ukaže potreba.

G mašina danas predstavlja jedan od čestih načina za implementaciju čisto-funkcionalnih programskih jezika, a prvobitno je upotrebljena za implementaciju jezika LML [Augustsson, 1984]. G mašina je opisana i korišćena i u [Augustsson, 1987], [Johnsson, 1987], [Field, Harrison, 1988], str. 387, [Peyton Jones, 1987] str. 293.

1.4.1.7 CAM mašina

Na osnovu teoretskog rada Lambek-a [1980], Curien [1986] je definisao razne kategoričke kombinatorske logike (engl. *Categorical Combinatory Logics*) (skraćeno CCL). Implementacija čisto-funkcionalnih programskih jezika posredstvom ovih logika se zasniva na:

- prevodjenju izvornog programa u λ izraz zapisan u de Bruijn-ovoj notaciji (bez identifikatora);
- prevodjenju λ izraza u CCL član;
- redukciji CCL člana;

Redukcija CCL člana se vrši CAM mašinom koju su definisali Cousineau, Curien i Mauny [1987]. CAM mašina se sastoji od 3 registra:

S stek (engl. *stack*), u kome se čuvaju medjurezultati izračunavanja;
C kontrola (engl. *control*) u kome se čuvaju CCL članovi koji još treba da se redukuju;
T član (engl. *term*) u kome se nalazi član koji se izračunava.

CAM mašina je opisana i korišćena i u [Diller, 1988] od str. 136, [Lins, Thompson, 1990].

1.4.1.8 Ostale tehnike implementacije

FP/M (skraćeno od *Functional Programming Machine*) je mašina slična SECD mašini, a definisana je za implementaciju jezika Hope. Neki autori [Field, Harrison, 1988] str. 362, FP/M mašinu nazivaju optimizovanom SECD mašinom.

SKIM [Stoye, 1983;1985] (skraćeno od *S, K, I Machine*) je varijanta SK redukcione mašine u kojoj se pravila redukcije kombinatora prevode u sekvence mašinskih instrukcija koje proizvode isti efekt. Odnos SK redukcione i SKIM mašine je analogan odnosu superkombinatorske i G mašine.

PAM (skraćeno od *Ponder Abstract Machine*) [Fairbairn, Wray, 1986] je apstraktna mašina nastala nezavisno od G mašine, ali po koncepciji i efikasnosti veoma slična G mašini. TIM [Fairbairn, Wray, 1987] (skraćeno od *Three Instruction Machine*) je zasnovana na superkombinatorima, ali po koncepciji različita od G mašine.

Mašine sa tokom podataka (engl. *Data Flow Machines*) se zasnivaju na konceptu toka podataka umesto na konceptu toka kontrole, kako je to u slučaju sekvencijalnih računara. Program mašine sa tokom podataka je izraz predstavljen grafom, u kome su operatori čvorovi grafa, a operandi grane. Pošto takve mašine nemaju globalnu memoriju, pogodne su za implementaciju čisto-funkcionalnih programskih jezika.

1.4.2 Čisto-funkcionalni programski jezici na osnovu implementacije

Obično se u literaturi navedene tehnike implementacije grupišu u sledeća 4 glavna pravca implementacije čisto-funkcionalnih programskih jezika:

- interpretatori,
- SECD mašine (i iz nje izvedene mašine),
- mašine zasnovane na redukciji (kombinatorskog) grafa i
- mašine sa tokom podataka (engl. *data-flow machines*).

Navedena podela, medjutim, više izražava hronologiju razvoja tehnika implementacije čisto-funkcionalnih programskih jezika nego što je zasnovana na njihovim stvarnim karakteristikama. Tako se, na primer, ne može poreći sličnost

G mašine i SECD mašine [Peyton Jones, 1987] str. 324, iako je G mašina zasnovana na redukciji (kombinatorskog) grafa.

Umesto gornje podele, uvedimo podelu na osnovu uticaja izvornog čisto-funkcionalnog programskog jezika na karakteristike apstraktne mašine koja ga implementira; i obrnuto, na osnovu uticaja apstraktne mašine na karakteristike jezika koji je moguće na njoj implementirati. Uvodjenjem takve podele, u skupu F uočavamo sledeće podskupove: F_L (jezici zasnovani na λ računu), F_{FP} (jezici u čijoj je osnovi programski jezik FP) i F_{DF} (jezici zasnovani na toku podataka).

Jezici iz F_L se podjednako uspešno i efikasno mogu implementirati i posredstvom SECD mašina i posredstvom mašina zasnovanih na redukciji grafa. Jezici iz F_{FP} se prirodno implementiraju posredstvom specijalizovanih sistema ili suženih SECD i redukcionih mašina. Jezici iz F_{DF} se najprirodnije izvršavaju na apstraktnim ili realnim mašinama sa tokom podataka.

Uz manje ili više napora se jezik iz svake od nabrojanih grupa može implementirati na svakoj vrsti apstraktnih mašina. Tako na primer Field i Harrison [1988] na str. 355-356 pokazuju ekvivalentnost mašine zasnovane na (redukciji) grafa i jedne klase mašina sa tokom podataka, a Mitić [1989] opisuje implementaciju jezika FP koji je posredstvom LispKit LISP-a implementiran SECD mašinom. Takva rešenja nisu prirodna i zahtevaju uvodjenje posebnih postupaka. Na primer, u mašinama sa tokom podataka se funkcije višeg reda, mogu implementirati samo uz komplikovane intervencije [Glaser, Hankin, Till, 1984] str. 105.

Elementi podskupa F_L su na primer verzije jezika LISP, Scheme, ISWIM, SASL, KRC, Miranda, Ponder, Orwell, ALFL, ML, LML, Hope, Haskell itd. Elementi skupa F_{FP} su na primer APL, FP, FL itd. Elementi podskupa F_{DF} su: Val, Id, SISAL, Lucid itd. Elementi podskupa F_L prema svojoj brojnosti, raspoloživosti, popularnosti, raširenosti, podržanosti literaturom i uticaju, danas predstavljaju glavnu granu istraživanja u oblasti implementacije čisto-funkcionalnih programskih jezika. O velikom uticaju ovog podskupa jezika svedoči i činjenica da je predloženi standard - jezik Haskell [Hudak, Peyton Jones, Wadler, 1991; 1992] uglavnom zasnovan na karakteristikama jezika iz ove grupe.

1.5 Opšti medjujezici

Istraživanje i definisanje opštih medjujezika je započelo krajem osamdesetih godina sa ciljem da se prevazidju teškoće u daljem razvoju tehnika implementacije funkcionalnih programskih jezika. Zasada u svetu postoji oko desetak medjujezika sa tendencijom da postanu šire upotrebljivi i opšti.

1.5.1 Pregled postojećih medjujezika

Sledi kratak prikaz najpoznatijih opštih medjujezika. Samo neki od sledećih medjujezika su posebno definisani kao opšti, dok su se neki drugi razvili iz internih medjujezika pojedinih istraživačkih timova.

U većini opisa medjujezika u narednom tekstu su prikazani rezultati translacije SASL programa (Sl. 1.3) za izračunavanje 10!.

1.5.1.1 Mašinski jezici apstraktnih mašina

Mašinski jezici apstraktnih mašina su medjujezici jer se nalaze "izmedju" izvornog čisto-funkcionalnog programskog jezika i mašinskog jezika konkretnog računara.

Medjutim jezici apstraktnih mašina ne zadovoljavaju neke od osnovnih kriterijuma za opšti medjujezik: nezavisnost od načina implementacije, na primer.

```
fac 10
  where
    fac 0 = 1
    fac n = n * fac (n-1)
?
```

Sl. 1.3 10! u SASL-u

1.5.1.2 Medjujezici iz literature

Gotovo svaka publikacija posvećena implementaciji funkcionalnih programskih jezika sadrži opis nekog medjujezika koji najčešće, zbog opštosti publikacije, potencijalno može poslužiti kao opšti medjujezik za više funkcionalnih programskih jezika. Medjutim, u svim slučajevima su izloženi medjujezici isključivo podredjeni materijalu prezentiranom u publikaciji i gotovo su beskorisni za praktičnu upotrebu.

Tako na primer, Glaser, Hankin, i Till [1984] str. 71-74, opisuju medjujezik koji služi za implementaciju jezika iz $F^S\_\cap F^T\_\cap F^U\_\cap F^C\_\cap F^X\_$. Field i Harrison [1988] str. 168-171 opisuju opšti medjujezik zasnovan na λ računu koji je proširen veoma malim brojem konstrukcija i tipova podataka. Peyton Jones [1987] takodje opisuje medjujezik zasnovan na λ računu, koji je medjutim neprecizno definisan i sadrži mnoge nepotrebne primitivne funkcije uvedene samo zbog preglednijeg izlaganja u knjizi (na primer "fatbar" operator). Medjutim, ideje izložene u ovoj knjizi su poslužile kasnije za definisanje medjujezika FLIC [Peyton Jones, 1988; Peyton Jones, Joy, 1990], o kome će kasnije biti više reči (odeljak 1.5.1.8).

1.5.1.3 LIF

LIF (skraćeno od *Lambda Intermediate Form*) je medjujezik koji je korišćen za implementaciju jezika ALFL (skraćeno od *A Lazy Functional Language*) [Hudak, 1984]. Kako ALFL pripada preseku $F^S\_\cap F^T\_\cap F^U\_\cap F^C\_+\cap F^X\_+$, to se LIF može koristiti i za implementaciju svih jezika iz istog preseka (SASL na primer).

Prema Young-u [1989] str. 23, funkcionalni program za izračunavanje faktorijela, bi u LIF-u imao zapis prikazan na Sl. 1.4 (oznake u stepenima pojedinih ugradjenih funkcija označavaju njihovu arnost i moraju se navoditi). Kako je LIF nestriktne semantike,

```
fac 10 whererec fac ==
        λn.IF³(=²(n,0),1,*²(n,fac(-²(n,1))))
```

Sl. 1.4 Primer programa u LIF-u

nestriktna semantika SASL programa je izražena implicitno. LIF ne sadrži mogućnosti za implementaciju jezika iz F^S_+.

1.5.1.4 Podskup jezika T

T [Rees, Adams, 1982] je dijalekt jezika Scheme, za koga postoje prevodioci koji proizvode kôd koji se po efikasnosti može meriti sa najefikasnijim imperativnim programskim jezicima [Kranz, 1988]. T i Scheme pripadaju preseku $F^S_+ \cap F^T_- \cap F^U_- \cap F^C_- \cap F^X_-$ i najpogodniji su za implementaciju jezika iz istog preseka (na primer ISWIM [Ivanović, Budimac, 1993]). Uz pomoć ugradjenih funkcija **delay** i **force** se u jeziku T može realizovati i nestriktna semantika [Young, 1989] str. 25-29. Tako se T programima mogu predstaviti programi jezika iz F^S_- (na primer SASL [Budimac, Nikolajević, 1993], ALFL [Young, 1989] itd.), ali ne tako jednostavno kao programi jezika iz F^S_+.

Program za izračunavanje faktorijela bi po Young-u [1989] str. 28-29 u podskupu jezika T (ili Scheme) imao zapis prikazan na Sl. 1.5. Nestriktna semantika SASL programa je u T programu izražena eksplicitno, korišćenjem ugradjenih funkcija **delay**, **force** i **trivial-delay**.

```
(LETREC
 ((D%FAC
   (TRIVIAL-DELAY
     (LAMBDA (D%N)
       (IF (= (FORCE D%N) '0)
           '1
           (* (FORCE D%N (DELAY
                    (FAC (- ((FORCE D%N) '1))))
 ))  ) )  )
(FAC '10)
)
```

Sl. 1.5 Primer programa u T-u

1.5.1.5 IF1

Jezik IF1 [Gurd, Kirkham, Watson, 1985] je opšti medjujezik za jezike iz podskupa F_{DF}. Kao takav se ne može uspešno iskoristiti za jednostavno predstavljanje jezika iz $F_L \cup F_{FP}$.

1.5.1.6 DACTL

Transformacija grafa je ubrzo od jednog od metoda implementacije programskih jezika postala nezavisan pravac istraživanja, te su se ubrzo javili formalizmi za zapisivanje grafova i transformacija nad njima [Kennaway, Klop, Sleep, de Vries, 1991]. Najpoznatiji predstavnici takvih formalizama su DACTL i CLEAN.

DACTL (skraćeno od: *Declarative Alvey Compiler Target Language*) [Glauert, Kennaway, Sleep, Somner, 1988] je "notacija za predstavljanje izračunljivih objekata orijentisanih grafovima i za definisanje izračunavanja koje je izraženo transformacijama grafa" [Glauert, Leth, Thomsen, 1991]. Programi pisani u programskim jezicima koji se mogu implementirati transformacijama grafa (logički, funkcionalni, zasnovani na transformacijama termova, kao i njihove kombinacije) se mogu translirati u odgovarajuće DACTL programe [Glauert, Leth,

Thomsen, 1991], str. 174. Kako se DACTL programima može izraziti veliki broj deklarativnih programskih jezika, DACTL je "previše opšti", a DACTL programi su često nečitki i komplikovani. Kao ilustracija neka posluži DACTL program sa Sl. 1.6 kojim se izračunava vrednost poziva funkcije **f(3)**, ako je **f** definisana kao **f(x)=x** [Glauert, Leth, Thomsen-a, 1991], str. 176.

```
INITIAL => *Read[z], *A[z], z:NewChan;
A[z] -> *X[f], *Put[f Cons[a z]],
        *Const[a 3], f:NewChan, a:NewChan;
X[u:Chan[Cons[Pair[x q]r]]] -> *Z[u x q], u:=Chan[r];
X[u:Chan[Nil]] -> #X[^u];
Z[u x q] -> *X[u], *Y[x q];
Y[x q:Chan[Cons[v r]]] -> *W[x q v], q:=Chan[r];
Y[x q:Chan[Nil]] -> #Y[x q];
W[x q v] -> *Y[x q], *Put[x v];
Const[u:Chan[Cons[v r]] k] -> *C[u k v], u:=Chan[r];
Const[u:Chan[Nil] k] -> #Const[u k];
C[u k v] -> *Const[u k], *Put[v k];
Read[z] => *Get[r], *Put[z r], r:NewChan;
```

Sl. 1.6 Primer programa u DACTL-u

1.5.1.7 CLEAN

CLEAN [Brus, van Eekelen, van Leer, Plasmeijer, 1987; van Eekelen, Nöcker, Plasmeijer, Smetsers, 1990; Nöcker, Smetsers, van Eekelen, Plasmeijer, 1991] je (za razliku od DACTL-a, koji je notacija) programski jezik višeg nivoa. CLEAN pripada $F^S_- \cap F^T_+ \cap F^U_+ \cap F^C_+ \cap F^X_+$ i najpogodniji je za predstavljanje jezika iz istog preseka klasa. CLEAN sadrži mogućnosti za implementaciju jezika iz drugih klasa, ali ga je moguće implementirati jedino na (apstraktnim) mašinama zasnovanim na redukciji grafa. Program za izračunavanje faktorijela u CLEAN-u je prikazan na Sl. 1.7 [Koopman, Smetsers, van Eekelen, Plasmeier, 1991] str. 226.

```
RULE
:: FAC NUM -> NUM          ;  || Definicija tipa funkcije FAC
   FAC 0 -> 1              |  || Graf FAC ---> 0, treba transformisati u čvor 1
   x:FAC n -> * a b        ,  || Graf FAC ---> n, treba transformisati u:
               a: n        ,  ||
               b: x c      ,  || * + ----> n <--------+
               c: - a 1    ;  ||   |                  |
                              ||   +------> FAC ----> - ----> 1
                              ||               X
                              ||                   ----> n
:: Start-> NUM             ;  || Tip početnog grafa
   Start-> FAC 10          ;  || Početni graf
```

Sl. 1.7 Primer programa u CLEAN-u

Nestriktna semantika SASL programa je izražena implicitno.

1.5.1.8 FLIC

FLIC [Peyton Jones, 1988; Peyton Jones, Joy, 1990] (skraćeno od: *Functional Language Intermediate Code*) je nastao od ideja o medjujeziku opisanih u [Peyton Jones, 1987]. FLIC je po svojim osobinama najpogodniji za

implementaciju jezika iz $F^S_- \cap F^T_+ \cap F^U_+ \cap F^C_+ \cap F^X_+$. Posredstvom FLIC-a se mogu implementirati i jezici iz drugih klasa, ali uz odredjeni napor: prema [Peyton Jones, Joy, 1990] str. 2-3, translacija programa pisanih u jezicima iz F^S_+ može biti izuzetno teška. Zapis programa nekog programskog jezika koji nije kompatibilan sa FLIC-om je komplikovan i težak za razumevanje.

Kao primer za poslednju tvrdnju neka posluži FLIC program (prikazan na Sl. 1.8) za izračunavanje izraza 5+6 [Joy, 1989] str. 2. Prikazani program je nečitak jer se po definiciji medjujezika FLIC podrazumeva da je provera tipova podataka izvršena pre translacije u FLIC program. Zbog toga se u FLIC programu istim strukturama podataka predstavljaju različiti tipovi podataka izvornog jezika (pa su tako celi brojevi predstavljeni označenim n-torkama, kao u primeru sa slike).

```
(\x \y PACK-1-3 (INT+ (SEL-1-0 x)
                      (SEL-1-0 y))
)
(PACK-1-3 5)(PACK-1-3 6)
```

Sl. 1.8 5+6 u FLIC-u

Funkcionalni program za izračunavanje faktorijela bi u FLIC-u imao zapis prikazan na Sl. 1.9. Nestriktna semantika SASL programa je u FLIC-u izražena implicitno. FLIC se prema [Peyton Jones, Joy, 1990] koristi u osam istraživačkih centara u svetu. Korišćen je za implementaciju Mirande, SASL-a, čak i FP-a, a implementiran je na više različitih načina. Neka od iskustava su navedena u [Lord, 1987; Joy, 1989; Koopman, Lee, Siewiorek, 1992].

```
&(FAC)(\n IF (POLY= n 0) 1
             (POLY* n (FAC (POLY- n 1)))
      )
FAC 10
```

Sl. 1.9 Primer programa u FLIC-u

1.5.1.9 Ostali medjujezici

Kao medjujezici se ponekad koriste i imperativni programski jezici (na primer C za prevodjenje SML-a [Tarditi, Lee, Acharya, 1992]). Imperativni programski jezici se ne mogu smatrati pogodnim medjujezikom za implementaciju jezika iz F, jer ne mogu na prirodan način podržati sve aspekte funkcionalnog programiranja i nisu nezavisni od načina implementacije.

Čisto-funkcionalni programski jezik Haskell je implementiran kao jezgro (minimalni skup jezika) i biblioteka funkcija realizovanih posredstvom funkcija iz jezgra. Jezgro jezika bi takodje moglo da posluži kao medjujezik za implementaciju drugih čisto-funkcionalnih programskih jezika, ali samo onih koje imaju slične karakteristike kao Haskell.

1.5.2 Rezime analize medjujezika

Na osnovu analize čisto-funkcionalnih programskih jezika i opštih medjujezika za njihovu implementaciju, preciziramo neke od zahteva koje opšti medjujezik treba da ispunjava, na sledeći način:

Zahtev 1: Neka je P program pisan u jeziku $L \in F^{I}_{+}$, $I \in \{S,T,C,U,X\}$ i neka je P' program nastao transliranjem programa P u medjujezik. Neka je P" program nastao transliranjem programa P u medjujezik ako $L \in F^{I}_{-}$. Tada kažemo da opšti medjujezik omogućava podjednako jednostavno predstavljanje jezika iz klasa F^{I}_{+} i F^{I}_{-}, $I \in \{S,T,C,U,X\}$ ukoliko programi P' i P" sadrže pozive istih ugradjenih funkcija medjujezika.

U principu, opšti medjujezik ne može podjednako jednostavno predstaviti jezike iz klasa F^{I}_{+} i F^{I}_{-}, $I \in \{U,X\}$, jer je podela na te klase nastala na osnovu mogućnosti koje jezici poseduju ili ne poseduju. U tom slučaju, medjujezik treba da omogući predstavljanje jezika iz F^{U}_{+} i F^{X}_{+}, a ne podjednako jednostavno predstavljanje jezika iz klasa F^{X}_{+} i F^{X}_{-}; odnosno F^{U}_{+} i F^{U}_{-}.

Opšti medjujezik može podjednako jednostavno predstaviti jezike iz klasa F^{I}_{+} i F^{I}_{-}, $I \in \{S,T,C\}$, jer je podela na te klase nastala na osnovu osobina izraza jezika. U tim slučajevima, medjujezik treba da omogući podjednako jednostavno predstavljanje jezika iz ovih klasa.

Zahtevi 2,3,5 i 6: Jednostavnost translacije izvornog programskog jezika u medjujezik, jednostavnost sintakse medjujezika, izmene u efikasnosti pri prenosu na drugu implementaciju medjujezika i visok nivo medjujezika se teško mogu meriti i uporedjivati. Može se reći da većina razmatranih medjujezika zadovoljava ova četiri uslova ili da njihovo nezadovoljenje ne utiče značajno na opštost medjujezika.

Zahtev 4: Semantiku medjujezika za implementaciju čisto-funkcionalnih programskih jezika je moguće uporedjivati ukoliko je medjujezik takodje čisto-funkcionalan. U tom slučaju je jednostavnost semantike medjujezika obrnuto srazmerna broju ugradjenih funkcija i operatora medjujezika.

Najbitnije osobine analiziranih medjujezika će biti navedene u sledećoj tabeli, osim medjujezika iz publikacija (jer su od malog praktičnog značaja) i mašinskih jezika apstraktnih mašina (jer po definiciji nisu nezavisni od načina implementacije).

Iz podataka prikazanih u tabeli sa Sl. 1.10 se može zaključiti sledeće:

- ni jedan od analiziranih medjujezika ne omogućava podjednako jednostavno predstavljanje nekih klasa čisto-funkcionalnih programskih jezika (na primer FLIC i T ne predstavljaju podjednako jednostavno jezike iz klasa F^{S}_{-} i F^{S}_{+}).
- LIF, T i FLIC su nezavisni od načina sopstvene implementacije.

Pored osobina prikazanih u tabeli, može se još napomenuti da nijedan medjujezik osim CLEAN-a ne podržava modularnost, nijedan medjujezik osim T-a ne podržava nizove (a ni T ih ne podržava na čisto-funkcionalan način), da FLIC, T, DACTL i CLEAN imaju ulazno-izlazne mogućnosti, ali ni jedan na čisto-funkcionalan način. Svi navedeni medjujezici su pogodni za implementaciju i na sekvencijalnim i na paralelnim računarima (zbog osobina čisto-funkcionalnih

	LIF	Podskup jezika T	IF1	DACTL	CLEAN	FLIC
NE zadovoljava zahteve	1	1	1,7	7	1,7	1
Omogućava predstav. podskupova	$F_L \cup F_{FP}$	$F_L \cup F_{FP}$	F_{DF}	F	$F_L \cup F_{FP}$	$F_L \cup F_{FP}$
Omogućava predstav. klasa	$F^S \cap F^T \cap F^U \cap F^X$	$F^T \cap F^U \cap F^X$	?	F	F^C_+	F

Sl. 1.10 Rezime analize postojećih medjujezika

programskih jezika), osim DATCL-a i CLEAN-a kojima treba dodati još nekoliko ugradjenih funkcija za paralelno redukovanje grafa.

Može se takodje zaključiti da je praktično nemoguće imati opšti medjujezik za sva tri uvedena podskupa čisto-funkcionalnih programskih jezika, te da je svaki podskup najbolje predstavljati i implementirati sopstvenim medjujezicima (kao što je to IF1 za podskup F_{DF}). Kako su jezici podskupa F_L trenutno najbrojniji i najuticajniji, definicija medjujezika za F_L je "opravdanija" od definicije medjujezika za druge podskupove. Pored toga, iz tabele sa Sl. 1.10 se vidi da se i jezici iz podskupa F_{FP} mogu prikazati i medjujezicima za jezike iz F_L.

Iz prethodnog odeljka je evidentno da opšti medjujezik podjednako pogodan za predstavljanje većine čisto-funkcionalnih programskih jezika zasnovanih na λ računu i koji je nezavisan od načina implementacije, ne postoji. Na osnovu povećanog obima istraživanja u tom pravcu od 1987. godine, evidentna je takodje i potreba za takvim medjujezikom.

U daljem tekstu se predlaže medjujezik pod nazivom LL, koji ispunjava zahteve za opšti medjujezik, a koji istovremeno sadrži i neke mogućnosti koje analizirani medjujezici ne razmatraju: čisto-funkcionalno ulazno izlazne operacije, čisto-funkcionalni nizovi, pozivanje "stranih" funkcija itd.

Glava 2

Opis medjujezika LL

U ovoj glavi je data definicija medjujezika LL. Sledeći iskustva data u [Lindsey, 1993] str. 122 o pisanju izveštaja i definicija programskih jezika, definicija medjujezika LL je propraćena podesnim primerima i kratkom diskusijom osobina jezika i motivacijom za njihovo uvodjenje. Pri tome se poredjenja najčešće vrše sa medjujezikom FLIC [Peyton Jones, 1988; Peyton Jones, Joy, 1990] koji je najbliži zahtevima za opšti medjujezik za implementaciju čisto-funkcionalnih programskih jezika. Da bi tekst bio što čitljiviji, komentari su pisani manjim slovima i na taj način vizuelno odvojeni od ostalog dela teksta.

2.1 Pregled medjujezika LL

Osnovne karakteristike medjujezika LL su sledeće:

- LL nije (u svim svojim aspektima) strogo definisani programski (medju)jezik, već notacija za zapisivanje najvećeg zajedničkog skupa mogućnosti čisto-funkcionalnih programskih jezika - slično kao što je DACTL notacija, a ne programski jezik strogo definisane sematike. Pojedine konstrukcije medjujezika LL imaju dvostruku definiciju semantike, zavisno od toga da li izvorni jezik pripada klasi F^S_+ ili F^S_-. Precizna definicija semantike tih konstrukcija je sadržana u implementaciji LL-a, tj. u pravilima prevodjenja LL-a u mašinski jezik apstraktne mašine. Tako i jezici iz F^S_+ i jezici iz F^S_- imaju istu reprezentaciju u LL-u.
- LL je po definiciji netipiziran medjujezik, ali se uskladjenost tipova podataka u LL programima ne mora proveravati tokom izvršenja programa. Uskladjenost tipova podataka u programima pisanim na jezicima iz F^T_+ se proverava tokom translacije izvornog programa u medjujezik, a rezultujući mašinski kod bez provere tipova podataka se generiše tokom prevodjenja programa medjujezika u mašinski kod. Uskladjenost tipova podataka u programima pisanim na jezicima iz F^T_- se ne

proverava tokom translacije izvornog programa u medjujezik, a rezultujući mašinski kod sa proverom tipova podataka se generiše tokom prevodjenja programa medjujezika u mašinski kod. Na taj način je provera tipova implementaciona tehnika medjujezika, a ne deo definicije medjujezika. Zbog toga medjujezik LL omogućava isti zapis programa jezika iz F^T_+ i F^T_-.

- LL sadrži anonimnu funkciju, pomoću koje se podjednako jednostavno mogu predstaviti jezici iz F^C_+ i F^C_-, po ekvivalenciji uvedenoj u odeljku 1.1 (strana 4).
- Pomoću poziva ugradjene funkcije _tuple, moguće je predstaviti jezike iz F^U_+. Diskusija se nalazi u odeljku 2.2.4, strana 29, a algoritam translacije u odeljku 3.4, strana 71 (formiranje pomoćnog skupa Cs) i strana 75 (translacija).
- Pomoću poziva ugradjene funkcije _case i odgovarajućih algoritama translacije, moguće je predstaviti jezike iz F^X_+. Algoritmi su dati translacionom funkcijom M u odeljku 3.3, strana 60 i odeljku 3.4, strana 77 (translacija jednačina), odeljku 3.4, strana 73 (translacija "čuvara") i 75 (translacija ZF-izraza).
- LL je zasnovan na λ računu bez tipova, proširenog lokalnim definicijama, konstantama i ugradjenim funkcijama.
- Osnovna unutrašnja struktura LL programa je binarno stablo koje se tekstualno predstavlja (na spoljnim nosiocima podataka) kao s-izraz. Po tome je zapis LL programa sličan zapisu LISP (Scheme ili T) programa. Ovakav zapis je pogodan za mašinsku obradu i razumljiv je čoveku, upravo stoga što podseća na LISP.
- LL sadrži skup ugradjenih funkcija potrebnih za realizaciju karakterističnih osobina i mehanizama jezika iz $F_L \cup F_{FP}$, koji je uvećan za one funkcije čije postojanje povećava efikasnost izvršavanja LL programa.
- LL je jezik višeg nivoa od FLIC-a, a nižeg od CLEAN-a i LIF-a. LL je jezik jednostavnije semantike od većine analiziranih medjujezika.
- LL sadrži (čisto-funkcionalne) nizove kao ugradjeni tip podataka i mehanizme za ostvarivanje modularnosti, čisto-funkcionalnog ulaza/izlaza i za pozivanje "stranih funkcija" (engl. *foreign function interface*).

Funkcionalni program za izračunavanje faktorijela bi u LL-u imao zapis prikazan na Sl. 2.1. U ovom primeru je LL program vrlo sličan sa zapisom odgovarajućeg programa u jeziku Scheme, T ili LispKit LISP.

```
(_letrec (fac (_quote 10))
   (fac (_lambda (n)
        (_if (_eq n (_quote 0))
             (_quote 0)
             (_mul n
                  (fac (_sub n (_quote 1))))
) )    )
```

Sl. 2.1 Program u LL-u

2.2 Medjujezik LL

LL je čisto-funkcionalni medjujezik, te takve njegove osobine neće u daljem tekstu biti naročito isticane. LL je zasnovan na λ računu bez tipova, koji je proširen mogućnostima lokalnih definicija i skupom operatora (ugradjenih funkcija). LL se može posmatrati i kao notacija za zapisivanje λ računa.

Na prvi pogled, LL podseća na čisto-funkcionalne podskupove raznih verzija jezika LISP. LL programi sa LISP programima dele vizuelnu sličnost (oba su oblika s-izraza) i definiciju konstanti primenom funkcije **quote**. Po svim drugim osobinama LL liči ili na neke druge (medju)jezike, ili je originalan.

2.2.1 Leksički elementi medjujezika LL

U ovom odeljku će biti definisani simboli medjujezika i razdvajači simbola (engl. *symbol separators*) medjujezika. Simboli medjujezika će biti podeljeni na posebne simbole, rezervisane reči, identifikatore i brojeve. Razdvajači simbola će biti podeljeni na praznine i komentare.

2.2.1.1 Razdvajači simbola

Razdvajači simbola se mogu nalaziti izmedju simbola medjujezika i ne utiču na značenje programa. Dele se na praznine i komentare. Praznine (često: "bele praznine" - engl. *white spaces*) su znaci: ' ', oznaka za tabulator i oznaka za novi red. Niz znakova izmedju /* i */ je komentar. Komentari mogu biti ugnježdeni.

2.2.1.2 Dozvoljeni znaci u simbolima

Simboli medjujezika se mogu graditi od svih znakova sa grafičkom reprezentacijom (printabilni znaci - engl. *printable characters*). Znaci koji nemaju grafičku reprezentaciju se predstavljaju kontrolnim sekvencama. Spisak svih kontrolnih sekvenci LL-a i njihovog značenja je dat u tabeli na Sl. 2.2.

Većina (funkcionalnih) programskih jezika pored slova i cifara podržava samo deo ostalih znakova. Kako se skupovi dozvoljenih znakova veoma razlikuju, u LL-u se dozvoljava upotreba svih znakova sa grafičkom reprezentacijom. Za predstavljanje ostalih znakova, usvojena je notacija kontrolnih sekvenci FLIC-a.

2.2.1.3 Posebni simboli i rezervisane reči

Posebni simboli medjujezika LL su znaci (,), [i]. Ovi znaci služe za zapisivanje s-izraza medjujezika LL. Rezervisane reči su prikazane na Sl. 2.3.

2.2.1.4 Brojevi

Broj medjujezika LL može biti zapisan kao ceo ili realan broj. Opsezi celog i realnog broja nisu unapred definisani i zavise od implementacije LL-a. Definicija broja je prikazana na Sl. 2.4.

Na primer, sledeći nizovi znakova odvojeni zarezom, su valjani zapisi celih brojeva u LL-u: **11111, -1, 123, 1**, a sledeći nisu: **3.14, =12, sto**. Sledeći nizovi znakova odvojeni zarezom, su valjani zapisi realnih brojeva u LL-u: **11.12, -1.01, 123E-10, 1.12E12**, a sledeći nisu: **314, 12Eabc, stolica, 3.**.

2.2.1.5 Identifikatori

Identifikatori medjujezika LL se grade od niza znakova koji ne počinje brojem i "pružaju" se do posebnog simbola ili razdvajača simbola. Identifikatori LL-a nisu brojevi i rezervisane reči LL-a. Unutar identifikatora se ne mogu nalaziti kontrolne sekvence osim u slučaju kada identifikator nije konstanta. Dužina identifikatora nije ograničena i pravi se razlika izmedju velikih i malih slova.

Kontrolna sekvenca	Značenje	Engleski naziv
#n	novi red	newline
#s	praznina	space
#t	tabulator	tab
#f	nova strana	form feed
#d	brisanje "tekućeg" znaka	delete
#b	brisanje "levog" znaka	backspace
##	znak '#'	
#xddd	znak čiji je kôd ddd (heksadecimalno)	

Sl. 2.2 Definisanje kontrolnih znaka u LL-u

_add	_chr	_from	_member	_readFile
_and	_cons	_if	_mod	_res
_appBChan	_cos	_index	_mul	_rest
_appBFile	_cosH	_lambda	_nil	_select
_appChan	_delay	_le	_not	_seq
_append	_deleteFile	_len	_nth	_sin
_appFile	_div	_leNum	_number	_sinH
_apply	_eq	_leq	_or	_sub
_arcTan	_eqNum	_leqNum	_ord	_success
_arcTanH	_eqStr	_leqStr	_quo	_tag
_array	_error	_leStr	_quote	_tuple
_atom	_exp	_let	_readBChan	_update
_car	_failure	_letrec	_readBFile	_writeBFile
_case	_force	_log	_readChan	_writeFile
_cdr	_foreign			

Sl. 2.3 Rezervisane reči medjujezika LL

Na primer, sledeći nizovi znakova odvojeni zarezom, su (različiti) identifikatori medjujezika LL: **sto, Sto, ljudi->plate, -abc1, $$$**. Sledeći nizovi simbola (odvojeni zarezom) nisu identifikatori medjujezika LL: **123A, ma/*ma, _let, -123**.

Rezervisane reči medjujezika LL ne mogu da budu identifikatori izvornog čisto-funkcionalnog programskog jezika koji se implementira translacijom u LL. Zbog toga rezervisane reči počinju znakom _ što smanjuje čitljivost rezultujućeg LL programa, ali takodje i smanjuje verovatnoću da takvi identifikatori budu izabrani od strane korisnika izvornog funkcionalnog jezika.

2.2.1.6 Logičke vrednosti

Logičke vrednosti *tačno* i *netačno* se u LL-u označavaju posebnim rezervisanim rečima: _true i _false.

2.2.1.7 Atomi

Identifikatori, rezervisane reči i brojevi će se nadalje jednom rečju (po tradiciji programskog jezika LISP) zvati atomima, pri čemu identifikatori i rezervisane reči predstavljaju simboličke atome, a brojevi numeričke atome. Sintaksa atoma medjujezika LL je data na Sl. 2.4.

```
atom            = identifikator ¦ rez_reč ¦ broj

identifikator   = znak {znak+ ¦ cifra} ¦ '-' znak+ {znak+ ¦ cifra}
znak            = svi znaci sa grafičkom reprezentacijom, osim cifara, posebnih simbola
                  i '-' ¦ kont_sekvenca
znak+           = znak ¦ '-'
posebni_simbol= '(' ¦ ')' ¦ '[' ¦ ']'
kont_sekvenca   = '#n' ¦ '#s' ¦ '#t' ¦ '#f' ¦ '#d' ¦ '#b' ¦'##' ¦ '#x' heksa heksa heksa
heksa           = cifra ¦ 'A' ¦ 'B' ¦ 'C' ¦ 'D' ¦'E' ¦ 'F'
cifra           = '0' ¦ '1' ¦ ... ¦ '9'

rez_reč         = videti Sl. 2.3

broj            = celi ¦ realni
celi            = ['-'] cifre
cifre           = cifra {cifra}
realni          = celi '.' cifre ['E' celi] ¦ celi 'E' celi
```

Sl. 2.4 Sintaksa atoma u LL-u

2.2.2 S-izraz

S-izraz (skraćeno od "simbolički izraz" - engl. *symbolic expression*) je struktura podataka poznata iz LISP-a, a koju podržava i LL. Svi izrazi medjujezika LL, njihovi operandi i vrednosti su s-izrazi. S-izraz medjujezika LL je u odnosu na s-izraz poznat iz dijalekata jezika LISP proširen označenim n-torkama i nizovima. U daljem tekstu će se terminom s-izraz označavati s-izraz medjujezika LL. S-izraz se gradi od atoma, posebnih simbola i znaka .. S-izraz je:

- atom,
- lista opšteg oblika $(s_1.s_2)$, pri čemu su s_1 i s_2 s-izrazi (i gde se s_1 često naziva glavom s-izraza, a s_2 repom s-izraza),
- označena n-torka oblika $[t.x_1\ x_2\ \ldots\ x_n]$, pri čemu se t naziva oznakom n-torke, a $x_1,\ldots,x_n$ elementima n-torke, $n \geq 0$.
- niz oblika $[x_1,\ldots,x_n]$ pri čemu su x_1 do x_n elementi niza, $n > 0$.

U zapisu s-izraza postoji konvencija po kojoj se niz znakova .(može izostaviti zajedno sa odgovarajućim). Takodje se niz znakova . _nil može u potpunosti izostaviti (u odeljku 2.2.4 će biti nešto više reči o rezervisanoj reči _nil i njegovoj ulozi u formiranju s-izraza).

S-izraz je fleksibilna struktura podataka, pogodna za predstavljanje LL programa, predstavljanje programa apstraktnih mašina i njihovih registara i predstavljanje grafa.

Na primer, s-izrazi su (odvojeni zarezom): **Dan, (add 2 5), (add (times b c) (quo a d)), ((brat . (ujak . sestra)) (sto stolica)), ([0 . 1 2] . 3)**. Sledeći zapisi s-izraza su ekvivalentni:

((brat . (ujak . sestra)) (sto stolica))	((brat ujak . sestra)(sto stolica))
(1 . (2))	(1 2)
(1 . _nil)	(1)
(1 . (2 . _nil))	(1 2)

Definicija sintakse s-izraza je data na Sl. 2.5.

```
s-izraz  = atom ¦ lista ¦ ntorka ¦ niz
lista    = '(' složen ')'
ntorka   = '[' s-izraz '.' (s-izraz) ']'
niz      = '[' s-izraz {',' s-izraz } ']'
složen   = s-izraz ¦ tačkast ¦ s-izraz složen
tačkast  = s-izraz '.' s-izraz
```

Sl. 2.5 Sintaksa S-izraza

2.2.3 Konstante i funkcije

2.2.3.1 Konstante

Sve konstante medjujezika LL se zapisuju na sledeći način: **(_quote *e*)**, pri čemu je ***e*** s-izraz koji predstavlja konstantu (identifikator, broj, logičku vrednost...). Vrednost izraza **(_quote *e*)** je ***e***.

Na primer, konstante **10**, **_true**, **(1 2 3)** i **otac**, se u LL pišu kao **(_quote 10)**, **(_quote _true)**, **(_quote (1 2 3))** i **(_quote otac)**. Vrednost navedena četiri LL izraza su redom **10**, **_true**, **(1 2 3)** i **otac**.

Umesto načina za zapisivanje konstanti koji je uobičajen u drugim programskim jezicima i medjujezicima koji se karaketriše različitim zapisivanjem konstanti zavisno od njenog tipa, u LL-u je odabran opisani način zbog svoje uniformnosti - prepoznavanje konstanti je jednostavnije na ovaj način, jer je dovoljno ispitati da li se poziva funkcija **_quote**. Potreba za prepoznavanjem konstanti se javlja veoma često tokom translacije izvornog programa (na primer tokom translacije jednačina, str. 60).

2.2.3.2 Primena (poziv) funkcije

Primena funkcije na argumente je s-izraz sledećeg oblika: $(e\ e_1\ e_2\ \ldots\ e_n)$, pri čemu je ***e*** ime funkcije ili s-izraz čija je vrednost funkcija, a e_i, $i=1,\ldots,n$ su s-izrazi i predstavljaju argumente funkcije ***e***.

Na primer, vrednost izraza **(_add (_quote 1) (_quote 3))** je **4** (funkcija **_add** će biti opisana kasnije.)

2.2.3.3 Definisanje funkcije

Korisnička funkcija se definiše anonimnom funkcijom, koja je sledećeg oblika: (**_lambda** (e_1 e_2 ... e_n) e) pri čemu su e_i, i=1,...,n identifikatori, a e je s-izraz. Vrednost anonimne funkcije je funkcija koja za argumente e_i definiše pravilo izračunavanja zadato s-izrazom e. Doseg (engl. *scope*) identifikatora e_i je **_lambda** izraz.

Na primer, vrednost izraza (**_lambda** (x) (**_add** x (**_quote** 1))) je funkcija koja će svom argumentu x dodati jedinicu. Vrednost izraza ((**_lambda** (x) (**_add** x (**_quote** 1))) (**_quote** 1)) je **2**.

Za razliku od FLIC-a, koji dozvoljava samo definisanje funkcije od jednog argumenta, LL dozvoljava definisanje funkcije od više argumenata. Ovo je potrebno zbog implementacije jezika iz F^C, a korisno je i zbog mogućnosti raznih optimizacija (videti stranu 65).

2.2.3.4 Lokalne definicije

Zbog veće čitljivosti LL programa i zbog veće efikasnosti njegovog izvršavanja, u LL-u postoji mogućnost zamene izraza identifikatorima. Ovakve definicije u LL-u (i mnogim drugim funkcionalnim programskim jezicima) imaju oblik tzv. **let** i **letrec** izraza (blokova).

let izraz je sledećeg oblika: (**_let** e (x_1 . e_1) (x_2 . e_2) ... (x_n . e_n)) pri čemu je e s-izraz u kome se pojavljuju identifikatori x_i, a e_i, i=1,...,n, su s-izrazi u kojima se identifikatori x_i ne pojavljuju. Vrednost **let** izraza je vrednost s-izraza e, u procesu čijeg izračunavanja učestvuju s-izrazi e_i koji su u s-izrazu e zamenjeni odgovarajućim identifikatorima x_i.

letrec izraz je sledećeg oblika: (**_letrec** e (x_1 . e_1) (x_2 . e_2) ... (x_n . e_n)) pri čemu je e s-izraz u kome se pojavljuju identifikatori x_i, i=1,...,n, a e_i su s-izrazi u kojima se identifikatori x_i takodje mogu pojavljivati. Vrednost **letrec** izraza je vrednost s-izraza e, u procesu čijeg izračunavanja učestvuju s-izrazi e_i koji su u s-izrazu e zamenjeni odgovarajućim identifikatorima x_i.

Razlika izmedju **let** i **letrec** izraza je samo u dosegu identifikatora x_i. Osobina **letrec** izraza da se x_i smeju pojaviti i unutar e_i omogućava predstavljanje rekurzivnih i medjusobno rekurzivnih definicija unutar **letrec** izraza. Doseg svih identifikatora x_i je **let/letrec** izraz.

Na primer, sledeća dva izraza definišu redom izraze čija je vrednost **2.71** i funkcija za izračunavanje faktorijela.

```
(_let (f (_quote 1) e)                  /* primena funkcije f na stvarne argumente */
    (f . (_lambda (x y) (_mul x y))     /* koja je definisana ovde i zamenjena oznakom f */
    (e . (_quote 2.71))                 /* i pri čemu je e definisano kao 2.71 */
)

(_letrec fac                            /* fac je ime funkcije */
    (fac . (_lambda (x)                 /* koja je definisana na sledeći način */
            (_if (_eq x (_quote 0))
```

```
                (_quote 1)
                (_mul x (fac (_sub x (_quote 1)))))
) ) )
```

Iako je poznato da **let** i **letrec** izrazi nisu neophodni u definiciji nekog funkcionalnog programskog jezika jer se slični efekti mogu postići i anonimnom funkcijom i izrazom λ računa Y, opšte je prihvaćeno mišljenje da postojanje **let** i **letrec** izraza jeste korisno, jer omogućavaju eksplicitno imenovanje zajedničkih podizraza, različite optimizacije i povećavaju efikasnost izvršavanja. Za još neke detalje o prednostima **let** i **letrec** izraza u odnosu na njihove ekvivalente, videti [Peyton Jones, 1987] str. 43; [Peyton Jones, 1988] str. 6.

Mnogi (pogotovu noviji) čisto-funkcionalni programski jezici sadrže samo konstrukcije ekvivalentne sa LL izrazom **letrec**. Bez obzira na to, postojanje **let** izraza je korisno u medjujeziku, jer ga je moguće mnogo jednostavnije i efikasnije implementirati nego **letrec** izraz. Zato bi lokalne definicije izvornog programa trebalo translirati u ekvivalent koji sadrži maksimalni mogući broj **let** izraza. Ova optimizacija se postiže tzv. "analizom zavisnosti" identifikatora (engl. *dependency analisys*) [Peyton Jones, 1987] str. 119. Da se **let** izraz mnogo efikasnije implementira od **letrec** izraza, vidi se na primer iz 4. glave ove teze u kojoj su data pravila prevodjenja oba izraza u jezike različitih apstraktnih mašina.

U medjujeziku FLIC se u **let** i **letrec** izrazima prvo navode svi identifikatori x_i, a potom svi izrazi e_i, i=1,..,n, što je pogodnije za implementaciju FLIC prevodilaca, ali značajno smanjuje čitljivost FLIC programa. Zbog toga su **let** i **letrec** blokovi u LL-u oblika kao u jezicima Scheme i LispKit LISP u kome se identifikator i izraz kojim je on definisan nalaze jedno do drugoga.

LL program je **let** ili **letrec** izraz. Primetimo takodje da u slučaju kada je LL programom definisana funkcija (kao u drugom primeru), argumenti na koje će biti primenjena treba da se zadaju u vreme izvršavanja programa.

2.2.4 Tipovi i konstruktori podataka

U LL-u postoje sledeći prosti tipovi podataka: logički, brojevi, znaci i funkcije. Skupu vrednosti logičkog tipa podataka pripadaju logičke konstante (**_false** i **_true**), skupu vrednosti brojevnog tipa pripadaju realni i celi brojevi, skupu vrednosti znakovnog tipa podataka pripadaju znaci koji postoje na konkretnom računaru i skupu vrednosti tipa funkcije pripadaju korisničke funkcije i ugradjene funkcije LL-a (odgovarajuće operacije tipova podataka će biti uvedene kasnije, u odeljku 2.2.5). Primetimo da se izmedju celih i realnih brojeva ne pravi razlika i da obe vrste brojeva pripadaju istom tipu podataka. U daljem tekstu će se (zbog potrebe jednostavnijeg objašnjavanja semantike ugradjenih funkcija jezika) pretpostaviti da svakom od nabrojanih tipova podataka pripada i specijalna vrednost u oznaci ⊥ (engl. *bottom*) koja označava ili beskonačno izračunavanje ili nedefinisanu vrednost.

Možda se na prvi pogled čini da bi i ⊥ trebalo da postoji kao konstanta medjujezika LL, te da ugradjene funkcije medjujezika u svim nedefinisanim slučajevima, umesto prekida rada programa, dobijaju specijalnu vrednost ⊥. U tom slučaju bi se efikasnost izvršavanja LL programa značajno smanjila, jer bi svaka funkcija pre bilo kakve akcije morala da ispituje jednakost argumenata sa ⊥ [Mitić, 1989]. Pored toga, poznato je ([Stoy, 1977] str. 174) da nije moguće unapred odlučiti da li će se neki izraz beskonačno dugo izračunavati da bi i u tom slučaju funkcija dobila vrednost ⊥. Zbog toga u LL-u ne postoji konstanta ⊥, a implementacija ugradjenih funkcija treba da prekine izvršavanje programa uvek

kada funkcija treba da "izračuna" vrednost $\perp$. Ovakvo rešenje je prihvaćeno i u većini ostalih funkcionalnih (medju)jezika: FLIC, SASL, Haskell itd.

Složeni tipovi podataka medjujezika LL su: liste, n-torke i nizovi. Podaci složenih tipova podataka se grade primenom tzv. konstruktora podataka (engl. *data constructors*). Konstruktori podataka igraju značajnu ulogu u prevodjenju jednačina izvornog jezika. Konstruktori podataka medjujezika LL su **_cons** i **_nil** za gradjenje listi i **_tuple** za gradjenje označenih n-torki. Iako je formalno i funkcija **_array** konstruktor podataka za gradjenje nizova, ona se zbog specifičnosti tipa podataka koga kreira, obično ne smatra konstruktorom u užem smislu. Zbog toga nadalje neće biti smatrana konstruktorom podataka medjujezika LL.

N-torke izvornog jezika i pozivi konstruktora podataka uvedenih tipova podataka izvornog jezika se transliraju u pozive LL konstruktora podataka **_tuple**. To je jedini LL konstruktor podataka koji je dovoljan za prevod bilo kog konstruktora uvedenog tipa podataka izvornog jezika (za diskusiju pogledati sledeći pasus, a za algoritam pogledati translaciju uvedenih tipova podataka Haskell-a, odeljak 4.4). Medjujezik FLIC sadrži više ugradjenih funkcija za predstavljanje različitih vrsta uvedenih tipova podataka izvornog jezika. Postojanje tako velikog broja funkcija se opravdava većom mogućnošću za efikasnu implementaciju medjujezika i potrebom da se razlikuju složeni tipovi podataka nastali "sabiranjem" drugih tipova podataka (engl. *sum types*) od onih nastalih "množenjem" drugih tipova podataka (engl. *product types*). Dosadašnja razmatranja efikasnosti implementacije LL-a i potreba za različitim ugradjenim funkcijama pokazuju da je samo konstruktor **_tuple** dovoljan.

Kako se i konstruktori podataka za listu mogu posmatrati kao uvedeni, postoji mogućnost da se i konstruktori **_cons** i **_nil** predstave posredstvom konstruktora **_tuple** ((_tuple 0 0), odnosno **[0 .]** umesto **_nil** i (_tuple 2 1 x y), odnosno [1 . x y] umesto (_cons x y)). Slično se može reći i za logičke ili znakovne konstante koje se mogu definisatio kao nabrojiv tip podataka i takodje predstaviti označenom n-torkom. Kao posledica prethodnog razmatranja, može se zaključiti da logičke konstante, znakovne konstante kao ni jedan konstruktor podataka osim konstruktora **_tuple** nisu potrebni u LL-u. Medjutim, predstavljanje tipova podataka n-torkom je moguće samo ukoliko se njihova uskladjenost tipova ne mora proveravati tokom izvršavanja programa. U tom slučaju se n-torkama iste strukture mogu predstavljati podaci koji pripadaju različitim tipovima jer su tipovi već provereni i program će uvek naći ono što očekuje (pa se informacija o tipu može potpuno "zaboraviti", jednom kad je proverena). Iz analize funkcionalnih jezika se vidi da se uvodjenje korisničkih tipova podataka i postojanje tipa podataka n-torka, javlja samo u tipiziranim jezicima, te se upravo zbog toga različiti tipovi podataka mogu predstaviti n-torkama (potencijalno) iste strukture*.

Pošto medjujezikom LL treba da se predstavljaju i programi netipiziranih jezika, usvojen je princip po kome u LL-u moraju postojati konstruktori podataka i konstante svih netipiziranih jezika, te da konstruktori LL-a (i logičke i znakovne konstante) ne smeju biti zamenjene n-torkama. Kako je kod netipiziranih jezika jedini složeni tip podataka lista, to su, pored konstruktora **_tuple** u LL-u potrebni jedino još **_cons** i **_nil**.

U FLIC-u se svi konstruktori podataka predstavljaju familijom konstruktora za gradjenje n-torki, jer je FLIC zasnovan na λ-računu bez tipova i podrazumeva da je uskladjenost tipova već proverena. Predstavljanje većine podataka n-torkama je jedan od osnovnih razloga nečitljivosti programa u FLIC-u.

---

* Naravno, moguće je definisati i jezik koji bi dozvoljavao uvodjenje novih tipova podataka, a da pri tome ne bude tipiziran. Takav jezik bi bilo nemoguće (efikasno) implementirati, tako da informacija o tipu bude očuvana i raspoloživa za proveru uskladjenosti tokom izvršavanja programa [Peyton Jones, 1987] str. 190.

2.2.5 Ugradjene funkcije medjujezika LL

Opis svake ugradjene funkcije medjujezika LL će biti istog oblika i sadržavaće definiciju sintakse, statičke semantike, semantike i primere (i inspirisan je opisom iz [Budimac, Ivanović, 1993b]). Statičkom semantikom su definisane vrednosti argumenata i vrednosti primene funkcija i biće dati sintaksnim konstrukcijama definisanim na Sl. 2.4 i Sl. 2.5.

Na primer, definicija sintakse poziva ugradjene funkcije **f** od dva argumenta je: (**f** *s-izraz s-izraz*). Ako je definicija statičke semantike: (**f** *s-izraz identifikator*) ⟶ *s-izraz*, vrednost prvog argumenta je s-izraz, vrednost drugog argumenta identifikator, a vrednost primene ugradjene funkcije na argumente je s-izraz.

Semantika ugradjenih funkcija će biti definisana jednakostima oblika $(f\ x_1\ \ldots\ x_n) = v$ kojima se opisuje vrednost v primene funkcije f na argumente $x_1,\ldots,x_n$ u zavisnosti od vrednosti argumenata. Doseg identifikatora koji učestvuju u jednakosti je ta jednakost. Ukoliko je semantika opisana sa više jednakosti, tada je semantika ugradjene funkcije definisana **i**-tom jednakošću, akko nijedna od prethodnih **j** ($0<j<i$) jednakosti nije primenljiva. Ukoliko ugradjena funkcija ima dve semantike (u zavisnosti da li predstavlja konstrukciju jezika iz klase F^s_+ ili F^s_-), dodatne jednakosti definišu striktnu semantiku - na taj način je definicija nestriktne semantika deo definicije striktne semantike. U jednakostima se vrednosti primene ugradjenih funkcija na argumente objašnjavaju matematičkim ili već definisanim relacijama i zakonitostima. U opisu semantike nekih ugradjenih funkcija koriste se i sledeće oznake, koje se ovde objašnjavaju intuitivno*: $e[p_1/x_1, \ldots, p_n/x_n]$ označava izraz **e** u kojem su sve pojave identifikatora x_i zamenjene izrazom p_i, $1 \le i \le n$, $n>0$; **Y** je standardni zatvoreni izraz λ računa pogodan za realizaciju rekurzije, **λx.e** označava (anonimnu) funkciju čiji je argument **x**, a telo **e**.

Na primer, semantika ugradjene funkcije _add će biti opisana jednakošću (_add a b) = a+b, pri čemu se sa leve strane znaka = nalazi primena funkcije na argumente, a sa desne (u ovom slučaju) matematički izraz.

U primerima će se podrazumevati vrednosti argumenata.

Na primer, kao ilustracija primene ugradjene funkcije _add će se navoditi primer oblika **(_add 1 3) -> 4**, umesto (na primer) (_add (_quote 1) (_quote 3)) -> 4.

Sve četiri vrste opisa će biti označene na slikama sledećim oznakama: **Si** (sintaksa), **SS** (statička semantika), **Se** (semantika) i **Pr** (primer). Ukoliko ugradjena funkcija ima i striktnu semantiku, dodatne jednakosti će biti označene sa **Se(S)**.

---

* Za formalne definicije pogledati literaturu o λ računu.

2.2.5.1 quote, lambda, let, letrec

Kompletnosti radi, navodimo opise uvedenih izraza i ugradjenih funkcija medjujezika LL, bez posebnih komentara (Sl. 2.6 - Sl. 2.9).

```
Si:  '(' '_quote' s-izraz ')'

SS:  (_quote s_izraz)              -→ s-izraz

Se:  (_quote e)                    = e

Pr:  (_quote 123)                  -→ 123
     (_quote (a b c (d) e))        -→ (a b c (d) e)
```

Sl. 2.6 _quote: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_lambda' '(' identifikator {identifikator} ')' s-izraz ')'

SS:  (_lambda (identifikator ... identifikator) s-izraz) -→ funkcija

Se:  (_lambda (x1 ... xn) e)       = λ(x1...xn).e
     (_lambda (x1 ... xn) ⊥)       = ⊥

Pr:  ((_lambda (x) (_add x (_quote 1))) (_quote 2)) -→ 3
     ((_lambda (x) (_add x (_quote 1))) (_quote 5)) -→ 6
     ((_lambda (x) (_mul x x)) (_quote 5))          -→ 25
```

Sl. 2.7 _lambda: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_let' s-izraz '('identifikator '.' s-izraz ')'
                 { '('identifikator '.' s-izraz ')' } ')'

SS:  (_let s-izraz (identifikator . s-izraz) ... (identifikator . s-izraz)) -→ s-izraz

Se:   (_let e (x1 . e1) ... (xn . en))                    = e[e1/x1,...,en/xn]
      (_let ⊥ (x1 . e1) ... (xn . en))                    = ⊥
Se(S):(_let e (x1 . e1) ... (xi . ⊥) ... (xn . en))       = ⊥,  1 ≤ i ≤ n

Pr:  (_let (_add x y)
           (x . (_quote 1))
           (y . (_quote 2)))     -→ 3

     (_let (ad1 x)
           (ad1 . (_lambda (x) (_add x (_quote 1))))
           (x . (_quote 2)))     -→ 3
```

Sl. 2.8 _let: sintaksa, statička semantika, semantika i primeri

2.2.5.2 Konstruktori podataka

Vrednost primene ugradjene funkcije **_cons** (Sl. 2.10) na argumente je lista, koju po konvenciji pišemo kao (*x* . *y*), pri čemu je *x* vrednost prvog argumenta

```
Si:  '(' '_letrec' s-izraz '('identifikator '.' s-izraz ')'
                  { '('identifikator '.' s-izraz ')' } ')'

SS:  (_letrec s-izraz (identifikator . s-izraz) ...(identifikator . s-izraz)) -→ s-izraz

Se:   (_letrec e (x1 . e1) ... (xn . en))                    = Y(λ(e1,...,en).(e1,...,en))
      (_letrec ⊥ (x1 . e1) ... (xn . en))                    = ⊥
Se(S):(_letrec e (x1 . e1) ... (xi . ⊥) ... (xn . en))       = ⊥,  1 ≤ i ≤ n

Pr:  (_letrec (fib (_quote 2))
              (fib . (_lambda (x)
                        (_if (_leq x (_quote 2))
                             (_quote 1)
                             (_add (fib (sub1 x)) (sub2 x))))
              )      )
              (sub1 . (_lambda (x) (_sub x (_quote 1))))
              (sub2 . (_lambda (x) (sub1 (sub1 x))))
     )                          -→ 1
```

Sl. 2.9 _letrec: sintaksa, statička semantika, semantika i primeri

funkcije **_cons**, a *y* vrednost drugog argumenta funkcije **_cons**. Vrednost primene ugradjene funkcije **_nil** (Sl. 2.11) je "prazna" lista, koju po konvenciji pišemo kao **nil**. Vrednost primene ugradjene funkcije **_tuple** (Sl. 2.12) je označena n-torka, koju po konvenciji pišemo kao $[t \,.\, x_1 \ldots x_n]$, pri čemu je *t* vrednost drugog argumenta funkcije **_tuple** (oznaka, engl. *tag*), x_i, i=1,...,n je vrednost i+2-gog argumenta funkcije **_tuple**, a *n* je vrednost prvog argumenta funkcije **_tuple**.

```
Si:  '(' '_cons' s-izraz s-izraz ')'

SS:  (_cons s-izraz s-izraz)              -→ lista

Se:   (_cons s t)                         = (s . t)
Se(S):(_cons ⊥ t)                         = ⊥
Se(S):(_cons s ⊥)                         = ⊥

Pr:  (_cons x 125)                              -→ (x . 125)
     (_cons danas (je lep dan)) -→ (danas . (je lep dan)) -→ (danas je lep dan)
```

Sl. 2.10 _cons: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_nil' ')'

SS:  (_nil)                    -→ s-izraz

Se:  (_nil)                    = nil

Pr:  (_cons 1 _nil)            -→ (1 . nil)          -→ (1)
     (_cons 1 (_cons 2 _nil))  -→ (1 . (2. nil))     -→ (1 2)
```

Sl. 2.11 _nil: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_tuple' s-izraz {s-izraz} ')'

SS:  (_tuple cifre atom s-izraz ... s-izraz)     -→ ntorka

Se:   (_tuple n t x1 ... xn)                  = [t . x1 ... xn]
      (_tuple ⊥ t x1 ... xn)                  = ⊥
Se(S):(_tuple n ⊥ x1 ... xn)                  = ⊥
      (_tuple n t x1 ... ⊥ ... xn)            = ⊥

Pr:  (_tuple 2 0 1 1)              -→ [0 . 1 1]
     (_tuple 3 1 mama tata sin) -→ [1 . mama tata sin]
```

Sl. 2.12 _tuple: sintaksa, statička semantika, semantika i primeri

2.2.5.3 Ugradjene funkcije za rad sa listama

U ovom odeljku su definisane ugradjene funkcije čiji bar jedan argument ima vrednost liste (Sl. 2.13 - Sl. 2.19).

```
Si:  '(' '_car' s-izraz ')'

SS:  (_car lista)                  -→ s-izraz

Se:  (_car (_cons x y))            = x
     (_car x)                      = ⊥ , inače

Pr:  (_car (a.b))                        -→ a
     (_car ((1) 2 3 4))                  -→ (1)
     (_car (1 ((2) 3 ((4)) 4)))          -→ 1
```

Sl. 2.13 _car: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_cdr' s-izraz ')'

SS:  (_cdr lista)                  -→ s-izraz

Se:  (_cdr (_cons x y))            = y
     (_cdr x)                      = ⊥ , inače

Pr:  (_cdr (a.b))                        -→ b
     (_cdr ((1) 2 3 4))                  -→ (2 3 4)
     (_cdr (1 ((2) 3 ((4)) 4)))          -→ (((2) 3 ((4)) 4))
```

Sl. 2.14 _cdr: sintaksa, statička semantika, semantika i primeri

U drugim medjujezicima se funkcije analogne ugradjenim LL funkcijama **_append**, **_member**, **_len** itd. obično realizuju kao bibliotečke funkcije. U LL su dodate zbog efikasnosti i zbog toga što realizuju često primenjivane operacije u funkcionalnim programima. Upravo zbog efikasnosti se preporučuje da se ove funkcije realizuju posebnim naredbama apstraktnih mašina, umesto da se realizuju prevodom u sekvence već postojećih naredbi.

Ugradjena funkcija za generisanje listi koje sadrže sve brojeve iz zadatog intervala (analogon Haskell-ovom **[n..m]**) ne postoji u LL-u, jer ga nije moguće implementirati tako da bude nezavisan od semantike izvornog jezika. Na primer, svi elementi liste izmedju 1 i 10000 bi u striktnom jeziku trebali da se generišu odmah, a u nestriktnom tek kada su potrebni u izračunavanju.

```
Si:  '(' '_append' s-izraz  s-izraz ')'

SS:  (_append lista s-izraz)                -→ s-izraz

Se:   (_append _nil s)                      = s
      (_append (_cons x s) t)               = (_cons x (_append s t))
      (_append x y)                         = ⊥ , inače
Se(S):(_append x ⊥)                         = ⊥

Pr:  (_append (a) (b))                      -→ (a b)
     (_append ((1) 2 3 4) (a))              -→ ((1) 2 3 4 a)
     (_append (a b) c)                      -→ (a b.c)
```

Sl. 2.15 _append: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_len' s-izraz ')'

SS:  (_len lista)                           -→ cifre

Se:  (_len _nil)                            = 0
     (_len (_cons x s))                     = (_add (_quote 1) (_len s))
     (_len x)                               = ⊥ , inače
Pr:  (_len (a))                             -→ 1
     (_len ((1) 2 (3 4)))                   -→ 3
     (_len nil)                             -→ 0
```

Sl. 2.16 _len: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_member' s-izraz s-izraz ')'

SS:  (_member s-izraz lista)     -→ logički

Se:   (_member x _nil)           = _false
      (_member x (_cons x s))    = _true
      (_member x (_cons y s))    = (_member x s), x≠y
      (_member x y)              = ⊥, inače
Se(S):(_member x ⊥)              = ⊥

Pr:  (_member a (b a c))         -→ _true
     (_member (a) (b a c))       -→ _false
     (_member (a) (b (a) c))     -→ _true
```

Sl. 2.17 _member: sintaksa, statička semantika, semantika i primeri

2.2.5.4 Ugradjene funkcije za rad sa n-torkama

U ovom odeljku su definisane ugradjene funkcije čiji bar jedan argument ima vrednost n-torke (Sl. 2.20 - Sl. 2.21).

2.2.5.5 Ugradjene funkcije za rad sa nizovima

U ovom odeljku su definisane ugradjene funkcije čiji bar jedan argument ima vrednost niza. Definisana je i ugradjena funkcija **_array** čija je vrednost niz (Sl. 2.22 - Sl. 2.24).

LL poseduje dve ugradjene funkcije za definisanje niza (**_array** i **_update**), što je posledica činjenice da su u čisto-funkcionalnim programskim jezicima definisanju nizova pristupa na dva različita

```
Si: '(' '_nth' s-izraz s-izraz ')'

SS: (_nth lista cifre)                    -→ s-izraz

Se:  (_nth (_cons x s) (_quote 1))        = x
     (_nth (_cons x s) n)                 = (_nth s (_sub n (_quote 1)))
     (_nth x ⊥)                           = ⊥
     (_nth x n)                           = ⊥, inače

Pr: (_nth (b a c) 1)            -→ b
    (_nth (b (a) c) 2)          -→ (a)
```

Sl. 2.18 _nth: sintaksa, statička semantika, semantika i primeri

```
Si: '(' '_rest' s-izraz s-izraz ')'

SS: (_rest lista cifre)                   -→ s-izraz

Se: (_rest (_cons x s) (_quote 1))        = s
    (_rest (_cons x s) n)                 = (_rest s (_sub n (_quote 1)))
    (_rest x ⊥)                           = ⊥
    (_rest x n)                           = ⊥ , inače

Pr: (_rest (b a c) 1)                     -→ (a c)
    (_rest (b (a) c) 2)                   -→ (c)
```

Sl. 2.19 _rest: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_tag' s-izraz ')'

SS:  (_tag ntorka)                        -→ atom

Si:  (_tag (_tuple n t x1 ... xn))        = t
     (_tag x)                             = ⊥ , inače

Pr:  (_tag (_tuple 3 2 1 2))              -→ 2
```

Sl. 2.20 _tag: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_select' s-izraz  s-izraz ')'

SS:  (_select ntorka celi)                -→ s-izraz

Se:  (_select (tuple n t x1 ... xn) i)    = xi  , 1 ≤ i ≤ n
     (_select x i)                        = ⊥, inače

Pr:  (_select [2 . a b] 2)                -→ b
     (_select [1 . 2 3] 1)                -→ 2
```

Sl. 2.21 _select: sintaksa, statička semantika, semantika i primeri

načina (monolitni i inkrementalni) [Hudak, 1986]. Ugradjena funkcija **_array** podržava monolitni pristup nizovima (niz se samo jednom kreira a više puta koristi), dok ugradjena funkcija **_update** podržava inkrementalni pristup, po kome se niz može menjati bilo kada tokom izvršavanja programa. Rezultat primene funkcije **_update** treba da bude novi niz, zbog potrebe da se održi čisto-funkcionalna priroda medjujezika. Kako takva realizacija ugradjene funkcije **_update** može da bude vrlo neefikasna, potrebno je pre kreiranja novog niza proveriti da li je moguće promeniti originalni niz "u mestu".

```
Si:  '(' '_array' s-izraz  s-izraz ')'

SS:  (_array cifre ((cifre . s-izraz) ... (cifre . s-izraz)))     -→ niz

Se:   (_array n ((1 . x1) ... (n . xn))))                        = [x1,...,xn]
      (_array n ((1 . x1) ... (⊥ . xi) ... (n . xn)))            = ⊥ , 1 ≤ i ≤ n
      (_array ⊥ x)                                               = ⊥
Se(S):(_array n ((1 . x1) ... (i . ⊥) ... (n . xn)))             = ⊥ , 1 ≤ i ≤ n

Pr:  (_array 3 ((1 . 3) (2 . 2) (3 . 1)))          -→ [3,2,1]
     (_array 1 ((1 . 5)))                          -→ [5]
```

Sl. 2.22 _array: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_update' s-izraz s-izraz s-izraz ')'

SS:  (_update niz cifre s-izraz))                                -→ niz

Se:   (_update (_array n ((1 . x1) ... (n . xn))) i x'i)          = [x1,...x'i,...,xn] , 1≤i≤n
      (_update (_array n ((1 . x1) ... (⊥ . xi) ... (n . xn)) i x'i) = ⊥ , 1≤i≤n
      (_update a ⊥ x)                                             = ⊥
Se(S):(_update (_array n ((1 . x1) ... (i . ⊥) ... (n . xn)) i x'i)  = ⊥ , 1≤i≤n
      (_update a i ⊥)                                             = ⊥

Pr:  (_update (_array 3 ((1 . 3) (2 . 2) (3 . 1))) 2 5)          -→ [3,5,1]
     (_update (_array 1 ((1 . 5))) 1 2)                          -→ [2]
```

Sl. 2.23 _update: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_index' s-izraz s-izraz ')'

SS:  (_index niz cifre)                                          -→ s-izraz

Se:  (_index (_array n ((1 . x1) ... (n . xn))) i)               = xi  , 1≤i≤n
     (_index a i)                                                = ⊥   , inače

Pr:  (_index (_array 3 ((1 . 3) (2 . 2) (3 . 1))) 3)             -→ 1
     (_index (_array 1 ((1 . 5))) 1)                             -→ 5
```

Sl. 2.24 _index: sintaksa, statička semantika, semantika i primeri

Strogo gledano, ugradjena funkcija **_array** nije neophodna i može se realizovati posredstvom jednostavnije funkcije kojom bi se kreirao "prazan" niz, koji bi se zatim popunio primenom kompozicije funkcije **_update**. Medjutim, kako je u realizaciju funkcije **_update** "ugradjena" potreba za optimizacijom i izbegavanjem nepotrebnog kopiranja (a što može da se realizuje na različite načine), to bi se realizacija funkcije **_array** posredstvom funkcije **_update** potencijalno veoma razlikovala po svojoj efikasnosti izmedju različitih implementacija LL-a, što je u suprotnosti sa jednim od zahteva za opšti medjujezik. Zbog toga **_array** postoji kao ugradjena funkcija.

Iz definicija semantike ugradjenih funkcija **_array** i **_update** se vidi da su njihove primene nedefinisane, ako je bar jedan od indeksa niza nedefinisan. Po toj osobini su nizovi analogni n-torkama. Drugačija definicija nizova je moguća, ali zasada nije poznata u praksi i značajno bi komplikovala implementaciju.

2.2.5.6 Ugradjene funkcije za rad sa znakovima

U ovom odeljku su definisane ugradjene funkcije **_chr** i **_ord** (Sl. 2.25).

```
Si:  '(' '_chr' s-izraz ')'
     '(' '_ord' s-izraz ')'

SS:  (_chr celi)                  -→ (znak+ ¦ cifra)
     (_ord (znak+ ¦ cifra) )      -→ celi

Se:  (_ord (_chr n))              = n
     (_chr (_ord x))              = x
     (_ord ⊥)                     = ⊥
     (_chr ⊥)                     = ⊥

Pr:  (_chr 32)                    -→ ' '
     (_ord #s)                    -→ 32
```

Sl. 2.25 _chr i **_ord**: sintaksa, statička semantika, semantika i primeri

2.2.5.7 Numeričke ugradjene funkcije

U ovom odeljku su definisane ugradjene funkcije čiji je bar jedan argument broj. Vrednosti primene ovih funkcija na argumente su takodje brojevi. Vrednost primene funkcije **_add** na argumente je zbir vrednosti argumenata, funkcije **_mul** je proizvod vrednosti argumenata, funkcije **_sub** je razlika vrednosti argumenata, funkcije **_quo** je količnik vrednosti argumenata, funkcije **_div** je količnik celog deljenja vrednosti njegovih argumenata i funkcije **_mod** je ostatak celog deljenja vrednosti argumenata. Sve nabrojane ugradjene funkcije imaju po dva argumenta. Vrednosti argumenata ugradjenih funkcija **_div** i **_mod** su celi brojevi. Zbog sličnosti u definicijama ovih funkcija i zbog njihove opšte prihvaćenosti, data je definicija samo funkcije **_add** (Sl. 2.26).

```
Si:  '(' '_add' s-izraz s-izraz ')'

SS:  (_add broj broj)             -→ broj

Se:  (_add x y)                   = x+y
     (_add ⊥ x)                   = ⊥
     (_add x ⊥)                   = ⊥

Pr:  (_add 2 2)                   -→ 4
     (_add 3.14 1)                -→ 4.14
     (_add 3.52 0.48)             -→ 4
```

Sl. 2.26 _add: sintaksa, statička semantika, semantika i primeri

Vrednost primene ugradjene funkcije **_sin** na argument je sinus vrednosti argumenta, funkcije **_cos** je kosinus vrednosti argumenta, funkcije **_sinH** je hiperbolični sinus vrednosti argumenta, funkcije **_cosH** je hiperbolični kosinus vrednosti argumenta, funkcije **_arcTan** je arkus tangens vrednosti argumenta, funkcije **_arcTanH** je hiperbolični arkus tangens vrednosti argumenata, funkcije **_log** je prirodni logaritam

vrednosti argumenta i funkcije _exp je eksponent vrednosti argumenta. Sve navedene funkcije imaju po jedan argument čija je vrednost brojevnog tipa. Zbog medjusobne sličnosti je dat opis samo jedne od ovih funkcija - _sin (Sl. 2.27).

```
Si:  '(' '_sin' s-izraz ')'

SS:  (_sin broj)          -→ broj

Se:  (_sin x)             = sin(x)
     (_sin ⊥)             = ⊥
```

Sl. 2.27 _sin: sintaksa, statička semantika, semantika i primeri

FLIC takodje sadrži sličan skup realnih funkcija čiji argumenti imaju vrednost (realnih) brojeva. Ostale se funkcije mogu izraziti posredstvom navedenih.

2.2.5.8 Relacijske ugradjene funkcije

U ovom odeljku su definisane relacijske ugradjene funkcije medjujezika LL (Sl. 2.28 - Sl. 2.30).

```
Si: '(' '_eq' s-izraz s-izraz ')'

SS: (_eq s-izraz s-izraz)          -→ logički

Se: (_eq n m)                      = n=m, n i m su brojevi
    (_eq a b)                      = a=x1x2...xi, b=y1y2...yj,
                                     ako i=j i xk=yk, k=1,...,i, a i b su identifikatori
    (_eq _true _true)                      = _true
    (_eq _false _false)                    = _true
    (_eq (_cons s t) (_cons r u))          = (_eq s r) ∧ (_eq t u)
    (_eq (_tuple n t x1 ... xn)
         (_tuple m s y1 ... ym))           = (_eq n m) ∧ (_eq t s) ∧
                                             (_eq x1 y1) ∧ ... ∧ (_eq xn ym)
    (_eq (_array n (1 .x1) ... (n .xn))
         (_array m (1 .y1) ... (m .ym))) = (_eq n m) ∧
                                             (_eq x1 y1) ∧ ... ∧ (_eq xn ym)
    (_eq x ⊥)                              = ⊥
    (_eq ⊥ x)                              = ⊥
    (_eq x y)                              = _false, inače

Pr: (_eq abc abc)                  -→ _true
    (_eq a 1)                      -→ _false
    (_eq (1 2 3) (1 2 a))          -→ _false
```

Sl. 2.28 _eq: sintaksa, statička semantika, semantika i primeri

Opisane ugradjene funkcije ispituju relacije izmedju vrednosti argumenata različitih tipova podataka (engl. *overloaded operator*). Ovakve ugradjene funkcije su potrebne u medjujeziku zbog implementacije netipiziranih čisto-funkcionalnih programskih jezika.

Iako su za realizaciju uobičajenih relacijskih operatora izvornih (čisto-funkcionalnih) programskih jezika ($=$, $\leq$, $<$, $\neq$, $\geq$, $>$) dovoljne dve ugradjene relacijske funkcije LL-a (na primer _eq i _le) i logička ugradjena funkcija _not, u definiciju LL-a je dodata i treća relacijska ugradjena funkcija da bi se održala čitljivost programa u LL-u. Zbog veće efikasnosti, preporučuje se i neposredna implementacija dodatne ugradjene funkcije

```
Si: '(' '_le' s-izraz s-izraz ')'

SS: (_le atom atom)     -→ logički

Se: (_le n m)   = n<m, n i m su brojevi
    (_le a b)   = _true, a=x1x2...xn, b=y1y2...ym,
                  ako (∃k 1≤k≤min(n,m), xi=yi, i=1,...,k-1 & xk<yk) ili (n<m, xi=yi,
                  i=1,...,n), a i b su identifikatori
    (_le x ⊥)       = ⊥
    (_le ⊥ x)       = ⊥
    (_le x y)       = _false, u svim ostalim slučajevima

Pr: (_le 12 13)         -→ _true
    (_le abc abc)       -→ _false
    (_le abc abd)       -→ _true
```

Sl. 2.29 _le: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_leq' s-izraz s-izraz ')'

SS:  (_leq atom atom)   -→ logički

Se:  (_leq x y)                 = (_le x y) ∨ (_eq x y)

Pr:  (_leq 12 13)               -→ _true
     (_leq abc abc)             -→ _true
```

Sl. 2.30 _leq: sintaksa, statička semantika, semantika i primeri

u jezicima apstraktnih mašina, umesto njeno izražavanje posredstvom postojećih. Ugradjene funkcije za utvrdjivanje relacija $\neq$, $\geq$ i $>$ nisu predvidjena u LL-u, jer se oni jednostavno mogu izraziti preko postojeća tri (negacijom, odnosno zamenom redosleda argumenata), a da se ne naruši čitljivost rezultujućeg programa.

Ugradjene funkcije **_eqNum**, **_leNum** i **_leqNum** su redom analogne funkcijama **_eq**, **_le** i **_leq**, ali je vrednost njihovih argumenata brojevna. Ugradjene funkcije **_eqStr**, **_leStr** i **_leqStr** su redom analogne funkcijama **_eq**, **_le** i **_leq**, ali su vrednosti njihovih argumenata identifikatori.

Posebne relacijske ugradjene funkcije za proste tipove podataka su potrebne u medjujeziku za implementaciju tipiziranih jezika. Translator izvornog jezika u medjujezik treba da, na osnovu tipova argumenata generiše poziv odgovarajuće ugradjene funkcije medjujezika, da bi se očuvala efikasnost izvršavanja programa tipiziranih jezika.

2.2.5.9 Logičke ugradjene funkcije

U ovom odeljku su definisane logičke ugradjene funkcije medjujezika LL (Sl. 2.31 - Sl. 2.33).

U LL-u je prihvaćena nestriktna semantika ugradjenih funkcija **_and** i **_or**. Pošto čak i mnogi imperativni programski jezici (koji su po pravilu striktni) implementiranju logičke operatore na isti, "lenji" način (C i Modula-2, na primer), odabrana semantika se može smatrati prednošću, a ne nedostatkom. Ako, medjutim, ipak postoji potreba da se u medjujeziku izraze striktni logički operatori

```
Si:  '(' '_and' s-izraz s-izraz ')'

SS:  (_and logički logički)      -→ logički

Se:  (_and _false x)             = _false
     (_and _true x)              = x
     (_and ⊥ x)                  = ⊥

Pr:  (_and _true _false)         -→ _false
     (_and _true _true)          -→ _true
```

Sl. 2.31 _and: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_or' s-izraz s-izraz ')'

SS:  (_or logički logički)       -→ logički

Se:  (_or _false x)                    = x
     (_or _true x)                     = _true
     (_or ⊥ x)                         = ⊥

Pr:  (_or _true _false)          -→ _true
     (_or _false _false)         -→ _false
```

Sl. 2.32 _or: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_not' s-izraz ')'

SS:  (_not logički)      -→ logički

Se:  (_not _false)       = _true
     (_not _true)        = _false
     (_not ⊥)            = ⊥
```

Sl. 2.33 _not: sintaksa, statička semantika i semantika

nekih izvornih (čisto-funkcionalnih) programskih jezika, tada se to može učiniti pozivom ugradjene funkcije _seq (videti translaciju odgovarajućih LispKit LISP funkcija na strani 50).

2.2.5.10 Uslovni izrazi

U ovom odeljku su definisane uslovne ugradjene funkcije (uslovni izrazi) medjujezika LL (Sl. 2.34 - Sl. 2.35). Vrednost primene uslovnih izraza na argumente je vrednost jednog od argumenata uslovnog izraza.

```
Si:  '(' '_if' s-izraz s-izraz s-izraz ')'

SS:  (_if logički s-izraz s-izraz)    -→ s-izraz

Se:  (_if _true x y)     = x
     (_if _false x y)    = y
     (_if ⊥ x y)         = ⊥
```

Sl. 2.34 _if: sintaksa, statička semantika i semantika

```
Si: '(' '_case' s-izraz '('_nil  '.' s-izraz ')'
                        '('_cons '.' s-izraz ')' ')'

Si: '(' '_case' s-izraz '(' '0' '.' s-izraz ')'
                        '(' '1' '.' s-izraz ')'
                            . . .
                        '(' 'r' '.' s-izraz ')' ')'

                r je maksimalni broj oznaka za n-torku koja je prvi argument
                operatora. Izmedju 0 i r se nalaze sve oznake n-torke, redom.

SS: (_case s-izraz (atom . s-izraz) {(atom . s-izraz)} )          -→ s-izraz

Se: (_case _nil (_nil . e0) (_cons . e1))                         = e0
    (_case (_cons x y) (_nil . e0) (_cons . e1))                  = e1
    (_case (_tuple n k x1 ... xn) (0 . e0) (1 . e1) ... (r . er)) = ek, 0 ≤ k ≤ r
    (_case x (_nil . e0) (_cons . e1))                            = ⊥ , inače
```

Sl. 2.35 _case: sintaksa, statička semantika i semantika

Semantika uslovnog izraza _if je nestriktna jer se izračunava drugi ili treći argument u zavisnosti od vrednosti prvog argumenta. Ovakva semantika _if-a je neophodna pri realizaciji i striktnih i nestriktnih čisto-funkcionalnih programskih jezika. Ako bi se svi argumenti (u striktnim jezicima) uslovnog izraza _if izračunavali pre primene funkcije _if, u često bi dolazilo do beskonačnog izračunavanja.

Vrednost prvog argumenta uslovnog izraza _case može biti lista i označena n-torka. Lista se posebno javlja u definiciji uslovnog izraza _case da bi se i netipizirani jezici mogli prevesti u LL, a da program ostane čitak. Naglasimo da uslovni izraz _case "selektuje" jedan od svojih argumenata samo na osnovu strukture svog prvog argumenta. Zbog toga se prvi argument funkcije ne izračunava u potpunosti.

Peyton Jones [1987] str. 74 predlaže operator case koji bi pored oznake (_nil, _cons ili 0 - r) sadržavao i onoliko identifikatora kolika je arnost konstruktora podataka. Tokom izvršavanja programa se identifikatori vezuju za odredjene elemente složene strukture podataka. U zavisnosti od semantike izvornog jezika (striktna ili nestriktna) identifikatori treba da se vezuju odmah po selekovanju odredjene "grane" case izraza (striktna semantika - "vredno prepoznavanje") ili tek kada su neophodne u izračunavanju (nestriktna semantika - "lenjo prepoznavanje"). Ovakvo rešenje ne može biti usvojeno u opštem medjujeziku kakav je LL, jer postoje jezici (Haskell, na primer) koji podržavaju i vredno i lenjo prepoznavanje i vezivanje identifikatora, te bi u medjujeziku morale postojati dve vrste case izraza. Usvojena je prikazana definicija uslovnog izraza _case, po kojoj je odluka o trenutku vezivanja identifikatora za elemente strukture podataka prepuštena translatoru izvornog čisto-funkcionalnog programskog jezika - izrazi kojim se definišu vrednosti identifikatora se "smeštaju" u izraze e_i. Mogući načini vezivanja identifikatora se mogu videti u definiciji funkcije M za translaciju Haskell-a (odeljak 3.4).

2.2.5.11 Predikati

U ovom odeljku su definisani predikati medjujezika LL (Sl. 2.36 - Sl. 2.37). Vrednost primene predikata na argumente je logička vrednost.

Predikati kojima se utvrdjuje da li je vrednost argumenta označena ntorka nisu potrebni u medjujeziku jer se taj tip podataka javlja samo u tipiziranim jezicima i može se prepoznati uslovnim izrazom _case.

2.2.5.12 Zadržano i forsirano izračunavanje (engl. *delayed, forced evaluation*)

U ovom odeljku su definisane ugradjene funkcije za definisanje i korišćenje struktura podataka kojima se ostvaruje tzv. "zadržano izračunavanje" (Sl. 2.38 - Sl.

```
Si:  '(' '_atom' s-izraz ')'

SS:  (_atom s-izraz)      -→ logički

Se:  (_atom a)            = _true, a je atom
     (_atom ⊥)            = ⊥
     (_atom x)            = _false, inače


Pr:  (_atom (1 (2) a b))        -→ _false
     (_atom Dan)                -→ _true
```

Sl. 2.36 _atom: sintaksa, statička semantika, semantika i primeri

```
Si:  '(' '_number' s-izraz ')'

SS:  (_number s-izraz)          -→ logički

Se:  (_number a)                = _true, a je broj
     (_number ⊥)                = ⊥
     (_number x)                = _false, inače

Pr:  (_number (1 (2) a b))      -→ _false
     (_number Dan)              -→ _false
```

Sl. 2.37 _number: sintaksa, statička semantika, semantika i primeri

2.40). Zadržanim izračunavanjem se naziva uredjeni par koji se sastoji od (neizračunatog) izraza i "okoline" (spiska identifikatora i njihovih vrednosti) u kojoj treba izračunati izraz. U daljem tekstu će se oznakom **T(e)** označavati zadržano izračunavanje izraza **e***. Vrednost izraza **e** čije je izračunavanje zadržano se izračunava (ili: "izračunavanje se forsira") primenom posebne ugradjene funkcije.

Struktura podataka kojom se predstavlja zadržano izračunavanje se često naziva neprevodivom engleskom kovanicom "thunk", a u okruženju apstraktne SECD mašine, "receptom" (engl. *recipe*).

```
Si: '(' '_seq' s-izraz s-izraz ')'

SS:  (_seq s-izraz s-izraz)     -→ s-izraz

Se:  (_seq ⊥ x)                 = ⊥
     (_seq x y)                 = y

Pr:  (_seq (_foreign line (_quote 10) (_quote 10)
                          (_quote 100) (_quote 100))
           (_quote _true)
     )                          -→ _true, i povučena linija na ekranu
```

Sl. 2.38 _seq: sintaksa, statička semantika, semantika i primeri

Funkcija _seq je vrlo često potrebna u čisto-funkcionalnim programskim jezicima (pogotovu ako su nestriktne semantike). Na primer, pri izvršavanju grafičkih ili ulazno-izlaznih operacija koje se inače

---

* Najjednostavniji način za realizaciju zadržanog izračunavanja izraza **e**, je definisanje "prazne" λ apstrakcije [Henderson,1980] str. 219. Dakle jedna moguća definicija je **T(e)=λ( )e**.

```
Si: '(' '_delay' s-izraz ')'

SS: (_delay s-izraz)              -→ recept

Se: (_delay e)                    = T(e)

Pr: (_cons (_quote 1) (_delay (f l)))    -→ (1 . x), x je još neizračunata vrednost
                                             poziva funkcije (f l)
```

Sl. 2.39 **_delay**: sintaksa, statička semantika, semantika i primeri

```
Si: '(' '_force' s-izraz ')'

SS: (_force recept)     -→ s-izraz

Se: (_force T(e))       = e

Pr: (_let (_force (_cdr a))
          (a (_cons (_quote 1) (_delay (f l))))
    ) -→ (1 . x), x je izračunata vrednost poziva funkcije (f l)
```

Sl. 2.40 **_force**: sintaksa, statička semantika, semantika i primeri

(zbog nestriktne semantike) ne bi izvršile. Operator SEQ sa istom semantikom kao i kod funkcije _seq, postoji i u FLIC-u. U FLIC-u pored ovog operatora postoji i operator STRICT, koji pre izračunavanja vrednosti funkcije izračunava prvo argument. Kako se medjutim STRICT može realizovati posredstvom funkcije _seq (a da se time ne naruši čitljivost rezultujućeg programa), analogon STRICT-a se ne nalazi u definiciji LL-a.

Ugradjene funkcije **_delay** i **_force** su nastale da bi se omogućilo zadržavanje izračunavanja u striktnim jezicima; ugradjena funkcija _seq je nastala da bi se forsiralo izračunavanje u nestriktnim jezicima. Realizacija funkcije _seq posredstvom funkcije **_force** ni realizacija funkcije **_force** posredstvom funkcije _seq stoga nije moguća. Ugradjene funkcije **_delay** i **_force** sa semantikom upravo opisanom se nalaze i u LispKit LISP-u u Scheme-u, jezicima sa striktnom semantikom.

2.2.5.13 Posebne ugradjene funkcije

Ugradjene funkcije **_apply** i **_foreign** (Sl. 2.41, Sl. 2.42) primenjuju vrednost svog prvog argumenta (funkciju) na vrednosti ostalih argumenata. Vrednost prvog argumenta funkcije **_apply** nije poznata za vreme prevodjenja LL programa. Tokom izvršavanja programa vrednost prvog argumenta je izvorni LL program (tj. **let** ili **letrec** izraz). Vrednost prvog argumenta funkcije **_foreign** je identifikator funkcije pisane u nekom drugom programskom jeziku (tj. "strane" funkcije - engl. *foreign function*).

Ugradjena funkcija slična funkciji **_apply** ne postoji u FLIC-u (ni u jednom drugom analiziranom medjujeziku), a njegovo postojanje u LL-u se može lako opravdati potrebom da se pišu programi koji treba da manipulišu drugim programima i odmah ih izvršavaju. Takva potreba postoji i pri implementaciji nekih funkcionalnih programskih jezika (na primer FP, [Mitić, 1989]).

Ugradjena funkcija slična funkciji **_foreign** ne postoji u drugim analiziranim medjujezicima, iako potreba za pozivanjem stranih funkcija često postoji u funkcionalnim programskim jezicima i vrlo

često je i realizovana. Jedan način za implementaciju poziva stranih funkcija je prikazan u [Ivanović, Budimac, 1990; Budimac, Ivanović, 1991a].

Primena funkcije _error (Sl. 2.43) uzrokuje prekid izvršavanja programa, pri čemu se vrednost argumenta ispisuje na ekranu terminala.

Kao što se vidi na slici, formalno je vrednost funkcije _error $\perp$. Kao što je već rečeno, konstanta analogna $\perp$ u medjujeziku LL ne postoji. Može se reći da je konstanta $\perp$ implementirana primenom funkcije _error.

```
Si: '(' '_apply' identifikator {s-izraz}')'

SS:  (_apply identifikator s-izraz ... s-izraz) -→ s-izraz

Se:   (_apply f x1 ... xn)                  = f(x1,...,xn)
Se(S):(_apply f x1 ... ⊥ ... xn)            = ⊥

Pr:  (_apply (_let add1 (add1 . (_lambda (x) (_add x (_quote 1)))))
             (_quote 1))          -→ 2
     (_apply (_let add1 (add1 . (_lambda (x) (_add x (quote 2)))))
             (_quote 2))          -→ 4
```

Sl. 2.41 _apply: sintaksa, statička semantika, semantika i primeri

```
Si: '(' '_foreign' identifikator {s-izraz} ')'

SS:  (_foreign identifikator s-izraz ... s-izraz) -→ s-izraz

Se:   (_foreign f x1 ... xn)     = f(x1,...,xn) , f je ime "strane" funkcije
Se(S):(_foreign f x1 ... ⊥ ... xn)       = ⊥
```

Sl. 2.42 _foreign: sintaksa, statička semantika i semantika

```
Si: '(' '_error' s-izraz ')'

SS:  (_error s-izraz)            -→ ⊥

Se:  (_error x)                  = ⊥

Pr:  (_error (_quote (Case: neocekivani slucaj)))  -→ prekida rad programa i ispisuje
                                                     poruku na ekranu
```

Sl. 2.43 _error: sintaksa, statička semantika, semantika i primeri

2.2.6 Bibliotečke definicije

Definicija identifikatora se učitava iz biblioteke na sledeći način: (*ime* . (_from *biblioteka*)) pri čemu je *ime* identifikator čija se definicija učitava ("uvozi") iz biblioteke, a *biblioteka* je naziv biblioteke iz koje se definicija identifikatora

uvozi. Zbog jednostavnosti implementacije, preporučuje se da se naziv biblioteke i naziv fajla u kome se ona nalazi podudaraju.

Uvoženje definicije iz biblioteke se može implementirati na dva načina: a) pronalaženje odgovarajuće definicije identifikatora u biblioteci i njeno "umetanje" umesto s-izraza (_from *b*) u toku prevodjenja glavnog LL programa; ili pronalaženje odgovarajuće prevedene definicije i njeno umetanje na odgovarajuće mesto u prevedenom glavnom LL programu, u toku povezivanja glavnog programa sa bibliotečkim definicijama. U oba slučaja je _from implementirana kao direktiva, a ne kao ugradjena funkcija medjujezika. U [Budimac, Ivanović, Putnik, Tošić, 1991] str. 48 je sa korisničke tačke gledišta opisana realizacija direktive **from** u LispKit LISP-u, kojom se umeće odgovarajuća definicija u toku prevodjenja glavnog programa. Iako bi umetanje prevedene definicije bilo mnogo efikasnije, takvo rešenje vrlo često ne bi bilo jednostavno - na primer prevod definicije funkcije u mašinski jezik SECD mašine se razlikuje u zavisnosti od konteksta u kome se funkcija prevodi.

Definicija bibliotečke funkcije je sledećeg oblika: _**libfun** *ime* **s-izraz.**

Na Sl. 2.44a se nalazi primer definicije bibliotečke funkcije za oduzimanje jedinice od vrednosti argumenta funkcije. Sve pomoćne definicije koje bibliotečka funkcija koristi se smeštaju unutar **_letrec** bloka. Na Sl. 2.44b se nalazi primer definicije funkcije **sub2**, koja oduzima dvojku od vrednosti svog argumenta i u kojoj se koristi pomoćna funkcija **sub1**. U tom slučaju se telo funkcije **sub2** stavlja u **_letrec** blok.

```
a)
libfun sub1   /* r-1 */
(lambda (r)
   (sub r ('1))
)

b)
(libfun sub2   /* r-2 */
  (lambda (r)
     (letrec (sub1 (sub1 r))
             (sub1 . (lambda (r)
                       (sub r ('1))
) ) )        )       )
```

Sl. 2.44 Primeri bibliotečkih funkcija

2.2.7 Čisto-funkcionalni ulaz i izlaz

Čisto funkcionalni ulaz i izlaz se u LL-u ostvaruje posredstvom tzv. tokova (engl. *stream*). Tokovi su u LL-u (kao i u mnogim drugim funkcionalnim programskim jezicima) liste nestriktne semantike. Komunikacija programa sa okolinom se odvija posredstvom **zahteva** (engl. *requests*) operativnom sistemu za izvršenje odredjenih radnji i **odgovora** (engl. *responds*) operativnog sistema. Svaki od zahteva može da prouzrokuje jedan od unapred odredjenih **odgovora.** U programu LL-a koji koristi ulazno-izlazne operacije se poredi rezultat svakog od zahteva sa očekivanim odgovorom i na osnovu toga odredjuje dalje izračunavanje. Poredjenje zahteva i odgovora se najčešće vrši uslovnim izrazom **_case.**

Primetimo da su tokovi (konceptualno beskonačne) liste koje (konceptualno) sadrže čitav fajl, odnosno čitav dijalog, pre početka izračunavanja vrednosti funkcije. U stvarnosti se elementi tokova "generišu" tek kada su potrebni, zahvaljujući lenjom izračunavanju. Čisto-funkcionalni ulaz/izlaz ima dakle smisla samo pri implementaciji jezika nestriktne semantike.

Svi dosada poznati načini realizacije čisto-funkcionalnog ulaza-izlaza se mogu implementirati posredstvom tokova, koji su podržani i u LL-u. Hudak i Sundaresh [1988] su pokazali ekvivalentnost druge dve

poznate realizacije čisto funkcionalnog ulaza-izlaza sa modelom tokova, a Haskell-ova standardna biblioteka **PreludeIO** [Hudak, Peyton Jones, Wadler, 1992] str. 124 prikazuje jednu moguću implementaciju ulaza-izlaza zasnovanu na tzv. nastavcima (engl. *continuation*) korišćenjem tokova.

Zahtevi i odgovori koji će biti implementirani u LL-u zavise od operativnog sistema na kome se LL implementira. Navodimo po jedan mogući skup zahteva i odgovora (sličan Haskell-ovim skupovima zahteva i odgovora). Podrazumeva se da je u narednom tekstu oznakom **ime** označen simbolički atom, a oznakom **s** s-izraz.

(_readFile ime)	učitava sadržaj (tekstualnog) fajla **ime**.
(_readBFile ime)	učitava sadržaj (binarnog) fajla **ime**.
(_writeFile ime s)	piše sadržaj liste **s** u (tekstualni) fajl **ime**.
(_writeBFile ime s)	piše sadržaj liste **s** u (binarni) fajl **ime**.
(_appFile ime s)	dodaje sadržaj liste **s** na sadržaj (tekstualnog) fajla **ime**.
(_appBFile ime s)	dodaje sadržaj liste **s** na sadržaj (binarnog) fajla **ime**.
(_deleteFile ime)	briše fajl **ime**.
(_readChan ime)	čita sa (tekstualnog) uredjaja **ime**.
(_readB ime)	čita sa (binarnog) uredjaja **ime**.
(_appChan ime s)	ispisuje sadržaj liste **s** na (tekstualni) uredjaj **ime**.
(_appBChan ime s)	ispisuje sadržaj liste **s** na (binarni) uredjaj **ime**.

Mogući odgovori operativnog sistema su sledeći:

(_success)	zahtev je uspešno izvršen.
(_res s)	rezultat zahteva se nalazi u listi **s**.
(_failure s)	zahtev nije uspešno izvršen, a razlog je opisan u listi **s**.

Ukoliko se od zahteva operativnom sistemu ne očekuje da vrati neki rezultat izvršavanja, tada su mogući odgovori **_success** ili **_failure** (na primer **_deleteFile**, **_writeFile**, **appChan** ...). Ukoliko se od zahteva operativnom sistemu očekuje da vrati neki rezultat izvršavanja, tada su mogući odgovori **_res** ili **_failure** (na primer **_readFile**, ...).

Semantika zahteva i odgovora operativnom sistemu su karakteristika operativnog sistema a ne medjujezika i zbog toga ovde neće biti detaljnije navodjena. Osnovni zahtevi koje operativni sistem treba da ispuni da bi komunicirao sa funkcionalnim programom su u obliku Haskell programa opisani u [Hudak, Peyton Jones, Wadler, 1992], dodatak D, str. 139.

Glava 3

Translacija čisto-funkcionalnih jezika u LL

U ovoj glavi su data pravila translacije za četiri čisto-funkcionalna programska jezika u medjujezik LL: LispKit LISP, ISWIM, SASL i Haskell. Izborom ova četiri jezika je ilustrovana translacija svih klasa uvedenih u prvoj glavi teze.

- LispKit LISP pripada preseku klasa $F^S_+ \cap F^T_- \cap F^C_- \cap F^U_- \cap F^X_-$ i predstavlja čisto funkcionalne podskupove dijalekata jezika LISP i Scheme,
- ISWIM pripada preseku klasa $F^S_+ \cap F^T_- \cap F^C_- \cap F^U_- \cap F^X_-$.
- SASL pripada preseku klasa $F^S_- \cap F^T_- \cap F^C_+ \cap F^U_- \cap F^X_+$. SASL programi se mogu pisati u obliku jednačina.
- Haskell pripada preseku klasa $F^S_- \cap F^T_+ \cap F^C_+ \cap F^U_+ \cap F^X_+$. Haskell programi se mogu pisati u obliku jednačina, a sadrže čuvare i ZF-izraze. Haskell podržava čisto-funkcionalne nizove i čisto-funkcionalne ulazno - izlazne operacije.

U ovom poglavlju su date definicije sintakse samo onih jezika kod kojih je prvobitna sintaksa na neki način izmenjena. Definicije semantike četiri navedena jezika ovde neće biti navodjene jer su neizmenjene u odnosu na prvobitne.

Translacija izvornih programskih jezika u medjujezik LL će biti definisana funkcijom T, koja preslikava programe izvornog jezika u odgovarajuće programe medjujezika LL. Funkcija T se definiše pravilima preslikavanja za svaku karakterističnu sintaksnu konstrukciju izvornog čisto-funkcionalnog programskog jezika. U procesu translacije se primenjuje prvo primenljivo pravilo.

Pravila preslikavanja predstavljaju opis (translacione) semantike izvornog jezika i mogu poslužiti kao "uputstvo" za implementaciju translatora u medjujezik LL.

Pri definisanju pravila za translaciju će se podrazumevati da su jezičke konstrukcije izvornog jezika sintaksno i statičko-semantički ispravne. Implementacija translatora medjutim mora uključiti proveru ispravnosti programa izvornog jezika.

U daljem tekstu se translacija sva četiri izvorna jezika predstavlja istom oznakom T uvek kada je nedvosmisleno jasno na koji izvorni jezik se funkcija odnosi.

3.1 LispKit LISP

LispKit LISP je definisao P. Henderson [1980] a definicija jezika, proširenja prvobitnog jezika i njegova implementacija je opisana i u [Henderson, Jones, Jones, 1983; Stojković i dr., 1984]. Verzija jezika koja se ovde opisuje [Ivanović, Budimac, Putnik, 1991] je nastala usvajanjem i proširenjem prvobitnih ideja i opisana je u [Budimac, 1990; Budimac, Ivanović, 1990b, 1991b, 1991c]. Jezik je kratko prikazan i u [Budimac, Ivanović, Putnik, Tošić, 1991].

3.1.1 Sintaksa i semantika

Sintaksa verzije jezika LispKit LISP koja će se koristiti u ovom odeljku (i poziva ugradjenih funkcija) je data na Sl. 3.1. Semantika je data u [Budimac, Ivanović, 1993b], a delimično i u [Henderson, 1980; Budimac, Ivanović, Putnik, Tošić, 1991].

3.1.2 Pravila translacije

Dole navedena pravila važe za proizvoljne sintaksno ispravne izraze LispKit LISP-a e, e_i, b_i, x_i, i=1,...,n.

$T[\![(\text{let } e\ (x_1\ .\ e_1)\ (x_2\ .\ e_2)\ \ldots\ (x_n\ .\ e_n))]\!]$
$= (\_\text{let}\ T[\![e]\!]\ (T[\![x_1]\!]\ .\ T[\![e_1]\!])\ (T[\![x_2]\!]\ .\ T[\![e_2]\!])\ \ldots\ (T[\![x_n]\!]\ .\ T[\![e_n]\!]))$
$T[\![(\text{letrec } e\ (x_1\ .\ e_1)\ (x_2\ .\ e_2)\ \ldots\ (x_n\ .\ e_n))]\!]$
$= (\_\text{letrec}\ T[\![e]\!]\ (T[\![x_1]\!]\ .\ T[\![e_1]\!])\ (T[\![x_2]\!]\ .\ T[\![e_2]\!])\ldots(T[\![x_n]\!]\ .\ T[\![e_n]\!]))$
$T[\![(\text{lambda } e_1\ e_2)]\!] = (\_\text{lambda}\ T[\![e_1]\!]\ T[\![e_2]\!])$

Uslovni izrazi

$T[\![(\text{cond } (b_1\ e_1)\ (b_2\ e_2)\ \ldots\ (b_n\ e_n))]\!]$
$= (\_\text{if}\ T[\![b_1]\!]\ T[\![e_1]\!]$
$(\_\text{if}\ T[\![b_2]\!]\ T[\![e_2]\!]$
$(\_\text{if}\ \ldots$
$(\_\text{if}\ T[\![b_n]\!]\ T[\![e_n]\!]\ (\_\text{error}\ (\_\text{quote cond\_error})))\ldots)))$
$T[\![(\text{if } e_1\ e_2\ e_3)]\!] = (\_\text{if}\ T[\![e_1]\!]\ T[\![e_2]\!]\ T[\![e_3]\!])$

```
Lispkit     = '(' idBloka lokDef {lokDef} ')'
idBloka     = 'let' ¦ 'letrec'
lokDef      = '(' simbolički '.' '(' s-izraz ')' ')' ¦
              '(' simbolički '.' '(' 'from' simbolički ')' ')'

s-izraz     = atom ¦ lista ¦ pozUgrFun
lista       = '(' složen ')'
složen      = s-izraz ¦ tačkast ¦ s-izraz složen
tačkast     = s-izraz '.' s-izraz

pozUgrFun   = poz1Arg ¦ poz2Arg ¦ poz3Arg ¦ pozNArg ¦ uslov ¦ blok

poz1Arg     = '(' fun1Arg s-izraz ')'
fun1arg     = 'quote'¦ '`'    ¦ 'car' ¦ 'cdr'¦ 'len' ¦ 'null' ¦ 'atom'¦ 'number'¦
              'delay'¦ 'force'¦ 'not'

poz2Arg     = '(' fun2Arg s-izraz s-izraz ')'
fun2arg     = 'cons'¦ 'append'¦ 'member'¦ 'nth'   ¦ 'rest' ¦ 'add' ¦ 'mul' ¦
              'sub' ¦ 'quo'   ¦ 'div'   ¦ 'mod'   ¦ '_'    ¦ 'eq'  ¦ 'le'  ¦
              'leq' ¦ 'con'   ¦ 'dis'   ¦ 'lambda'¦ 'apply'

poz3Arg     = '(' 'if' s-izraz s-izraz s-izraz ')'

pozNArg     = '(' funNArg s-izraz {s-izraz} ')'
funNarg     = 'or' ¦ 'and' ¦ 'times' ¦ 'plus' ¦ 'list' ¦ 'seq'

uslov       = '(' 'cond'  '(' s-izraz s-izraz ')' {'(' s-izraz s-izraz ')'} ')'

atom        = simbolički ¦ numerički ¦ logički
simbolički  = specijalni ¦ slovo {slovo ¦ cifra}
specijalni  = svi znaci osim slova i cifara

numerički   = celi ¦ realni
celi        = ['-'] cifre
cifre       = cifra {cifra}
realni      = celi '.' cifre ['E' celi] ¦ celi 'E' celi
logički     = 'T' ¦ 'F'

slovo       = 'A' ¦ 'B' ¦ ... ¦ 'Z'
cifra       = '0' ¦ '1' ¦ ... ¦ '9'
```

Sl. 3.1 Sintaksa jedne verzije jezika LispKit LISP

Ugradjene funkcije za rad sa listama

$$T[\![(\text{list } e_1\ e_2 \ldots e_n)]\!] = (\_\text{cons } T[\![e_1]\!]\ (\_\text{cons } T[\![e_2]\!]\ (\ldots(\_\text{cons } T[\![e_n]\!]\ \_\text{nil})\ldots)))$$

$T[\![\text{cons}]\!]$	= _cons	$T[\![\text{append}]\!]$	= _append
$T[\![\text{member}]\!]$	= _member	$T[\![\text{nth}]\!]$	= _nth
$T[\![\text{rest}]\!]$	= _rest	$T[\![\text{car}]\!]$	= _car
$T[\![\text{cdr}]\!]$	= _cdr	$T[\![\text{len}]\!]$	= _len

Aritmetičke ugradjene funkcije

$$T[\![(\text{plus } e_1\ e_2 \ldots e_n)]\!] = (\_\text{add } T[\![e_1]\!]\ (\_\text{add } T[\![e_2]\!]\ (\ldots(\_\text{add } T[\![e_n]\!])\ (\_\text{quote } 0)\ldots)))$$

$T[\![(\text{times } e_1\ e_2 \ldots\ e_n)]\!]$
$\quad = (\_\text{mul } T[\![e_1]\!]\ (\_\text{mul } T[\![e_2]\!]\ (\ldots(\_\text{mul } T[\![e_n]\!])\ (\_\text{quote } 1)\ldots)))$

$T[\![\text{add}]\!]$	$= \_\text{add}$	$T[\![\text{sub}]\!]$	$= \_\text{sub}$
$T[\![\text{quo}]\!]$	$= \_\text{quo}$	$T[\![\text{div}]\!]$	$= \_\text{div}$
$T[\![\text{mod}]\!]$	$= \_\text{mod}$		

Predikati, logičke i relacijske ugradjene funkcije

$T[\![(\text{null } e)]\!] = (\_\text{eq } T[\![e]\!]\ \_\text{nil})$
$T[\![(\text{and } e_1\ e_2 \ldots\ e_n)]\!]$
$\quad = (\_\text{and } T[\![e_1]\!]\ (\_\text{and } T[\![e_2]\!]\ (\ldots(\_\text{and } T[\![e_n]\!]\ (\_\text{quote } \_\text{true}))\ldots)))$
$T[\![(\text{con } e_1\ e_2)]\!] = (\_\text{seq } T[\![e_2]\!]\ (\_\text{and } T[\![e_1]\!]\ T[\![e_2]\!]))$
$T[\![(\text{or } e_1\ e_2 \ldots\ e_n)]\!]$
$\quad = (\_\text{or } T[\![e_1]\!]\ (\_\text{or } T[\![e_2]\!]\ (\ldots(\_\text{or } T[\![e_n]\!]\ (\_\text{quote } \_\text{false}))\ldots)))$
$T[\![(\text{dis } e_1\ e_2)]\!] = (\_\text{seq } T[\![e_2]\!]\ (\_\text{or } T[\![e_1]\!]\ T[\![e_2]\!]))$

$T[\![\text{le}]\!]$	$= \_\text{le}$	$T[\![\text{leq}]\!]$	$= \_\text{leq}$
$T[\![\text{eq}]\!]$	$= \_\text{eq}$	$T[\![\text{atom}]\!]$	$= \_\text{atom}$
$T[\![\text{number}]\!]$	$= \_\text{number}$	$T[\![\text{not}]\!]$	$= \_\text{not}$

Posebne funkcije i konstante

$T[\![(\text{delay } e)]\!] = (\_\text{delay } T[\![e]\!])$
$T[\![(\text{force } e)]\!] = (\_\text{force } T[\![e]\!])$
$T[\![(\text{seq } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!]$
$\quad = (\_\text{seq } T[\![e_1]\!]\ (\_\text{seq } T[\![e_2]\!]\ (\ldots(\_\text{seq } T[\![e_{n-1}]\!]\ T[\![e_n]\!])\ldots)))$
$T[\![(\text{apply } e\ (e_1 \ldots\ e_n))]\!] = (\_\text{apply } T[\![e]\!]\ T[\![e_1]\!] \ldots T[\![e_n]\!])$
$T[\![(\_\ e\ (e_1 \ldots\ e_n))]\!] = (\_\text{foreign } T[\![e]\!]\ T[\![e_1]\!] \ldots T[\![e_n]\!])$

$T[\![(\text{from } e)]\!] = (\_\text{from } T[\![e]\!])$

$T[\![(e_1\ e_2 \ldots\ e_n)]\!] = (T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_n]\!])$
$T[\![(\text{`}\ e)]\!] = T[\![(\_\text{quote } e)]\!]$
$T[\![(\text{quote t})]\!] = (\_\text{quote } \_\text{true})$
$T[\![(\text{quote f})]\!] = (\_\text{quote } \_\text{false})$
$T[\![(\text{quote } e)]\!] = (\_\text{quote } T[\![e]\!])$

$T[\![e]\!] = e$

3.1.3 Komentari

Poziv ugradjene funkcije LispKit LISP-a (seq e_1 e_2 ... e_{n-1} e_n), kojim se (po [Budimac, Ivanović, 1993b]) izračunavaju svi izrazi $e_1,\ldots,e_{n-1}$, a vrednost poziva je vrednost izraza e_n, može da se prevede u LL i na način prikazan na Sl. 3.2, pri čemu su d_i, i=1,...,n-1 identifikatori koji nisu slobodni u izrazima e_i, i=1,...,n.

Poziv funkcije **seq** se može prevesti na ovaj način jedino ako je izvorni jezik striktne semantike, jer u protivnom ne bi došlo do izračunavanja izraza e_i, i=1,...,n-1. Redosled izračunavanja izraza e_i, i=1,...,n-1 nije unapred poznat, već zavisi od implementacije **let** izraza medjujezika LL.

$$
\begin{aligned}
&T[\![(\text{seq } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!] \\
&\quad\longrightarrow (\_\text{let } T[\![e_n]\!] \\
&\qquad\qquad (d_1\ T[\![e_1]\!]) \\
&\qquad\qquad (d_2\ T[\![e_2]\!]) \\
&\qquad\qquad \ldots \\
&\qquad\qquad (d_{n-1}\ T[\![e_{n-1}]\!]))
\end{aligned}
$$

Sl. 3.2 Moguća translacija ugradjene funkcije **seq**

Pozivi ugradjenih funkcija LispKit LISP-a **con** i **dis** se transliraju na prikazani način, da bi se izračunala i vrednost drugog argumenta (što je po semantici tih funkcija potrebno). Funkcije LL-a **_and** i **_or** su nestriktne semantike i njihova implementacija ne izračunava vrednost drugog argumenta, ako je vrednost poziva funkcije izračunljiva na osnovu vrednosti prvog argumenta.

Pravila za translaciju primene LispKit LISP funkcija **plus**, **times**, **and** i **or** mogu da budu definisana i bez razvijanja do primene na izraz e_n i neutralni element operacije. Na primer:

$$
\begin{aligned}
&T[\![(\text{plus } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!] \\
&\quad = (\_\text{add } T[\![e_1]\!]\ (\_\text{add } T[\![e_2]\!]\ (\ldots(\_\text{add } T[\![e_{n-1}]\!]\ T[\![e_n]\!])\ldots))) \\
&T[\![(\text{times } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!] \\
&\quad = (\_\text{mul } T[\![e_1]\!]\ (\_\text{mul } T[\![e_2]\!]\ (\ldots(\_\text{mul } T[\![e_{n-1}]\!]\ T[\![e_n]\!])\ldots))) \\
&T[\![(\text{and } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!] \\
&\quad = (\_\text{and } T[\![e_1]\!]\ (\_\text{and } T[\![e_2]\!]\ (\ldots(\_\text{and } T[\![e_{n-1}]\!]\ T[\![e_n]\!])\ldots))) \\
&T[\![(\text{or } e_1\ e_2 \ldots\ e_{n-1}\ e_n)]\!] \\
&\quad = (\_\text{or } T[\![e_1]\!]\ (\_\text{or } T[\![e_2]\!]\ (\ldots(\_\text{or } T[\![e_{n-1}]\!]\ T[\![e_n]\!])\ldots)))
\end{aligned}
$$

Ovde prikazani LL izrazi pri izvršavanju za n-1 više pristupaju repu liste od LL izraza generisanih pravilima iz prethodnog odeljka.

3.1.4 Primeri

U ovom odeljku su prikazani neki primeri programa pisanih u LispKit LISP-u i njihove translacije u medjujezik LL. Na Sl. 3.3 i Sl. 3.4 se nalaze redom definicije funkcija za obrtanje liste i crtanje spirale tzv. "kornjačinom" grafikom (engl. *turtle graphics*) [Budimac, Ivanović, Putnik, Tošić, 1991] str. 202.

```
(letrec reverse
  (reverse lambda (l)
    (if (null l)
        ('nil)
        (append
            (reverse (cdr l))
            (list (car l))
) ) )   )
```

```
(_letrec reverse
  (reverse _lambda (l)
    (_if (_eq l _nil)
         _nil
         (_append
             (reverse (_cdr l))
             (_cons (_car l) _nil)
) ) )   )
```

Sl. 3.3 Funkcije za obrtanje liste (LispKit LISP i LL)

```
(letrec spirala
  (spirala lambda (k n)
    (strana (quo ('360) k) n))
  (strana lambda (ug n)
     (if (leq n ('0))
         ('t)
         (seq (forward n)
              (turn ug)
              (strana ug
                   (sub n ('2)))
  ) )   )
  (forward from turtle)
  (turn from turtle)
)
```

```
(_letrec spirala
  (spirala _lambda (k n)
    (strana (_quo (_quote 360) k) n))
  (strana _lambda (ug n)
    (_if (_leq n (_quote 0))
         (_quote _true)
         (_seq (forward n)
               (_seq (turn ug)
                     (strana ug
                          (_sub n (_quote 2)))
  ) )   )
  (forward . (_from turtle))
  (turn . (_from turtle))
)
```

Sl. 3.4 Funkcije za crtanje spirale (LispKit LISP i LL)

3.2 ISWIM

ISWIM je definisao P. Landin [1966]. U [Ivanović, Budimac, 1993] je prikazana jedna, u odnosu na prvobitnu verziju, poboljšana verzija jezika i data su pravila za translaciju jezika u podskup jezika Scheme. Verzija jezika koja se ovde opisuje [Budimac, Ivanović, Živkov, 1992] je dalje unapredjenje. Korisničko uputstvo za jezik je dato u [Živkov, Budimac, Ivanović, 1993].

3.2.1 Sintaksa i semantika

Sintaksa verzije jezika ISWIM koja se ovde opisuje se razlikuje od sintakse jezika predložene u [Ivanović, Budimac, 1993], uglavnom po prioritetu operatora i proširenim mogućnostima pojave izraza jezika unutar drugih konstrukcija jezika.

Definicija sintakse jezika prikazana na Sl. 3.6 zadovoljava kriterijume za LL(1) gramatiku ([Aho, Sethi, Ullman, 1985] str. 181, [Tremblay, Sorenson, 1985] str. 236), te je direktno primenljiva kao ulazni podatak za gotovo sve raspoložive generatore sintaksnih analizatora (na primer COCO [Rechenberg, Mössenböck, 1989], COCO-2 [Dobler, Pirklbauer, 1990], YACC [Johnson, 1975] itd.). Semantika verzije ISWIM-a koja se ovde opisuje je istovetna sa semantikom verzije iz [Ivanović, Budimac, 1993].

Svi operatori jezika su dati u tabeli na Sl. 3.5, po opadajućem prioritetu i sa naznačenim vezivanjem operanada (engl. *associativity*).

3.2.2 Pravila translacije

Dole navedena pravila važe za proizvoljne znake c_1, c_2, ..., c_s i sintaksno ispravne izraze programskog jezika ISWIM: f, e, e_i, x_i, y_j, z_t, i=1,...,n, j=1,...,m, t=1,...,k.

Operatori	Vezivanje
~, hd, tl, #, atom	unarni
*, /, div, mod, !, @	levo
+, -	levo
=, <, <=, >, >=, ~=, in	levo
&	levo
|	levo
++, :	desno

Sl. 3.5 Operatori ISWIM-a

$T[\![\{e \text{ where } e_1 \text{ and } e_2 \ldots \text{ and } e_n\}]\!]$
$= (\_\text{let } T[\![e]\!]$
$\quad T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_n]\!])$

$T[\![\{e \text{ where rec } e_1 \text{ and } e_2 \ldots \text{ and } e_n\}]\!]$
$= (\_\text{letrec } T[\![e]\!]$
$\quad T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_n]\!])$

$T[\![\{\text{let } e_1 \text{ and } e_2 \ldots \text{ and } e_n \ ; e\}]\!]$
$= (\_\text{let } T[\![e]\!]\ T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_n]\!])$

$T[\![\{\text{let rec } e_1 \text{ and } e_2 \ldots \text{ and } e_n \ ; e\}]\!]$
$= (\_\text{letrec } T[\![e]\!]\ T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_n]\!])$

Lokalne definicije

$T[\![f(x_1,\ldots,x_n)(y_1,\ldots,y_m)\ldots(z_1,\ldots,z_k) = e]\!]$
$= (T[\![f]\!] \ .\ (\_\text{lambda } (T[\![x_1]\!] \ldots T[\![x_n]\!])$
$(\_\text{lambda } (T[\![y_1]\!] \ldots T[\![y_m]\!]) \ldots$
$(\_\text{lambda } (T[\![z_1]\!] \ldots T[\![z_k]\!])\ T[\![e]\!]) \ldots))$

$T[\![f = e]\!] = (T[\![f]\!] \ .\ T[\![e]\!])$

Relacijski operatori

$T[\![e_1 \sim = e_2]\!] = (\_\text{not } (\_\text{eq } T[\![e_1]\!]\ T[\![e_2]\!]))$
$T[\![e_1 > e_2]\!] = T[\![e_2 < e_1]\!]$
$T[\![e_1 >= e_2]\!] = (\_\text{not } T[\![e_1 < e_2]\!])$

$T[\![e_1\ o\ e_2]\!] = (T[\![o]\!]\ T[\![e_1]\!]\ T[\![e_2]\!])$, o je binarni operator

$T[\![<]\!] = \_\text{le}$ $\quad T[\![=]\!] = \_\text{eq}$
$T[\![<=]\!] = \_\text{leq}$ $\quad T[\![\text{in}]\!] = \_\text{member}$

Uslovni izraz

$T[\![e_1 \text{ -> } e_2; e_3]\!] = (\_\text{if } T[\![e_1]\!]\ T[\![e_2]\!]\ T[\![e_3]\!])$

```
Block       = '(' ( WhereBlock ¦ LetBlock ) ')'
WhereBlock  = Exp WhereRec Def {'AND' Def}
LetBlock    = LetRec Def {'AND' Def} ';' Exp
Def         = Id {IdList} ('=' Exp ¦ 'FROM' Id)
WhereRec    = 'WHERE' ['REC']
LetRec      = 'LET' ['REC']
IdList      = '(' [ Id { ',' Id } ] ')'

Exp         = SeqExp {SeqOp SeqExp} [Cond]
Cond        = '->' Exp ';' Exp
SeqExp      = OrExp {OrOp OrExp}
OrExp       = AndExp {AndOp AndExp}
AndExp      = SimpleExp [RelOp SimpleExp]
SimpleExp   = Term {AddOp Term}
Term        = ['-'] Factor {MulOp Factor}
Factor      = ( '(' (Exp ¦ Anon) ')' ¦ Block ¦ Id ) {ExpList}  ¦  '[' Sequence ']' ¦
              UnaryOp Factor ¦ Constant
Constant    = Number ¦ 'NIL' ¦ QuotedId ¦ 'TRUE' ¦ 'FALSE'
Anon        = '\' Id { ',' Id }  '->' Exp
ExpList     = '(' [Exp { ',' Exp } ] ')'
Sequence    = Exp [',' Sequence]

UnaryOp     = '~' ¦ 'HD' ¦ 'TL' ¦ '#' ¦ 'ATOM'
MulOp       = '*' ¦ '/' ¦ 'DIV' ¦ 'MOD' ¦ '!' ¦ '@'
AddOp       = '+' ¦ '-'
RelOp       = '=' ¦ '<' ¦ '<=' ¦ '>' ¦ '>=' ¦ '~=' ¦ 'IN'
AndOp       = '&'
OrOp        = '¦'
SeqOp       = '++' ¦ ':'

Id          = letter {letter ¦ digit}
Character   = bilo koji znak sa grafičkom prezentacijom
QuotedId    = '"' Character {Character} '"'
Number      = digit {digit} ['.' digit {digit}] ['E' ['-'] digit {digit}]

letter      = 'a' ¦ 'b' ¦ ... ¦ 'z' ¦ 'A' ¦ 'B' ... ¦ 'Z'
digit       = '0' ¦ '1' ¦ ... ¦ '9'
```

Sl. 3.6 Sintaksa jedne verzije jezika ISWIM

Aditivni operatori

$T[\![+]\!]$	= _add	$T[\![-]\!]$	= _sub	
$T[\![\mid]\!]$	= _or	$T[\![++]\!]$	= _append	(*)

Multiplikativni operatori

$T[\![-e]\!]$	= (_sub (_quote 0) $T[\![e]\!]$)			
$T[\![*]\!]$	= _mul	$T[\![/]\!]$	= _quo	
$T[\![\text{div}]\!]$	= _div	$T[\![@]\!]$	= _rest	
$T[\![\text{mod}]\!]$	= _mod	$T[\![\&]\!]$	= _and	
$T[\![!]\!]$	= _nth	$T[\![:]\!]$	= _cons	(*)

Izrazi u zagradi

$T[\![(e)]\!]$ = ($T[\![e]\!]$)

Primena funkcije

$$T[\![f(x_1,\ldots,x_n)(y_1,\ldots,y_m)\ldots(z_1,\ldots,z_k)]\!] = (\ldots(((T[\![f]\!]\ T[\![x_1]\!]\ \ldots\ T[\![x_n]\!])\ T[\![y_1]\!]\ \ldots\ T[\![y_m]\!])\ \ldots)\ T[\![z_1]\!]\ \ldots\ T[\![z_k]\!])$$

Lista kao konstanta

$$T[\![[e_1, e_2, \ldots, e_n]]\!] = (\_\text{cons}\ T[\![e_1]\!]\ (\_\text{cons}\ T[\![e_2]\!]\ (\_\text{cons}\ldots(\_\text{cons}\ T[\![e_n]\!]\ \_\text{nil})\ \ldots)))$$

Unarni operatori

$T[\![o\ e]\!]$ = $(T[\![o]\!]\ T[\![e]\!])$, o je unarni operator

$T[\![\sim]\!]$	= _not	$T[\![\text{hd}]\!]$	= _car
$T[\![\text{tl}]\!]$	= _cdr	$T[\![\#]\!]$	= _len
$T[\![\text{atom}]\!]$	= _atom		

Konstante

$T[\![\text{true}]\!]$	= (_quote _true)
$T[\![\text{false}]\!]$	= (_quote _false)
$T[\![\text{nil}]\!]$	= _nil
$T[\![\text{"}c_1c_2\ldots c_s\text{"}]\!]$	= (_quote $c_1c_2\ldots c_s$)
$T[\![n]\!]$	= (_quote n), n je broj

Anonimna funkcija

$$T[\![(\backslash\ e_1, e_2, \ldots, e_n \text{ -> } e)]\!] = (\_\text{lambda}\ (T[\![e_1]\!]\ T[\![e_2]\!]\ \ldots\ T[\![e_n]\!])\ T[\![e]\!])$$

Svi ostali slučajevi

$$T[\![e]\!] = e$$

3.2.3 Komentari

Operatori označeni zvezdicom u prethodnom odeljku se vezuju za operande sa desna u levo, za razliku od svih ostalih operatora koji se vezuju sa leva u desno, o čemu treba povesti računa prilikom implementacije tih pravila. Dakle:

$$T[\![e_1 + e_2 + \ldots + e_n]\!] = T[\![\ ((\ldots(e_1 + e_2) + \ldots + e_{n-1}) + e_n)]\!]$$
$$= (\_\text{add}\ (\_\text{add}\ldots(\_\text{add}\ T[\![e_1]\!]\ T[\![e_2]\!])\ldots T[\![e_{n-1}]\!])\ T[\![e_n]\!])$$

ali

$$T[\![e_1 : e_2 : \ldots : e_n]\!] = T[\![(e_1 : (e_2 : \ldots (e_{n-1} : e_n)\ldots))]\!]$$
$$= (\_\text{cons } T[\![e_1]\!] (\_\text{cons } T[\![e_2]\!] \ldots (\_\text{cons } T[\![e_{n-1}]\!] \; T[\![e_n]\!])\ldots))$$

Sintaksa ISWIM-a je "dvodimenzionalna", što znači da i zapis programa utiče na tumačenje pojedinih sintaksnih konstrukcija. Tako se na primer, znaci { i } mogu zameniti odgovarajućim "nazubljivanjem" teksta. Pravila zamenjivanja razdvajača jezika odgovarajućim izgledom ISWIM programa su data u [Landin, 1966].

3.2.4 Primeri

U ovom odeljku su prikazani primeri programa pisanih u ISWIM-u i njihove translacije u LL. Na Sl. 3.7 do Sl. 3.9 se nalaze definicije redom funkcija za izračunavanje broja particija jednog broja, obrtanje liste na svim nivoima i pronalaženje maksimuma liste korišćenjem funkcije višeg reda.

```
{part
  where rec
   part(n) = par(n,n) and
   par(m,n) =
          n=1 -> 1;
          m=1 -> 1;
          m<n -> par(m,m);
          m=n -> par(m,m-1) + 1;
          par(m,n-1)+par(m-n,n)
}
```

```
(_letrec part
 (part _lambda (n)
   (par n n)
 )
 (par _lambda (m n)
   (_if (_eq n (_quote 1))
        (_quote 1)
        (_if (_eq m (_quote 1))
             (_quote 1)
             (_if (_le m n)
                  (par m m)
                  (_if (_eq m n)
                       (_add
                         (par m
                           (_sub m (_quote 1)))
                           (_quote 1))
                       (_add
                         (par m
                           (_sub n (_quote 1)))
                           (par (_sub m n) n))
))      )    )    )
```

Sl. 3.7 Funkcija za izračunavanje particija broja (ISWIM i LL)

3.3 SASL

SASL je definisao D. A. Turner [1976]. Jezik je kasnije proširivan na razne načine, ali se ovde opisuje prvobitna verzija.

```
{revAll where rec
 revAll(e) =
  e=nil -> nil;
  {let h = hd e and
       t = tl e;
       atom h  ->
         revAll(t) ++ [h];
         revAll(t)++ [revAll(h)]
  }
}
```

```
(_letrec revall
  (revall _lambda (e)
    (_if (_eq e _nil)
         _nil
         (_let (_if (_atom h)
                    (_append (revall t)
                             (_cons h _nil))
                    (_append (revall t)
                             (cons (revall h)
                                   _nil))
               )
               (h . (_car e))
               (t . (_cdr e))
  ) ) )  )
```

Sl. 3.8 Funkcije za "dubinsko" obrtanje liste (ISWIM i LL)

```
{max where rec
     max(e) =
       fold( (\a,b-> a<=b->b;a),
             hd e, tl e)
 and fold(f,init,e) =
       e=nil->init;
       f(hd e, fold(f,init,tl e))
}
```

```
(_letrec max
   (max _lambda (e)
        (fold (_lambda (a b)
                  (_if (_leq a b) b a))
              (_car e) (_cdr e))
        )
   (fold _lambda (f init e)
         (_if (_eq e _nil)
              init
              (f (_car e)
                 (fold f init (_cdr e)))
  ) )    )
```

Sl. 3.9 Funkcije za traženje maksimuma liste (ISWIM i LL)

3.3.1 Sintaksa i semantika

Opis sintakse u [Turner, Joy, 1990] sadrži greške i nepreciznosti*, te je na Sl. 3.10 data ispravljena definicija sintakse jezika SASL [Budimac, Nikolajević, 1993]. Iz prikazane definicije su izostavljeni delovi koji se odnose na komande SASL sistemu (implementaciji jezika SASL). Definicija sintakse SASL-a takodje zadovoljava kriterijume za LL(1) gramatiku. Semantika jezika je neizmenjena u odnosu na semantiku prikazanu u [Turner, 1976; Turner, Joy, 1990].

Tipovi podataka u SASL-u su brojevi, znaci, funkcije, logički tip i liste. Konstruktori podataka za (jedini složeni tip podataka) listu su : i ( ). Izrazi SASL-a: ( ), *x*:*y*, konstante i identifikatori su karakteristični oblici (engl. *pattern*) jezika, koji se mogu koristiti u jednačinama programa.

Svi operatori jezika su dati u tabeli, po opadajućem prioritetu i sa naznačenim vezivanjem operanada (engl. *associativity*).

---

* Na primer, po sintaksi opisanoj u [Turner, Joy, 1990] str. 13, operator : se ne sme pojavljivati sa leve strane jednakosti u jednačinama, što nije tačno.

```
Sasl          = command {command}
command       = exp '?' | '/DEF' defs '?'
exp           = allexp | '[' allexp 'WHERE' defs ']'
allexp        = opexp (condexp | listexp)
condexp       = '->' exp ';' exp
listexp       = ',' opexp listexp
opexp         = exp2 { ( ':' | '++') exp2 }
exp2          = exp3 {orop exp3}
exp3          = exp4 {'&' exp4 }
exp4          = { '~' } exp5 { relop exp5 }
exp5          = exp6 { addop exp6 }
exp6          = {addop} exp78 {multop exp78}
exp78         = arg { arg} { '.' arg { arg }}
arg           = name | constant | '(' exp ')'
constant      = numeral | float | logicalconst | string | char | '()'
float         = dfloat
logicalconst  = 'TRUE' | 'FALSE'
defs          = def ';' { def ';' }
def           = name ( functionform | nmst ) | formal1 nmst '=' exp
nmst          = namelist | namelist1
namelist1     = ':' (name | formal1) namelist1
namelist      = ',' (name | formal1) namelist
formal1       = constant | '(' (name | formal1)
                         { ':' (name | constant ) } namelist ')'
functionform  = name | formal1 {name | formal1}
orop          = '|'
relop         = '>' | '>=' | '=' | '~=' | '<=' | '<'
addop         = '+' | '-'
multop        = '*' | '/' | 'DIV' | 'MOD'
numeral       = digit {digit}
dfloat        = digit {digit} '.' digit {digit}
string        = '"' [character {character}] '"'
name          = letter {letter | digit}
char          = '%' character
```

Sl. 3.10 Definicija sintakse jezika SASL

3.3.2 Pravila translacije

Dole navedena pravila važe za znake c_1, c_2, ..., c_s, sintaksno ispravne izraze programskog jezika SASL f, f_i, e, e_j i sintaksno ispravne oblike jezika SASL p_j^q, i=1,...,k, j=1,...,m, q=1,...n.

$$T[\![\{e \text{ where } e_1\ e_2\ \ldots\ e_m\}]\!] \quad = (\_\text{letrec } T[\![e]\!]\ T[\![e_1]\!]\ T[\![e_2]\!] \ldots T[\![e_m]\!])$$

pri čemu su e_j, i=1,...,m grupe jednačina kojim su definisane različite funkcije (tj. počinju različitim identifikatorom).

Translacija jednačina

Jednačine SASL programa se transliraju u ugnježdene pozive ugradjene funkcije **_case** medjujezika LL. Kako je translacija jednačina komplikovanija od ostalih konstrukcija SASL-a, uvodimo posebnu funkciju M za njihovu translaciju, po ugledu na funkcije *match* opisane u [Wadler, 1987a; Maranget, 1992]. Funkcija M se poziva sa 4 argumenta Y, U, P i E, koji imaju sledeće značenje: Y je lista

novih identifikatora, U je lista uslova koji treba da budu ispunjeni da bi se izrazi $e_1,\ldots,e_m$ izračunali, P su jednačine programa bez identifikatora funkcije i E je izraz čija vrednost treba da se izračuna ako poredjenje ni sa jednom jednačinom iz liste P nije uspelo. Početne vrednosti za ove argumente su redom: lista identifikatora $y_1,\ldots,y_n$; lista od m uvek ispunjenih uslova **(_quote _true)**, jednačine koje definišu funkciju *f*, bez identifikatora funkcije; i poruka o grešci.

Operatori	Vezivanje
*, /, div, mod	levo
+, -	unarni
+, -	levo
=, <, <=, >, >=, ~=	levo
~	unarni
&	levo
|	levo
++, :	desno
.	levo

Sl. 3.11 Operatori SASL-a

$$
T\left[\begin{bmatrix} f\ p_1^1\ p_1^2\ \ldots\ p_1^n = e_1; \\ f\ p_2^1\ p_2^2\ \ldots\ p_2^n = e_2; \\ \ldots \\ f\ p_m^1\ p_m^2\ \ldots\ p_m^n = e_m; \end{bmatrix}\right] =
$$

```
(f.
  (_lambda(y1)
    (_lambda(y2)
      ...
      (_lambda(yn)
```

$$
M\left(\begin{bmatrix} [y_1,\ldots,y_n] \\ [(\text{\_quote\_true}),\ldots,(\text{\_quote \_true})] \\ \begin{bmatrix} T[\![p_1^1]\!]\ T[\![p_1^2]\!]\ \ldots\ T[\![p_1^n]\!] = T[\![e_1]\!] \\ T[\![p_2^1]\!]\ T[\![p_2^2]\!]\ \ldots\ T[\![p_2^n]\!] = T[\![e_2]\!] \\ \ldots \\ T[\![p_m^1]\!]\ T[\![p_m^2]\!]\ \ldots\ T[\![p_m^n]\!] = T[\![e_m]\!] \end{bmatrix} \\ [(\text{\_error(\_quote greska\_case)})] \end{bmatrix}\right)
$$

```
) ) ) )
```

pri čemu su y_1, y_2, ..., y_n "novi" identifikatori (tj. identifikatori koji se ne pojavljuju slobodni u e_j, j=1,...,m).

$$T[\![e_1 = e_2;]\!] = (T[\![e_1]\!] \ . \ T[\![e_2]\!])$$

Pri definisanju funkcije M, koristi se i notacija E[x/y] koja označava LL izraz E u kome su sve pojave identifikatora y zamenjene izrazom x, i oznake q_j^i i h_j, j=1,...,m, i=1,...,n; koje zamenjuju redom $T[\![p_j^i]\!]$ i $T[\![e_j]\!]$. Funkcija M je definisana na sledeći način:

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \left[\begin{array}{llll} q_1^i & \ldots & q_1^{i+k} & = h_1\\ & \ldots & & \\ q_j^i & \ldots & q_j^{i+k} & = h_j\\ & \ldots & & \\ q_m^i & \ldots & q_m^{i+k} & = h_m\end{array}\right]\\ E\end{pmatrix} = M\begin{pmatrix}[y_{i+1},\ldots,y_{i+k}]\\ [u_1,\ldots,(\_\text{and}(\_\text{eq } y_i\ q_j^i)\ u_j),\ldots,u_m]\\ \left[\begin{array}{llll} q_1^{i+1} & \ldots & q_1^{i+k} & = h_1[y_i/q_1^i]\\ & \ldots & & \\ q_j^{i+1} & \ldots & q_j^{i+k} & = h_j[y_i/q_j^i]\\ & \ldots & & \\ q_m^{i+1} & \ldots & q_m^{i+k} & = h_m[y_i/q_m^i]\end{array}\right]\\ E\end{pmatrix}$$

ako su $q_1^i,\ldots,q_m^i$ identifikatori ili konstante jezika LL, tako da je za proizvoljan broj $j\in\{1..m\}$, q_j^i, konstanta.

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \left[\begin{array}{llll} q_1^i & \ldots & q_1^{i+k} & = h_1\\ & \ldots & & \\ q_j^i & \ldots & q_j^{i+k} & = h_j\\ q_{j+1}^i & \ldots & q_{j+1}^{i+k} & = h_{j+1}\\ & \ldots & & \\ q_m^i & \ldots & q_m^{i+k} & = h_m\end{array}\right]\\ E\end{pmatrix} =$$

$$(\_\text{case } y_i$$

$$(\_\text{nil} \;.\; M\begin{pmatrix}[y_{i+1},\ldots,y_{i+k}]\\ [u_1,\ldots,u_j]\\ \left[\begin{array}{llll} q_1^{i+1} & \ldots & q_1^{i+k} & = h_1\\ & \ldots & & \\ q_j^{i+1} & \ldots & q_j^{i+k} & = h_j\end{array}\right]\\ E\end{pmatrix})$$

$$(\_\text{cons} \;.\; (\_\text{let } M\begin{pmatrix}[z_1,z_2,y_{i+1},\ldots,y_{i+k}]\\ [u_{j+1},\ldots,u_m]\\ \left[\begin{array}{llll} a_{j+1} & b_{j+1} & q_{j+1}^{i+1} & \ldots\ q_{j+1}^{i+k} = h_{j+1}\\ & & \ldots & \\ a_m & b_m & q_m^{i+1} & \ldots\ q_m^{i+k} = h_m\end{array}\right]\\ E\end{pmatrix}$$

$$(z_1 \;.\; (\_\text{car } y_i))$$

$$(z_2 \;.\; (\_\text{cdr } y_i))$$

$$))$$

$$))$$

ako su oblici q_1^i do q_j^i jednaki **_nil**, a oblici q_{j+1}^i do q_m^i jednaki (**_cons** a_k b_k), $k=j+1,\ldots,m$, pri čemu su z_1 i z_2 novi identifikatori. Redosled jednačina u okviru jednog konstruktora se ne sme menjati.

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \left[\begin{array}{l} q_1^{\,i}\ \ldots\ q_1^{\,i+k} = h_1\\ \quad\ldots\\ q_m^{\,i}\ \ldots\ q_m^{\,i+k} = h_m\end{array}\right]\\ E\end{pmatrix} = M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_{j_1}]\\ \left[\begin{array}{l} q_1^{\,i}\ \ldots\ q_1^{\,i+k} = h_1\\ \quad\ldots\\ q_{j_1}^{\,i}\ \ldots\ q_{j_1}^{\,i+k} = h_{j_1}\end{array}\right]\\ M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_{j_1+1},\ldots,u_{j_2}]\\ \left[\begin{array}{l} q_{j_1+1}^{\,i}\ \ldots\ q_{j_1+1}^{\,i+k} = h_{j_1+1}\\ \quad\ldots\\ q_{j_2}^{\,i}\ \ldots\ q_{j_2}^{\,i+k} = h_{j_2}\end{array}\right]\\ \ldots\\ M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_{j_r-1},\ldots,u_{j_r}]\\ \left[\begin{array}{l} q_{j_r-1}^{\,i}\ \ldots\ q_{j_r-1}^{\,i+k} = h_{j_r-1}\\ \quad\ldots\\ q_{j_r}^{\,i}\ \ldots\ q_{j_r}^{\,i+k} = h_{j_r}\end{array}\right]\\ E\end{pmatrix}\end{pmatrix}\end{pmatrix}$$

ako grupe $q_1^{\,i}$ do $q_{j_1}^{\,i}$, $q_{j_1+1}^{\,i}$ do $q_{j_2}^{\,i}$, $q_{j_2+1}^{\,i}$ do $q_{j_3}^{\,i}$, ..., $q_{j_r-1}^{\,i}$ do $q_{j_r}^{\,i}$, pripadaju istoj vrsti oblika, a $1 \le i \le i+k \le n$, $j_r = m$. Istim vrstama oblika se smatraju: a) identifikatori i konstante medjujezika LL; b) konstruktor podataka **_nil**; c) konstruktor podataka **_cons** ($r \le 3$). Pri odredjivanju grupa jednačina koje pripadaju istoj vrsti, originalni redosled jednačina se ne sme menjati, osim zamene mesta dvema jednačinama koje počinju različitim konstruktorima podataka. Posle uzastopnih primena prethodnih pravila, skup jednačina SASL programa se svodi na sledeće slučajeve:

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_m]\\ \left[\begin{array}{l} = h_1\\ \quad\ldots\\ = h_m\end{array}\right]\\ E\end{pmatrix} = \begin{array}{l}(\_\mathrm{if}\ u_1\ h_1\\ \quad(\_\mathrm{if}\ u_2\ h_2\\ \qquad\ldots\\ \qquad(\_\mathrm{if}\ u_m\ h_m\ E)\\ \qquad\ldots\\ \quad)\ )\end{array}$$

$$M\begin{pmatrix}[\,]\\ A\\ [\,]\\ E\end{pmatrix} = E$$

Bibliotečke funkcije

$$T[\![\ \{/\text{def } e\ ?\}\ ]\!] = L[\![\ e\ ]\!]$$

$$L\left[\!\!\left[\begin{array}{l} f\ p_1^1\ p_1^2\ \ldots\ p_1^n = e_1; \\ f\ p_2^1\ p_2^2\ \ldots\ p_2^n = e_2; \\ \quad\ldots \\ f\ p_m^1\ p_m^2\ \ldots\ p_m^n = e_m; \end{array}\right]\!\!\right] =$$

```
libfunf
(_lambda(y1)
 (_lambda(y2)
  ...
  (_lambda(yn)
   M[[y1,...,yn]
     [(_quote_true),...,(_quote_true)]
     [[T[p1^1] T[p1^2] ... T[p1^n] = T[e1]]
      [T[p2^1] T[p2^2] ... T[p2^n] = T[e2]]
                ...
      [T[pm^1] T[pm^2] ... T[pm^n] = T[em]]]
     [(_error(_quote greska_case))]]
  ) ) )
```

ako definicija funkcije *f* ne zavisi ni od jedne druge definicije.

$$L\left[\!\!\left[\begin{array}{l} f\ p_1^1\ p_1^2\ \ldots\ p_1^n = e_1; \\ f\ p_2^1\ p_2^2\ \ldots\ p_2^n = e_2; \\ \quad\ldots \\ f\ p_m^1\ p_m^2\ \ldots\ p_m^n = e_m; \end{array}\right]\!\!\right] =$$

```
_libfunf
(_lambda(y1)
 (_lambda(y2)
  ...
  (_lambda(yn)
   (_letrec M[[y1,...,yn]
               [(_quote_true),...,(_quote_true)]
               [[T[p1^1] T[p1^2] ... T[p1^n] = T[e1]]
                [T[p2^1] T[p2^2] ... T[p2^n] = T[e2]]
                          ...
                [T[pm^1] T[pm^2] ... T[pm^n] = T[em]]]
               [(_error(_quote greska:case))]]
              T[f1] T[f2] ... T[fk] )
  ) ) )
```

ako definicija funkcije *f* zavisi od definicija funkcija $f_1, f_2, \ldots, f_k$.

Strukturne definicije

$$T[\![\ e_1, e_2, \ldots, e_n = e\ ]\!] = (e_1\ .\ (\_\text{nth } T[\![\ e\ ]\!]\ (\_\text{quote } 1)))$$
$$(e_2\ .\ (\_\text{nth } T[\![\ e\ ]\!]\ (\_\text{quote } 2)))$$
$$\ldots.$$
$$(e_n\ .\ (\_\text{nth } T[\![\ e\ ]\!]\ (\_\text{quote } n)))$$

$$T[\![\ e_1: \ldots: e_{n-1}: e_n = e\ ]\!] = (e_1\ .\ (\_\text{nth } T[\![\ e\ ]\!]\ (\_\text{quote } 1)))$$
$$\ldots.$$
$$(e_{n-1}\ .\ (\_\text{nth } T[\![\ e\ ]\!]\ (\_\text{quote } n\text{-}1)))$$
$$(e_n\ .\ (\_\text{rest } T[\![\ e\ ]\!]\ (\_\text{quote } n\text{-}1)))$$

Uslovni izraz

$T[\![e_1 \text{ -> } e_2; e_3]\!]$ = (_if $T[\![e_1]\!]$ $T[\![e_2]\!]$ $T[\![e_3]\!]$)

Lista

$T[\![e_1, e_2, \ldots, e_n]\!]$
= (_cons $T[\![e_1]\!]$ (_cons $T[\![e_2]\!]$ (_cons ... (_cons $T[\![e_n]\!]$ _nil)...)))
$T[\![e,]\!]$ = (_cons $T[\![e]\!]$ _nil)

Operatori

$T[\![\sim e]\!]$ = (_not $T[\![e]\!]$)
$T[\![- e]\!]$ = (_sub (_quote 0) $T[\![e]\!]$)
$T[\![+ e]\!]$ = $T[\![e]\!]$
$T[\![e_1 \sim = e_2]\!]$ = (_not (eq $T[\![e_1]\!]$ $T[\![e_2]\!]$))
$T[\![e_1 > e_2]\!]$ = $T[\![e_2 < e_1]\!]$
$T[\![e_1 >= e_2]\!]$ = (_not $T[\![e_1 < e_2]\!]$)
$T[\![e_1 . e_2]\!]$ = (_lambda (x) e_1 (e_2 x))
x nije slobodni identifikator u e_1 i e_2
$T[\![e_1 \; o \; e_2]\!]$ = ($T[\![o]\!]$ $T[\![e_1]\!]$ $T[\![e_2]\!]$)
o je binarni operator SASL-a

$T[\![++]\!]$	= _append			(*)
$T[\![:]\!]$	= _cons			(*)
$T[\![\mid]\!]$	= _or	$T[\![\&]\!]$	= _and	
$T[\![<]\!]$	= _le	$T[\![=]\!]$	= _eq	
$T[\![<=]\!]$	= _leq	$T[\![+]\!]$	= _add	
$T[\![-]\!]$	= _sub	$T[\![*]\!]$	= _mul	
$T[\![/]\!]$	= _quo	$T[\![\text{div}]\!]$	= _div	
$T[\![\text{mod}]\!]$	= _mod			

Primena funkcije

$T[\![e_1 \; e_2]\!]$ = ($T[\![e_1]\!]$ $T[\![e_2]\!]$)
$T[\![e_1 \; e_2]\!]$ = (_nth $T[\![e_1]\!]$ $T[\![e_2]\!]$) , ako $T[\![e_1]\!]$ ima vrednost funkcije

$T[\![(e)]\!]$ = ($T[\![e]\!]$)

Konstante

$T[\![\text{true}]\!]$ = (_quote _true)
$T[\![\text{false}]\!]$ = (_quote _false)
$T[\![()]\!]$ = _nil

$T[\![\%c]\!]$ = (_quote c)
$T[\!["c_1c_2...c_s"]\!]$ = (_quote (c_1 c_2 ... c_s))
$T[\![n]\!]$ = (_quote $T[\![n]\!]$), n je broj

$T[\![e]\!]$ = e

3.3.3 Komentari

U definiciji funkcije M iz prethodnog odeljka se "krije" algoritam* L. Augustsson-a [1985] za translaciju jednačina izvornog programa (ovde SASL programa) u ugnježdene pozive ugradjenih funkcija _case ciljnog jezika (ovde medjujezika LL). Za razliku od originalnog algoritma, definicija funkcije M opisuje i transliranje konstanti kao oblika. Augustsson-ov algoritam je opisan i u [Peyton Jones, 1987] str. 81, a alternativni načini transliranja jednačina izvornog jezika u ugnježdjeni **case** izraz (tj. alternativni načini implementacije funkcije M) se mogu pronaći, na primer, u [Field, Harrison, 1988] str. 173 i [Maranget, 1992]. Algoritam transliranja jednačina izvornog programa u primene standardnih operatora bilo kog funkcionalnog programskog jezika (a ne primene specijalizovane _case funkcije) je prikazan u [Nikolajević, Budimac, 1992; Ivanović, Budimac, Nikolajević, 1993].

Implementacija funkcije M ne mora da formira pozive ugnježdjenih uslovnih izraza kada je odgovarajući uslov u_i jednak (_quote _true). Primeri translacije prikazani u sledećem odeljku podrazumevaju upravo takvu implementaciju.

Translacijom strukturnih definicija se dobija niz naizgled nezavisnih LL s-izraza. Znajući, medjutim, da se strukturna definicija po sintaksi SASL jezika može naći samo unutar **where** bloka, konačan rezultat je valjani LL izraz, jer se nalazi unutar LL **letrec** izraza (translacija SASL izraza where).

Strukturne definicije se mogu translirati i na drugačiji način, koji je ilustrovan translacijom Haskell izraza oblika $p = e$ (pri čemu je p oblik), u sledećem odeljku.

Pri translaciji SASL izraza koristi se samo **letrec** izraz medjujezika LL, jer on odgovara po semantici izrazu **where** SASL-a. Nad **letrec** izrazom LL-a se medjutim može izvršiti tzv. "analiza zavisnosti" (engl. *dependency analysis*), koja polazni **letrec** izraz može transformisati u ekvivalentan izraz koji sadrži i **let** izraze. Takav izraz se po pravilu efikasnije izvršava. Opšti principi analize zavisnosti su dati u [Peyton Jones, 1987] str. 118, [Diller, 1988], str. 138, [Putnik, Budimac, Ivanović, 1992a]. U [Putnik, Ivanović, Budimac, 1993] je opisana implementacija analize zavisnosti za medjujezik LL.

---

* Algoritam prevodjenja jednačina u ugnježdeni _case izraz ne treba, medjutim, mešati sa definicijom funkcije M. Funkcija M opisuje oblike rezultujućeg _case izraza u zavisnosti od oblika jednačina izvornog jezika, a ne detaljan način transformacije jednačina u rezultujući izraz.

U većini implementacija SASL-a, podrazumeva se postojanje relativno velike rezidentne biblioteke korisničkih funkcija, dok LL poznaje samo spoljašnje biblioteke funkcija. Zbog toga svaki poziv (*rezidentne*) bibliotečke funkcije u SASL-u, treba da se prevede u poziv odgovarajuće (spoljašnje) bibliotečke funkcije LL-a (direktivu **_from**).

Operatori označeni zvezdicom u prethodnom odeljku se vezuju sa desna u levo, za razliku od svih ostalih operatora koji se vezuju sa leva u desno, o čemu treba povesti računa prilikom implementacije tih pravila. SASL takodje ima "dvodimenzionalnu" sintaksu, u kojoj je važan i izgled programa. Detaljnija pravila zamene razdvajača jezika izgledom programa su dati u [Turner, 1976; Turner, Joy, 1990].

U pojedinim slučajevima je moguće izbeći translaciju (Curry-jevih) definicija funkcija SASL-a u ugnježdene anonimne funkcije LL-a, na primer ukoliko se može utvrditi da ta funkcija nikad neće biti pozivana sa manjim brojem argumenata od broja sa kojim je definisana. Ukoliko se izvrši takva analiza, definicije funkcija se mogu translirati samo u minimalno ugnježdene anonimne funkcije. Ova optimizacija se naziva i *uncurrying*.

3.3.4 Primeri

U ovom odeljku su prikazani neki primeri programa pisanih u SASL-u i njihove translacije u LL. Na Sl. 3.12 do Sl. 3.14 se nalaze redom definicije funkcije čija je vrednost lista sa prvih *n* elemenata vrednosti argumenta; bibliotečke funkcije čija je vrednost poslednji element liste; i za preslikavanje proizvoljne liste u listu brojeva od 0 do n-1 (pri čemu je *n* dužina liste).

```
(take
  where
   take 0 x = ();
   take n () = ();
   take n (a:x) = a:take (n-1) x
)
```

```
(_letrec take
  (take _lambda (y1)
          (_lambda (y2)
           (_if (_eq y1 (_quote 0))
               _nil
               (_case y2
                 (_nil . _nil)
                 (_cons .
                    (_let
                      (_cons y3
                        (take
                         (_sub y1 (_quote 1))
                         y4
                      ) )
                      (y3 . (_car y2))
                      (y4 . (_cdr y2))
) )       ))   ) ) )
```

Sl. 3.12 Funkcija za izračunavanje prvih n elemenata liste (SASL i LL)

```
/def
 last (a:()) = a;
 last (a:b) = last b
?
```

```
_libfun last
(_lambda (y1)
    (_case y1
      (_nil . (_error _quote (Error: Case)))
      (_cons .
        (_let
          (_case y3
            (_nil . y2)
            (_cons .
              (_let (last y3)
                    (y4 . (_car y3))
                    (y5 . (_cdr y3))
              ) )  )
          (y2 . (_car y1))
          (y3 . (_cdr y1))
)  ) ) )
```

Sl. 3.13 Bibliotečka funkcija za izračunavanje poslednjeg elementa liste (SASL i LL)

```
(index "hello"
  where
   index x =
     (f 0 x
        where
          f n () = ();
          f n (a:x) = n:f(n+1) x
     )
)
```

```
(_letrec
   (index (_quote (h e l l o))
   (index _lambda (x)
      (_letrec (f (_quote 0) x)
         (f . _lambda (y1)
              (_lambda (y2)
               (_case y2
                 (_nil . _nil)
                 (_cons .
                   (_let
                     (_cons y1
                       (f (_add y1 (_quote 1))
                          y4
                     ) )
                     (y3 . (_car y2))
                     (y4 . (_car y2))
) ) ) ) )      )) ) )
```

Sl. 3.14 Funkcija za "indeksiranje" liste (SASL i LL)

3.4 Haskell

Haskell je definisala grupa istraživača na čelu sa Hudak-om, Peyton Jones-om i Wadler-om [1991] u želji da stvori "de-fakto" standard u oblasti čisto-funkcionalnih programskih jezika. Haskell poseduje mnoge osobine drugih funkcionalnih programskih jezika (ML, Hope, KRC, Miranda itd...). Poslednja

revizija Haskell-a je definisana u [Hudak, Peyton Jones, Wadler, 1992], gde je predložena njegova implementacija u dva dela:

- implementacija tzv. jezgra jezika Haskell (engl. *Haskell core*), koja sadrži minimum potrebnih operatora jezika i
- implementacija ostalih operatora jezika posredstvom operatora iz jezgra. Ove definicije se nalaze u 8 biblioteka funkcija čiji nazivi počinju sa "Prelude".

Ovakav način implementacije Haskell-a je ilustrativan i u definiciji jezika se ohrabruju i drugačije implementacije. Ovaj odeljak opisuje implementaciju Haskell-a translacijom u medjujezik LL, čime će se u potpunosti iskoristiti ugradjene funkcije LL-a i smanjiti potreba za definisanjem nekih operatora unutar standardnih biblioteka. Opis Haskell-a će zbog toga sadržavati dva dela: translacija nekih od konstrukcija Haskell-a u konstrukcije LL-a i izmene koje je potrebno izvršiti unutar pojedinih standardnih (**Prelude**) biblioteka Haskell-a.

Od brojevnih tipova podataka, Haskell sadrži **Int**, **Float** i **Double** i po tome nije kompatibilan sa LL-om, koji sadrži samo jedan brojevni tip podataka i ne pravi razliku izmedju celih i realnih brojeva. Po toj osobini je LL bliži jeziku Scheme [Clinger, Rees, 1991] nego Haskell-u, iako se tvrdi da je Haskell-ovo tretiranje brojeva gotovo u potpunosti preuzeto od jezika Scheme. Zbog ove nekompatibilnosti, implementacija Haskell-a posredstvom LL-a ne sadrži gornja tri tipa podataka.

3.4.1 Sintaksa i semantika

Opisuje se verzija jezika iz [Hudak, Peyton Jones, Wadler, 1992] str. 132, te će opis sintakse i semantike jezika biti izostavljen.

Tipovi podataka u Haskell-u su brojevi (nekoliko tipova), znaci, stringovi, funkcije, logičke vrednosti, liste, n-torke, prosti tip podataka (tzv. "*unit*"), binarni tok i niz. Moguće je uvesti korisničke tipove podataka i to kao algebarske ili apstraktne tipove podataka.

Konstruktori podataka u Haskell-u su konstruktori za gradjenje listi (: , [ ] i $[e_1,\ldots,e_n]$), konstruktori za gradjenje n-torki ($(e_1,\ldots,e_n)$), konstruktori za gradjenje prostog tipa podataka ("prazne n-torke") (( )) i konstruktori korisničkih tipova podataka (uvedeni deklaracijom **data**).

Oblici jezika Haskell (engl. *patterns*) su svi konstruktori podataka Haskell-a (ugradjeni i uvedeni), konstante jezika, identifikatori, anonimni identifikator (_), kao i izrazi oblika $x+k$, pri čemu je x (anonimni) identifikator, a k ceo broj. Pored toga oblici Haskell-a su i takozvani imenovani oblici (originalno: *as-pattern*) oblika $x@p$, pri čemu je x identifikator, a p proizvoljan oblik; i uspešni oblici (originalno: *irrefutable patterns*), koji su oblika $\sim p$.

Svi operatori jezika su dati u tabeli, po opadajućem prioritetu i sa naznačenim vezivanjem (engl. *associativity*). Prikazana tabela je drugačije organizovana u odnosu na tabele operatora za ISWIM i SASL, zbog većeg broja operatora i njihovih vezivanja.

Levo	Bez	Desno
!, !!, //		.
		**, ^, ^^
*	%, /, 'div', 'mod', 'rem', 'quot'	
+, -	:+	
	\\	:, ++
	/=, <, <=, ==, >, >=, 'elem', 'notElem'	
		&&
		||
	:=	
		$

Sl. 3.15 Operatori Haskell-a

3.4.2 Pravila translacije

Dole navedena pravila podrazumevaju sintaksno ispravne (Haskell) izraze. Provera uskladjenosti tipova u Haskell programu se ovde ne prikazuje, jer je osnovni cilj prikazati tehniku implementacije Haskell-a posredstvom LL-a. Uskladjenost tipova se može proveriti na standardne načine na osnovu algoritama Robinson-a [1965] ili Milner-a [1978] (a opisanih i u [Field, Harrison, 1988] str. 143, [Hancock, 1987a;1987b], [Sokolowski, 1991] str. 145).

U navedenim pravilima translacije Haskell-a u LL, koristiće se sledeće oznake: i, i_0 i_1, ..., i_m za identifikatore jezika Haskell; p, p_1, ..., p_k za oblike jezika; t, t_1, ..., t_s za definiciju tipa podataka jezika Haskell; e, e', e_1, e'_1, ..., e_n, e'_n za proizvoljne izraze Haskell-a.

U opisu translacije će se koristiti i oznaka ϵ koja označava prazan LL izraz. Uvodimo i sledeće skupove:

- **Es(i)**, koji sadrži sve identifikatore koje modul pod nazivom **i** izvozi;
- **Cs**, koji sadrži podatke za translaciju korisničkih tipova podataka i sastoji se od trojki oblika **(i,k,n)**, pri čemu je **i** ime uvedenog konstruktora podataka, **k** je njegov redni broj pojavljivanja u deklaraciji kojom je uveden i **n** je njegova arnost, ukoliko je poznata tokom translacije;
- **Ts**, koji sadrži podatke za translaciju operatorskih klasa i objekata i sastoji se od četvorki oblika **(i,t,l,d)**, pri čemu je **i** ime (oznaka) operatora, **d** njegova translacija u LL-u u kojoj figurišu argumenti iz liste **l**, a koja će se uključiti u rezultujući LL program ako su argumenti originalnog Haskell izraza tipa **t**. Umesto oznake tipa **t**, četvorka može sadržati i nedefinisanu vrednost (u oznaci ?) u kom slučaju se definicija **d** primenjuje ukoliko za operator **i** i tip **t** ne postoji odgovarajuća četvorka;

- **Rs**, koji sadrži podatke o preimenovanju nekih identifikatora i čiji su elementi parovi oblika **(x,y)**, pri čemu je **x** novo, a **y** staro ime;
- **Os**, koji sadrži sve uvedene operatore jezika direktivama **infix**;
- **Us**, koji sadrži operatore Haskell-a koji su implementirani translacijom u pozive odgovarajućih funkcija LL-a.

Početna vrednost za sve ove skupove je ∅, osim za skup **Us** čiji elementi su {:, &&, ||, !!, ++, 'elem', //}. Sve operacije nad ovim skupovima će biti navodjene uz odgovarajuće pravilo za translaciju i biće oblika S' := S, pri čemu je S' nova sadržina skupa, a S stara sadržina skupa S.

Translacija Haskell-a će biti opisana funkcijom TT čiji su argumenti redom skup definicija Haskell programa i izraz na koji se date definicije primenjuju. Rezultat funkcije TT je odgovarajući s-izraz medjujezika LL. Funkcija TT će biti definisana uz pomoć sledećih funkcija:

- R priprema Haskell program za dalju translaciju, izbacujući deklaracije, transformišući odgovarajuće Haskell izraze i popunjavajući skupove potrebne za dalju translaciju. Funkcija R koristi sledeće pomoćne funkcije: I kreira pomoćne izraze oblika i = **_from** i_1 ili popunjava odgovarajuće skupove; C popunjava skup **Cs**.
- S translira klase i objekte Haskell programa,
- L translira bibliotečke module,
- T i T_F transliraju izraze Haskell programa,
- TQ translira ZF-izraze Haskell programa,
- M translira jednačine Haskell jezika,
- Z translira znake Haskell programa u znake medjujezika LL.

Početna transformacija programa

$$TT[\![\text{module Main (main) where } e_1;\ e_2;\ldots e_n;]\!]\,\text{main} = TT[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots\ R[\![e_n;]\!]]\!]\,\text{main}$$

$$TT[\![\text{module Main where } e_1;\ e_2;\ \ldots\ e_n;]\!]\,\text{main} = TT[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots\ R[\![e_n;]\!]]\!]\,\text{main}$$

$$TT[\![\text{module } i\ (e'_1,\ldots,e'_k) \text{ where } e_1;\ldots;e_n;]\!]\,\text{main} = L_i[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots\ R[\![e_n;]\!]]\!]\ ;\ Es(i) := Es(i) \cup \{e'_1,\ldots,e'_k\}$$

$$TT[\![\text{module } i \text{ where } e_1;\ldots;e_n;]\!]\,\text{main} = L_i[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots\ R[\![e_n;]\!]]\!]\ ;\ Es(i) := Es(i) \cup \{e'_1,\ldots,e'_k\}$$

pri čemu su $e'_1,\ldots,e'_k$ identifikatori definisani u modulu i

$$R[\![\text{import } i\ (e_1,\ldots,e_n);]\!] = I[\![T[\![e_1]\!]]\!]i\ \ldots\ I[\![T[\![e_n]\!]]\!]i$$

$$R[\![\text{import } i \text{ hiding } (e_1,\ldots,e_n);]\!] = I[\![T[\![e'_1]\!]]\!]i\ \ldots\ I[\![T[\![e'_k]\!]]\!]i$$

pri čemu $e'_1,\ldots,e'_k \in Es(i)\backslash\{e_1,\ldots,e_n\}$

$$R[\![\text{import } i;]\!] = I[\![T[\![e'_1]\!]]\!]i\ \ldots\ I[\![T[\![e'_k]\!]]\!]i$$

pri čemu $e'_1,\ldots,e'_k \in$ Es(i)

R⟦ import i ($e'_1,\ldots,e'_n$) renaming (e'_i to e_i, …,e'_j to e_j); ⟧
= I⟦ T⟦ e'_1 ⟧ ⟧i … I⟦ T⟦ e'_n ⟧ ⟧i, $1 \le i \le j \le n$
; Rs := Rs ∪ {(e_i,e'_i), …, (e_j,e'_j)}

R⟦ import i hiding ($e_1,\ldots,e_n$) renaming (e'_i to e_i, …,e'_j to e_j); ⟧
= I⟦ T⟦ e'_1 ⟧ ⟧i … I⟦ T⟦ e'_k ⟧ ⟧i, $1 \le i \le j \le n$,
pri čemu $e'_1,\ldots,e'_k \in$ Es(i)\{$e_1,\ldots,e_n$}
; Rs := Rs ∪ {(e_i,e'_i), …, (e_j,e'_j)}

R⟦ import i renaming (e'_i to e_i, …,e'_j to e_j); ⟧
= I⟦ T⟦ e'_1 ⟧ ⟧i … I⟦ T⟦ e'_k ⟧ ⟧i, $1 \le i \le j \le n$
pri čemu $e'_1,\ldots,e'_k \in$ Es(i)
; Rs := Rs ∪ {(e_i,e'_i), …, (e_j,e'_j)}

I⟦ i_1 ⟧i = i_1 = _from i

I⟦ (i_1) ⟧i = i_1 = _from i

I⟦ $i_0(i_1,\ldots,i_n)$ ⟧i = ϵ
; Cs := Cs ∪ {(i_1,1,?),…,(i_n,n,?)}

I⟦ i_0(..) ⟧i = ϵ
; Cs := Cs ∪ {(i_1,1,?),…,(i_n,n,?)}

pri čemu su $i_1,\ldots,i_n$ svi konstruktori tipa podataka i

R⟦ infixl n $i_1,\ldots,i_n$; ⟧ = ϵ ; Os := Os ∪ {T⟦ i_1 ⟧,…,T⟦ i_n ⟧}

R⟦ infixr n $i_1,\ldots,i_n$; ⟧ = ϵ ; Os := Os ∪ {T⟦ i_1 ⟧,…,T⟦ i_n ⟧}

R⟦ infix n $i_1,\ldots,i_n$; ⟧ = ϵ ; Os := Os ∪ {T⟦ i_1 ⟧,…,T⟦ i_n ⟧}

R⟦ type t i_1 … i_n = t; ⟧ = ϵ

R⟦ default $t_1,\ldots,t_n$ = t; ⟧ = ϵ

R⟦ data ($e_1,\ldots,e_k$) => i i_1 … i_n = e_1 | e_2 | … | e_n e; ⟧
= R⟦ data i i_1 … i_n = e_1 | e_2 | … | e_n e; ⟧

R⟦ data i i_1 … i_n = e_1 | e_2 |…| e_n; ⟧ = C⟦ T⟦ e_1 ⟧ ⟧1 … C⟦ T⟦ e_n ⟧ ⟧n

R⟦ data i i_1 … i_n = e_1 | e_2 |…| e_n deriving (c_1, c_2,…, c_k) ; ⟧
= C⟦ T⟦ e_1 ⟧ ⟧1 … C⟦ T⟦ e_n ⟧ ⟧n
R⟦ instance c_1 i where w_1^1;…;$w_{m,1}^{\ 1}$; ⟧
R⟦ instance c_2 i where w_1^2;…;$w_{m,2}^{\ 2}$; ⟧
….
R⟦ instance c_k i where w_1^k;…;$w_{m,k}^{\ k}$; ⟧

pri čemu su izrazi $w_{ij}^{\ j}$, i=1,…,m; j=1,…,k nastali na osnovu definicije konstruktora podataka i. Način automatskog kreiranja izraza w, kao i uslovi pod kojima je to moguće, je opisan u [Hudak, Peyton Jones, Wadler, 1992] str. 145. Kako je definicija izvedenih objekata vezano za programiranje u Haskell-u, u ovoj tezi neće biti dalje objašnjavano.

C⟦ i ⟧k = ϵ ; Cs := Cs ∪ {(i,k,0)}

C⟦ i i_1 … i_n ⟧k = ϵ ; Cs := Cs ∪ {(i,k,n)}

R⟦ class ($e_1,\ldots,e_k$) => e; ⟧ = R⟦ class e; ⟧

$R[\![\text{class } e;]\!] = \epsilon$
$R[\![\text{class } class \text{ where } e_1;\ldots;e_n;]\!] = S_?[\![R[\![e_1;]\!] \ldots R[\![e_n;]\!]]\!]$
$R[\![\text{instance } (e_1,\ldots,e_k) => c\ t;]\!] = R[\![\text{instance } c\ t;]\!]$
$R[\![\text{instance } c\ t;]\!] = \epsilon$
$R[\![\text{instance } c\ t \text{ where } e_1;\ldots;e_n;]\!] = S_t[\![R[\![e_1;]\!] \ldots R[\![e_n;]\!]]\!]$
$R[\![i_1,i_2,\ldots,i_n :: (e_1,\ldots,e_k) => t;]\!] = \epsilon$
$R[\![i_1,i_2,\ldots,i_n :: t;]\!] = \epsilon$

$R[\![e_1\ `i`\ e_2]\!] = i\ e_1\ e_2$
$R[\![e_1\ i\ e_2]\!] = i\ e_1\ e_2 \quad \text{ako } i \in Os$
$R[\![(i)\ e_1\ e_2]\!] = i\ e_1\ e_2 \quad \text{ako } i \in Os$
$R[\![(e)]\!] = (R[\![e]\!])$
$R[\![e]\!] = e$

Funkcija TT

$$TT[\![i_1\ e_1^1\ e_1^2 \ldots e_1^n\ m_1;\ \ldots;\ i_m\ e_m^1\ e_m^2 \ldots e_m^r\ m_m;]\!]e = (\_\text{letrec } e\ (i'_1\ .\ T[\![f_1]\!])\ (i'_2\ .\ T[\![f_2]\!]) \ldots (i'_k\ .\ T[\![f_k]\!]))$$

pri čemu izrazi e_i^j, i=1,...,m; j=1,...,n mogu biti i prazni. Identifikatori $i'_1,\ldots i'_k$ predstavljaju klase originalnih identifikatora $i_1,\ldots,i_m$ tako da svaki i_j', j=1,...,k predstavlja medjusobno jednake identifikatore iz skupa $\{i_1,\ldots,i_m\}$, tj. $i_j' = \{i_l, i_{l+1}, \ldots, i_{l+l'}\}$, tako da $i_l = i_{l+1} = \ldots = i_{l+l'}$ i $i_l \neq i_{l-1}$ i $i_{l+l'} \neq i_{l+l'+1}$.

Izrazi f_j, j=1,...,k predstavljaju skupove definicija za identifikatore i'_j. U ovim skupovima definicija ne figurišu identifikatori.

$$TT\ [T[\![p]\!]\ T[\![m]\!]]\ e = TT\ [\![p'=e_1]\!]\ e = (\_\text{let } M \begin{pmatrix} [y] \\ [(\_\text{quote } \_\text{true})] \\ [\tilde{}p' = e] \\ [(\_\text{error}(\_\text{quotegreska:case}))] \end{pmatrix} (y\ .\ e_1))$$

pri čemu je p oblik Haskell jezika (a nije identifikator), a y je identifikator koji se ne pojavljuje slobodan u izrazima e i m.

$$TT\begin{bmatrix} e_1\ e_1^1\ e_1^2\ \ldots\ e_1^n\ m_1; \\ \ldots \\ e_m\ e_m^1\ e_m^2\ \ldots\ e_m^r\ m_m; \end{bmatrix} e = \begin{array}{l}(\_\text{let } M \begin{pmatrix} [y] \\ [(\_\text{quote\_true})] \\ [\sim T[\![(\sim i_1', \ldots, \sim i_k')]\!] = e] \\ [(\_\text{error}(\_\text{quote greska:case}))] \end{pmatrix} \\ \quad (y\ .\ T[\![(f_1, \ldots, f_k)]\!]) \\ )\end{array}$$

pri čemu je bar jedan od izraza e_i, i=1,...,m oblik Haskell jezika osim identifikatora a i_j' i f_j, j=1,...,k imaju već uvedeno značenje.

Funkcija S

$$S_t[\![i_1\ e_1^1\ e_1^2\ \ldots\ e_1^n\ m_1;\ \ldots;\ i_m\ e_m^1\ e_m^2\ \ldots\ e_m^r\ m_m;]\!] = \epsilon$$
$$; Ts := Ts \cup (i'_1, t, (x_1,\ldots,x_{m,1}), T_F[\![f_1]\!])$$
$$\cup (i'_2, t, (x_1,\ldots,x_{m,2}), T_F[\![f_2]\!])$$
$$\ldots.$$
$$\cup (i'_k, t, (x_1,\ldots,x_{m,k}), T_F[\![f_k]\!])$$

pri čemu izrazi e_i^j, i=1,...,m; j=1,...,n mogu biti i prazni. Identifikatori $i'_1,\ldots i'_k$ i izrazi f_j, j=1,...,k su uvedeni na isti način kao u definiciji funkcije TT.

Funkcija L

$$L_i[\![i_1\ e_1^1\ e_1^2\ \ldots\ e_1^n\ m_1;\ \ldots;\ i_m\ e_m^1\ e_m^2\ \ldots\ e_m^r\ m_m;]\!]$$
$$= \_\text{libfun } i'_1\ T[\![f_1]\!]\ \ldots\ \_\text{libfun } i'_k\ T[\![f_k]\!]$$

ako definicija identifikatora $i'_1,\ldots, i'_k$ ne zavisi od definicija drugih identifikatora.

$$L_i[\![i_1\ e_1^1\ e_1^2\ \ldots\ e_1^n\ m_1;\ \ldots;\ i_m\ e_m^1\ e_m^2\ \ldots\ e_m^r\ m_m;]\!]$$
$$= \_\text{libfun } i_j'\ (\_\text{lambda } (y_1)\ \ldots\ (\_\text{lambda}(y_{n'})\ (\_\text{letrec}\ T_F[\![f_j]\!]\ (i_{j,1}'\ .\ T_F[\![f_{j,1}]\!])\ \ldots\ (i_{j,k}'\ .\ T_F[\![f_{j,k}]\!]))))$$

za svaki identifikator i_j' koji zavisi od definicije identifikatora $i_{j,1}\ldots i_{j,k}$ i čija arnost je n'. U gornjim definicijama izrazi e_i^j, i=1,...,m; j=1,...,n mogu biti i prazni, a identifikatori $i'_1,\ldots i'_k$ i izrazi f_j, j=1,...,k su uvedeni na isti način kao u definiciji funkcije TT.

Funkcije T i T_F - grupe jednačina

$$T\begin{bmatrix} p_1^1 \ \ldots\ p_1^n\ m_1^n; \\ p_2^1 \ \ldots\ p_2^n\ m_2^n; \\ \ldots \\ p_m^1 \ \ldots\ p_m^n\ m_m^n; \end{bmatrix} = \begin{array}{l} (\_\text{lambda}\,(y_1) \\ \ (\_\text{lambda}\,(y_2) \\ \ \ \ldots \\ \ \ (\_\text{lambda}\,(y_n) \\ \ \ \ M\left(\begin{array}{l} [y_1,\ldots,y_n] \\ [(\_\text{quote\_true}),\ldots,(\_\text{quote\_true})] \\ \begin{bmatrix} T[\![p_1^1]\!]\ \ldots\ T[\![p_1^n]\!]\ T[\![m_1^n]\!] \\ T[\![p_2^1]\!]\ \ldots\ T[\![p_2^n]\!]\ T[\![m_2^n]\!] \\ \ldots \\ T[\![p_m^1]\!]\ \ldots\ T[\![p_m^n]\!]\ T[\![m_m^n]\!] \end{bmatrix} \\ [(\_\text{error}\ (\_\text{quote greska:case}))] \end{array}\right) \\ ))) \end{array}$$

pri čemu su y_1, y_2, ..., y_n identifikatori koji se ne pojavljuju slobodni u m_i, i=1,...,m, a izrazi m_i će biti definisani nešto kasnije.

$T[\![i_1\ i_2\ \ldots\ i_n\ m]\!]$ = (_lambda (i_1) (_lambda (i_2)...(_lambda (i_n) $T[\![m]\!]$)...))

ako su svi i_j, j=1,...,n identifikatori i ako je to jedina jednačina.

Definicija funkcije T_F je identična definiciji funkcije T, osim što u definicije funkcije T_F ne figurišu uvodni **_lambda** izrazi.

Pri translaciji jednačina Haskell programa se koristi translaciona funkcija M koja ima iste argumente kao i za translaciju SASL programa, ali se od nje razlikuje po definiciji.

Funkcija T - izrazi m_i

U navodjenju pravila za izraze m_i će se koristiti strelica umesto uobičajenog znaka jednakosti zbog veće preglednosti, jer je znak jednakosti deo izraza m_i.

$T[\![m]\!] \longrightarrow T[\![= e]\!] \longrightarrow = T[\![R[\![e]\!]]\!]$

$T[\![m]\!] \longrightarrow T[\![| g_1 = e_1;\ | g_2 = e_2;\ \ldots;\ | g_k = e_k]\!]$

$\longrightarrow$ = (_if $T[\![R[\![g_1]\!]]\!]$ $T[\![R[\![e_1]\!]]\!]$
(_if $T[\![R[\![g_2]\!]]\!]$ $T[\![R[\![e_2]\!]]\!]$ ...
(_if $T[\![R[\![g_k]\!]]\!]$ $T[\![R[\![e_k]\!]]\!]$ (_error (_quote Greska))) ...))

$T[\![m]\!] \longrightarrow T[\![m$ where $\{e_1;e_2;\ldots;e_n\}]\!]$

$\longrightarrow$ = $TT[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots; R[\![e_n;]\!]]\!] m$

Funkcija T - izrazi

$T[\![\backslash i_1,\ldots,i_n \text{->} e]\!]$ = (_lambda (i_1) ... (_lambda (i_n) $T[\![e]\!]$) ...)

$$
\begin{aligned}
T[\![\backslash\ p_1\ p_2\ \ldots\ p_n \rightarrow e]\!] = {} & (\_\text{lambda}\,(y_1)\\
& (\_\text{lambda}\,(y_2)\\
& \quad \ldots\\
& (\_\text{lambda}\,(y_n)\\
& M\begin{pmatrix}[y_1,\ldots,y_n]\\ [(\_\text{quote\_true})]\\ [\![\, T[\![p_1]\!]\ T[\![p_2]\!]\ \ldots\ T[\![p_n]\!] = T[\![e]\!]\,]\!]\\ [(\_\text{error}\ (\_\text{quote greska:case}))]\end{pmatrix}\\
&) \ldots)
\end{aligned}
$$

pri čemu su y_i i=1,...,n identifikatori koji nisu slobodni u izrazu e).

$$
T\begin{bmatrix}\text{case } e \text{ of } & \{p_1\ m_1\\ & p_2\ m_2\\ & \ldots\\ & p_n\ m_n\}\end{bmatrix} = (\_\text{let}\ M\begin{pmatrix}[y]\\ [(\_\text{quote\_true}),\ldots,(\_\text{quote\_true})]\\ \begin{bmatrix}T[\![p_1]\!]\ T[\![m_1]\!]\\ T[\![p_2]\!]\ T[\![m_2]\!]\\ \ldots\ \ldots\\ T[\![p_n]\!]\ T[\![m_n]\!]\end{bmatrix}\\ [(\_\text{error}\ (\_\text{quote greska:case}))]\end{pmatrix} (y\ .\ T[\![e]\!])\ )
$$

pri čemu y nije slobodno u izrazima p_i i e_i, i=1,...,n.

$T[\![\text{let } \{e_1, e_2,\ldots,e_n\} \text{ in } e]\!] = TT[\![R[\![e_1;]\!]\ R[\![e_2;]\!]\ \ldots;\ R[\![e_n;]\!]\,]\!]T[\![e]\!]$

$T[\![-e]\!] = (\_\text{sub}\ (\_\text{quote}\ 0)\ T[\![e]\!])$

$T[\![(i\ e)]\!] = (\_\text{lambda}\ (x)\ (T[\![i]\!]\ x\ T[\![e]\!]))$
$T[\![(e\ i)]\!] = (\_\text{lambda}\ (x)\ (T[\![i]\!]\ T[\![e]\!]\ x))$

ako $i \in Os \cup Us$ i pri čemu je x idntifikator koji nije slobodan u e

$T[\![\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\!] = (\_\text{if}\ T[\![e_1]\!]\ T[\![e_2]\!]\ T[\![e_3]\!])$
$T[\![[e_1,\ldots,e_n]]\!] = (\_\text{cons}\ T[\![e_1]\!]\ \ldots\ (\_\text{cons}\ T[\![e_n]\!]\ \_\text{nil})\ \ldots\ )$
$T[\![[\,]]\!] = \_\text{nil}$
$T[\![(e_1,\ldots,e_n)]\!] = (\_\text{tuple}\ n\ 0\ T[\![e_1]\!]\ \ldots\ T[\![e_n]\!])$
$T[\![(\,)]\!] = (\_\text{tuple}\ 0\ 0)$
$T[\![(e)]\!] = (T[\![e]\!])$

Funkcija TQ - ZF-izrazi

$T[\![[e \mid q]]\!] = TQ[\![[e \mid q]]\!][]$

pri čemu oznaka *q* predstavlja listu. Svaki element liste je ili (logički) izraz ***e*** ili izraz oblika ***p <- e***.

```
                                (_letrec(h T[[e1]])
                                  (h . (_lambda(us)
                                    (_case us
                                      (_nil . T[[L]])
                                      (_cons . (_let
                                                                    ⎛ [y]                                    ⎞
TQ[[[e | p<-e1, q]]]L =                                  (_lambda(y) M ⎜ [(_quote_true)]                        ⎟ )
                                                                    ⎜ [[T[[p]] = TQ[[[e | q]]](h us')]]    ⎟
                                                                    ⎝ [(h us')]                              ⎠
                                          (u . (_car us))
                                          (us' . (_cdr us))
                                      )
                                )) ))
```

pri čemu se identifikatori *y*, *h*, *u*, *us*, *us'* ne pojavljuju slobodni u izrazima *p*, *q*, *e* i e_1.

$$TQ[\![[e \mid e_1, q]]\!]L = (\_\text{if } T[\![e_1]\!]\ TQ[\![[e \mid q]]\!]L\ T[\![L]\!])$$
$$TQ[\![[e \mid]]\!]L = (\_\text{cons } T[\![e]\!]\ T[\![L]\!])$$

Funkcija T - primene funkcija, operatori, identifikatori i konstante

$T[\![i@p]\!] = x@T[\![p]\!]$
$T[\![\_]\!] = \_$
$T[\![\sim p]\!] = \sim T[\![p]\!]$
$T[\![e\ //\ [e_1 := e_2]]\!] = (\_\text{update } T[\![e]\!]\ T[\![e_1]\!]\ T[\![e_2]\!])$
$T[\![e_1\ `i`\ e_2]\!] = ((T[\![i]\!]\ T[\![e_1]\!])\ T[\![e_2]\!])$
$T[\![e_1\ i\ e_2]\!] = ((T[\![i]\!]\ T[\![e_1]\!])\ T[\![e_2]\!])$,$i \in$ Os
$T[\![(i)\ e_1\ e_2]\!] = ((T[\![i]\!]\ T[\![e_1]\!])\ T[\![e_2]\!])$,$i \in$ Os
$T[\![e_1\ i\ e_2]\!] = (T[\![i]\!]\ T[\![e_1]\!]\ T[\![e_2]\!])$,$i \in$ Us
$T[\![(i)\ e_1\ e_2]\!] = (T[\![i]\!]\ T[\![e_1]\!]\ T[\![e_2]\!])$,$i \in$ Us
$T[\![i\ i_1\ i_2\ \ldots\ i_n]\!] = (\_\text{tuple } n\ i'\text{-}1\ T[\![i_1]\!]\ T[\![i_2]\!]\ \ldots\ T[\![i_n]\!])$

ako $(T[\![i]\!], i', n) \in$ Cs ili $(T[\![i]\!], i', ?) \in$ Cs. U ovom drugom slučaju je još potrebno, zbog efikasnosti u daljem transliranju, učiniti i Cs := Cs \ $\{(T[\![i]\!], i', ?)\} \cup \{(T[\![i]\!], i', n)\}$.

$$T[\![i\ i_1\ i_2\ \ldots\ i_n]\!] = d[i_1/x_1, i_2/x_2, \ldots, i_n/x_n]$$

ako $(T[\![i]\!], t, (x_1, \ldots x_n), d) \in$ Ts i ako su i_j, j=1,...,n tipa *t*.

$$T[\![i\ i_1\ i_2\ \ldots\ i_n]\!] = d[i_1/x_1, i_2/x_2, \ldots, i_n/x_n]$$

ako $(T[\![i]\!], ?, (x_1, \ldots x_n), d) \in$ Ts.

$T[\![i\ i_1\ i_2\ \ldots\ i_n]\!]$ = $(T[\![i]\!]\ T[\![i_1]\!]\ T[\![i_2]\!]\ \ldots\ T[\![i_n]\!])$

$T[\![$ negate $e]\!]$ = (_sub (_quote 0) $T[\![e]\!]$)
$T[\![$ primQuotRem $e_1\ e_2]\!]$ = (_tuple 2 0 (_div $T[\![e_1]\!]\ T[\![e_2]\!]$)
(_mod $T[\![e_1]\!]\ T[\![e_2]\!]$))
$T[\![$ elem $e_1\ e_2]\!]$ = (_member $T[\![e_1]\!]\ T[\![e_2]\!]$)
$T[\![$ drop $e_1\ e_2]\!]$ = (_rest $T[\![e_1]\!]\ T[\![e_2]\!]$)
$T[\![$ primArrayCr $(e_1,e_2)\ e]\!]$ = (_array (_sub $T[\![e_2]\!]\ T[\![e_1]\!]$) $T[\![e]\!]$)
$T[\![$ primArrayIndex $e\ e_1]\!]$ = (_index $T[\![e]\!]\ T[\![e_1]\!]$)
$T[\![i\ i_1\ \ldots\ i_n]\!]$ = $(\ldots(T[\![i]\!]\ T[\![i_1]\!])\ldots T[\![i_n]\!])$
$T[\![$ True $]\!]$ = (_quote _true)
$T[\![$ False $]\!]$ = (_quote _false)
$T[\![$ otherwise $]\!]$ = (_quote _true)
$T[\![$ "$c_1 \ldots c_n$" $]\!]$ = (_quote (_cons $Z[\![c_1]\!]$ ... (_cons $Z[\![c_n]\!]$ _nil) ...))
$T[\![$ '$c_1 \ldots c_n$' $]\!]$ = (_quote $Z[\![c_1]\!] \ldots Z[\![c_n]\!]$)
$T[\![$ '$c]\!]$ = (_quote $Z[\![c]\!]$)
$T[\![n]\!]$ = (_quote n) , n je broj

$T[\![$ && $]\!]$	= _and	$T[\![$ || $]\!]$	= _or
$T[\![$ not $]\!]$	= _not	$T[\![$!! $]\!]$	= _nth
$T[\![$ ++ $]\!]$	= _append	$T[\![$: $]\!]$	= _cons
$T[\![$ head $]\!]$	= _car	$T[\![$ tail $]\!]$	= _cdr
$T[\![$ error $]\!]$	= _error	$T[\![$ genericLength $]\!]$	= _len
$T[\![$ primMinInt $]\!]$	= _minInt	$T[\![$ primMaxInt $]\!]$	= _maxInt
$T[\![$ primIntToChar $]\!]$	= _chr	$T[\![$ primCharToInt $]\!]$	= _ord
$T[\![$ primEq $]\!]$	= _eqNum	$T[\![$ primLe $]\!]$	= _leqNum
$T[\![$ primLess $]\!]$	= _le	$T[\![$ primPlus $]\!]$	= _add
$T[\![$ primSub $]\!]$	= _sub	$T[\![$ primMul $]\!]$	= _mul
$T[\![$ primDiv $]\!]$	= _quo	$T[\![$ primSin $]\!]$	= _sin
$T[\![$ primCos $]\!]$	= _cos	$T[\![$ primExp $]\!]$	= _exp
$T[\![$ primSinh $]\!]$	= _sinH	$T[\![$ primCosh $]\!]$	= _cosH
$T[\![$ primLog $]\!]$	= _log	$T[\![$ primAtan $]\!]$	= _arcTan
$T[\![$ primAtanH $]\!]$	= _arcTanH	$T[\![$ AppendChan $]\!]$	= _appChan
$T[\![$ AppendFile $]\!]$	= _appFile	$T[\![$ AppenBinChan $]\!]$	= _appBChan
$T[\![$ AppendBinFile $]\!]$	= _appBFile	$T[\![$ DeleteFile $]\!]$	= _deleteFile
$T[\![$ WriteFile $]\!]$	= _writeFile	$T[\![$ WriteBinFile $]\!]$	= _writeBFile
$T[\![$ ReadChan $]\!]$	= _readChan	$T[\![$ ReadBinChan $]\!]$	= _readBChan
$T[\![$ ReadFile $]\!]$	= _readFile	$T[\![$ ReadBinFile $]\!]$	= _readBFile
$T[\![$ Failure $]\!]$	= _failure	$T[\![$ Success $]\!]$	= _success
$T[\![$ Str $]\!]$	= _res	$T[\![$ Bn $]\!]$	= _res

$T[\![$ primPi $]\!]$ = (_quote 3.1415926536)
$T[\![i]\!]$ = i' , ako $(i,i') \in$ Rs
$T[\![i]\!]$ = i , ako $(i,i') \notin$ Rs

Funkcija Z - znaci

$Z[\![\backslash xn]\!]$ = #xn , ako je n heksadecimalni broj

$Z[\![\backslash n]\!]$ = #n

$Z[\![\backslash s]\!]$ = odgovarajući specijalni znak predstavljen u LL-u, ako je s neki od sekvenci Haskell-a za opisivanje specijalnih znaka opisan u sintaksi jezika. Konretna pravila se zbog brojnosti izostavljaju.

$Z[\![c]\!]$ = c , u ostalim slučajevima

Funkcija M - translacija jednačina

Pri definisanju funkcije M, koristiće se i notacija E[x/y] koja označava LL izraz E u kome su sve pojave identifikatora y zamenjene izrazom x, i oznake q_j^i i h_j, j=1,...,m, i=1,...,n; koje zamenjuju redom $T[\![p_j^i]\!]$ i $T[\![m_j]\!]$.

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \begin{bmatrix} q_1^i & \ldots & q_1^{i+k} = h_1\\ & \ldots & \\ x@q_j^i & \ldots & q_j^{i+k} = h_j\\ & \ldots & \\ q_m^i & \ldots & q_m^{i+k} = h_m \end{bmatrix}\\ E\end{pmatrix} = M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \begin{bmatrix} q_1^i & \ldots & q_1^{i+k} = & h_1\\ & \ldots & & \\ q_j^i & \ldots & q_j^{i+k} = & (\_\text{let } h_j(x\ .\ y_i))\\ & \ldots & & \\ q_m^i & \ldots & q_m^{i+k} = & h_m \end{bmatrix}\\ E\end{pmatrix}$$

za svako $1 \le j \le m$ za koje je oblik $x@q_j^i$.

$$M\begin{pmatrix}[y_i,\ldots,y_{i+k}]\\ [\ldots,u_v,\ldots,u_c,\ldots,u_a,\ldots,u_r,\ldots,u_s,\ldots]\\ \begin{bmatrix} & \ldots & & \\ q_v^i & \ldots & q_v^{i+k} = & h_v\\ & \ldots & & \\ q_c^i & \ldots & q_c^{i+k} = & h_c\\ & \ldots & & \\ \_ & \ldots & q_a^{i+k} = & h_a\\ & \ldots & & \\ \sim q_r^i & \ldots & q_r^{i+k} = & h_r\\ & \ldots & & \\ x+c & \ldots & q_s^{i+k} = & s_m\\ & \ldots & & \end{bmatrix}\\ E\end{pmatrix} = M\begin{pmatrix}[y_{i+1},\ldots,y_{i+k}]\\ [\ldots,u_v,\ldots,(\_\text{and}(\_\text{eq } y_i\ q_c^i)\ u_c),\ldots,u_a,\ldots,u_r,\\ \quad\ldots,(\_\text{and}(\_\text{leq } c\ y_i)\ u_s),\ldots]\\ \begin{bmatrix} & \ldots & & \\ q_v^{i+1} & \ldots & q_v^{i+k} = & h_v[y_1/q_v^i]\\ & \ldots & & \\ q_c^{i+1} & \ldots & q_c^{i+k} = & h_c[y_1/q_c^i]\\ & \ldots & & \\ q_a^{i+1} & \ldots & q_a^{i+k} = & h_a\\ & \ldots & & \\ q_r^{i+1} & \ldots & q_r^{i+k} = & h'_r\\ & \ldots & & \\ q_s^{i+1} & \ldots & q_s^{i+k} = & (\_\text{let } h_s[x'/x]\ (x'\ .\ y_1\text{-}c)) \\ & \ldots & & \end{bmatrix}\\ E\end{pmatrix}$$

ako su svi oblici q_j^i, j=1,...,m identifikatori, konstante, anonimni identifikatori, uspešni oblici i oblici $x+c$ (c je konstanta); i neka su oblici q_v^i, $1 \le v \le m$

identifikatori, q_c^i $1 \le c \le m$ konstante, q_a^i $1 \le a \le m$ anonimni identifikatori (_), q_r^i $1 \le r \le m$ uspešni oblici ($\sim p$) i q_s^i $1 \le s \le m$ oblici $x+c$. x' je novi identifikator. h_r' je sledećeg oblika:

$$
\begin{aligned}
h'_r = \quad & (\_\text{letrec } h_r[x'_1/x_1, \ldots, x'_l/x_l] \\
& \quad (x'_1 \;.\; M \begin{pmatrix} [y_i] \\ [u_r] \\ [\![q_r^i = x_1]\!] \\ E \end{pmatrix}) \\
& \quad (x'_2 \;.\; M \begin{pmatrix} [y_i] \\ [u_r] \\ [\![q_r^i = x_2]\!] \\ E \end{pmatrix}) \\
& \quad \ldots \\
& \quad (x'_l \;.\; M \begin{pmatrix} [y_i] \\ [u_r] \\ [\![q_r^i = x_l]\!] \\ E \end{pmatrix}) \\
&)
\end{aligned}
$$

x'_i, i=1,...,l su svi identifikatori koji se nalaze unutar oblika q_r^i, a x_i' su novi identifikatori.

$$
\begin{aligned}
& M \begin{pmatrix} [y_i, \ldots, y_{i+k}] \\ [u_1, \ldots, u_j, \ldots, u_m] \\ \left[\!\!\left[\begin{array}{llll} q_1^i & \ldots & q_1^{i+k} & = h_1 \\ & \ldots & & \\ q_j^i & \ldots & q_j^{i+k} & = h_j \\ q_{j+1}^i & \ldots & q_{j+1}^{i+k} & = h_{j+1} \\ & \ldots & & \\ q_m^i & \ldots & q_m^{i+k} & = h_m \end{array} \right]\!\!\right] \\ E \end{pmatrix} = \\
& \quad (\_\text{case } y_i \\
& \qquad (\_\text{nil} \;.\; M \begin{pmatrix} [y_{i+1}, \ldots, y_{i+k}] \\ [u_1, \ldots, u_j] \\ \left[\!\!\left[\begin{array}{llll} q_1^{i+1} & \ldots & q_1^{i+k} & = h_1 \\ & \ldots & & \\ q_j^{i+1} & \ldots & q_j^{i+k} & = h_j \end{array} \right]\!\!\right] \\ E \end{pmatrix}) \\
& \qquad (\_\text{cons} \;.\; (\_\text{let } M \begin{pmatrix} [z_1, z_2, y_{i+1}, \ldots, y_{i+k}] \\ [u_{j+1}, \ldots, u_m] \\ \left[\!\!\left[\begin{array}{llll} a_{j+a} & b_{j+1} & q_{j+1}^{i+1} & \ldots \; q_{j+1}^{i+k} = h_{j+1} \\ & & & \ldots \\ a_m & b_m & q_m^{i+1} & \ldots \; q_m^{i+k} = h_m \end{array} \right]\!\!\right] \\ E \end{pmatrix} \\
& \qquad\qquad (z_1 \;.\; (\_\text{car } y_i)) \\
& \qquad\qquad (z_2 \;.\; (\_\text{cdr } y_i)) \\
& \qquad) \qquad) \\
&)
\end{aligned}
$$

ako su oblici q_1^i do q_j^i jednaki **_nil**, a oblici q_{j+1}^i do q_m^i jednaki (**_cons** a_k b_k), k=j+1,...,m, pri čemu su z_1 i z_2 novi identifikatori. Redosled jednačina u okviru jednog konstruktora se ne sme menjati.

$$
\mathrm{M}\left(\begin{array}{l}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \left[\begin{array}{l} q_1^{\,i}\ \ldots\ q_1^{\,i+k}=h_1\\ \quad\ldots\\ q_j^{\,i}\ \ldots\ q_j^{\,i+k}=h_j\\ \quad\ldots\\ q_m^{\,i}\ \ldots\ q_m^{\,i+k}=h_m\end{array}\right]\\ \mathrm{E}\end{array}\right) =
$$

$$
\begin{array}{l}(\_\mathrm{case}\ y_i\\ \quad(0\ .\ (\_\mathrm{let}\ \mathrm{M}\left(\begin{array}{l}[z_1,\ldots,z_{a_0},y_{i+1},\ldots,y_{i+k}]\\ [u_1,\ldots,u_{j_0}]\\ \left[\begin{array}{l} w_1^1\ \ldots\ w_1^{a_0}\ q_1^{i+1}\ \ldots\ q_1^{i+k}=h_1\\ \quad\ldots\\ w_{j_0}^1\ \ldots\ w_{j_0}^{a_0}\ q_{j_0}^{i+1}\ \ldots\ q_{j_0}^{i+k}=h_{j_0}\end{array}\right]\\ \mathrm{E}\end{array}\right)\\ \qquad(z_1\ .\ (\_\mathrm{select}\ y_i\ (\_\mathrm{quote}\ 1)))\\ \qquad\ldots\\ \qquad(z_{a_0}\ .\ (\_\mathrm{select}\ y_i\ (\_\mathrm{quote}\ a_0)))\\ \quad)\quad)\\ \quad\ldots\\ \quad(r\ .\ (\_\mathrm{let}\ \mathrm{M}\left(\begin{array}{l}[z_1,\ldots,z_{a_r},y_{i+1},\ldots,y_{i+k}]\\ [u_{j_{r-1}},\ldots,u_m]\\ \left[\begin{array}{l} w_{j_{r-1}}^1\ \ldots\ w_{j_{r-1}}^{a_r}\ q_{j_{r-1}}^{i+1}\ \ldots\ q_{j_{r-1}}^{i+k}=h_{j_{r-1}}\\ \quad\ldots\\ w_m^1\ \ldots\ w_m^{a_r}\ q_m^{i+1}\ \ldots\ q_m^{i+k}=h_m\end{array}\right]\\ \mathrm{E}\end{array}\right)\\ \qquad(z_1\ .\ (\_\mathrm{select}\ y_i\ (\_\mathrm{quote}\ 1)))\\ \qquad\ldots\\ \qquad(z_{a_r}\ .\ (\_\mathrm{select}\ y_i\ (\_\mathrm{quote}\ a_r)))\\ \quad)\quad)\\ )\end{array}
$$

ako su oblici $q_1^{\,i}$ do $q_m^{\,i}$ jednaki (_tuple a i x_1,...,x_a). Indeksima 1,...,j_0; ...; j_{r-1},...j_r(=m) su predstavljene sekvence označenih n-torki cija su oznake redom 0; ...;r. Oznake a_0,...,a_r redom predstavljaju arnosti označenih n-torki sa oznakama 0, ...,r. Redosled jednačina u okviru jednog konstruktora se ne sme menjati.

$$
\mathrm{M}\left(\begin{array}{l}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_j,\ldots,u_m]\\ \left[\begin{array}{l} q_1^{\,i}\ \ldots\ q_1^{\,i+k}=h_1\\ \quad\ldots\\ q_m^{\,i}\ \ldots\ q_m^{\,i+k}=h_m\end{array}\right]\\ \mathrm{E}\end{array}\right) = \mathrm{M}\left(\begin{array}{l}[y_i,\ldots,y_{i+k}]\\ [u_1,\ldots,u_{j_1}]\\ \left[\begin{array}{l} q_1^{\,i}\ \ldots\ q_1^{\,i+k}=h_1\\ \quad\ldots\\ q_{j_1}^{\,i}\ \ldots\ q_{j_1}^{\,i+k}=h_{j_1}\end{array}\right]\\ \mathrm{M}\left(\begin{array}{l}[y_i,\ldots,y_{i+k}]\\ [u_{j_1+1},\ldots,u_{j_2}]\\ \left[\begin{array}{l} q_{j_1+1}^{\,i}\ \ldots\ q_{j_1+1}^{\,i+k}=h_{j_1+1}\\ \quad\ldots\\ q_{j_2}^{\,i}\ \ldots\ q_{j_2}^{\,i+k}=h_{j_2}\end{array}\right]\\ \ldots\\ \mathrm{M}\left(\begin{array}{l}[y_i,\ldots,y_{i+k}]\\ [u_{j_{r-1}},\ldots,u_{j_r}]\\ \left[\begin{array}{l} q_{j_r-1}^{\,i}\ \ldots\ q_{j_r-1}^{\,i+k}=h_{j_r-1}\\ \quad\ldots\\ q_{j_r}^{\,i}\ \ldots\ q_{j_r}^{\,i+k}=h_{j_r}\end{array}\right]\\ \mathrm{E}\end{array}\right)\end{array}\right)\end{array}\right)
$$

ako grupe $q_1^{\,i}$ do $q_{j_1}^{\,i}$, $q_{j_1+1}^{\,i}$ do $q_{j_2}^{\,i}$, $q_{j_2+1}^{\,i}$ do $q_{j_3}^{\,i}$, ..., $q_{j_r-1}^{\,i}$ do $q_{j_r}^{\,i}$, pripadaju istoj vrsti oblika, a $1 \le i \le i+k \le n$, $j_r = m$. Iste vrste oblika su: a) identifikatori,

konstante, anonimni identifikatori, uspešni oblici i oblici forme $x+c$; b) konstruktor podataka _nil; c) konstruktor podataka _cons; d) označene n-torke sa oznakom 0; e) označene n-torke sa oznakom 1 itd.

Posle uzastopne primene gornjih pravila, grupa jednačina se svodi na jedan od sledeća dva slučaja:

$$M\begin{pmatrix} [y_i, \ldots, y_{i+k}] \\ [u_1, \ldots, u_m] \\ \begin{bmatrix} & = h_1 \\ \ldots & \\ & = h_m \end{bmatrix} \\ E \end{pmatrix} = \begin{array}{l} (\_\text{if } u_1\ h_1 \\ \quad (\_\text{if } u_2\ h_2 \\ \qquad \ldots \\ \qquad (\_\text{if } u_m\ h_m\ E) \\ \qquad \ldots \\ ) \ ) \end{array}$$

$$M\begin{pmatrix} [] \\ A \\ [] \\ E \end{pmatrix} = E$$

3.4.3 Komentari

Translacija Haskell programa se vrši u dva prolaza. U prvom prolazu se formiraju potrebni pomoćni skupovi, proverava uskladjenost tipova podataka i pojedini izrazi Haskell programa se transformišu i pripremaju za drugi prolaz. U drugom prolazu se Haskell program translira u odgovarajući LL program.

Translator Haskell-a u medjujezik LL podrazumeva da je svaki Haskell modul smešten u poseban fajl i da se naziv fajla i modula podudaraju. Takva odluka nije u suprotnosti sa definicijom jezika Haskell [Hudak, Peyton Jones, Wadler, 1992] str. 43.

Translaciona funkcija S služi za translaciju klasa i objekata Haskell operatora, formirajući skup Ts, koji se kasnije koristi u translaciji primene funkcija (operatora). Izborom pogodne strukture za predstavljanje skupa Ts, translacija ovih konstrukcija može biti veoma efikasno. Alternativne implementacije operatorskih klasa su predmet vrlo skorih istraživanja (na primer [Peterson, Jones, 1993]).

Rezultat translacije funkcije L_i se smešta u poseban fajl čije ime je oblika *i*.LLB. Prilikom odredjivanja zavisnosti funkcija, može se koristiti lista identifikatora koji se iz modula izvoze - ostale definicije treba da budu lokalne u definicijama identifikatora koji se izvoze.

case izraz Haskell jezika se ne translira direktno u uslovni izraz _case LL-a jer nemaju istu semantiku. Dok se unutar Haskell-ovog **case** izraza mogu nalaziti bilo kakvi oblici, unutar LL-ovog _case izraza se mogu nalaziti samo svi mogući konstruktori odredjenog tipa podataka i to u odredjenom redosledu.

Za pojedine Haskell izraze koji u sebi mogu da sadrže oblike, data su dva pravila translacije: jedan specijalni, za izraze u kome ne postoje oblici i drugi, opšti. Prvo pravilo gotovo uvek generiše efikasniji i brži kod.

Funkcija TQ je prilagodjenje Wadler-ove funkcije TQ [Wadler, 1987b] za translaciju ZF izraza. Prikazana je poslednja verzija ove funkcije koja je najmanje čitljiva, ali i najefikasnija. Manje efikasne, ali čitljivije i jasnije šeme translacije ZF izraza u prostiji (medju)jezik se mogu takodje pronaći u [Wadler, 1987b; Hudak, Peyton Jones, Wadler, 1992] str. 16.

Uvedeni konstruktori podataka se transliraju u označene n-torke medjujezika LL, sa oznakama od 0 do n-1, ako je n broj konstruktora podataka u okviru jednog tipa.

Na nekim mestima se operatori koji pripadaju skupu **Os** transliraju na različit način od operatora koji pripadaju skupu **Us**. Razlika se mora praviti jer se svi uvedeni operatori transliraju kao Curry-jeve funkcije (ugnježdenim anonimnim funkcijama), dok se neki (iz skupa **Us**) transliraju direktno u pozive odgovarajućih ugradjenih funkcija LL-a. Zbog toga mora postojati i razlika u njihovom pozivanju.

Pri translaciji Haskell programa, uglavnom se koristi translacija u **_letrec** izraz LL-a. Zbog toga je pre translacije LL programa u jezike apstraktnih mašina poželjno izvršiti analizu zavisnosti i transformisati LL program.

Pri translaciji primene funkcija potrebno je konsultovati skup **Cs**, da bi se utvrdilo da li se radi o primeni uvedenog konstruktora podataka ili pozivu funkcije. Uvedeni konstruktori podataka u Haskell-u moraju počinjati velikim slovom, te je dovoljno ispitivati samo prvo slovo identifikatora. Tek ako je ono veliko, treba potražiti odgovarajuće podatke u skupu **Cs**.

Pri translaciji izraza koji sadrže binarne operatore, treba voditi računa o načinu na koji vezuju svoje operande. Pored operatora koji vezuju "udesno" datih u tabeli na Sl. 3.15, u Haskell-u je moguće uvesti i korisničke operatore koji vezuju "udesno" direktivama **infixr**.

Podsetimo se da svi mogući konstruktori podataka izvornog funkcionalnog programskog jezika transliraju uvek u samo tri moguće konstrukcije LL-a: **_nil**, **_cons** i **_tuple**, te da zato funkcija M prepoznaje samo ta dva slučaja (**_nil** i **_cons**, odnosno familija ugradjene funkcije **_tuple** sa različitim oznakama). Podsetimo se takodje da zbog tipiziranosti jezika u kojima je moguće uvesti korisničke tipove podataka, nikad ne može doći do "mešanja" različitih oblika unutar LL izraza (**_nil** sa **_tuple**, recimo). Takodje uvek važi (iz istog razloga) da se familija označenih n-torki koja se u jednom trenutku nadje kao argument funkcije M, uvek odnosi samo na jedan uvedeni tip podataka i predstavlja sve njegove moguće konstruktore.

Haskell moduli ne moraju eksplicitno da uvoze identifikatore iz standardnih Haskell modula (svi čiji naziv počinje sa **Prelude**) - oni su implicitno uveženi. Translator Haskell-a u LL treba da sve primene funkcija iz standardnih Haskell biblioteka translira u odgovarajuće pozive bibliotečkih funkcija LL-a.

Haskell ima "dvodimenzionalnu" sintaksu, u kojoj je važan i izgled programa. Pravila zamene razdvajača {, } i ; prazninama, uvlačenjima redova i novim redovima, data su u [Hudak, Peyton Jones, Wadler, 1992] str. 3.

3.4.4 Izmene u standardnim bibliotekama Haskell-a

Veliki deo operatora i ugradjenih funkcija Haskell-a je implementiran u standardnim bibliotekama. U implementaciji Haskell-a predloženoj u ovoj tezi, potrebno je izvršiti neke izmene u standardnim bibliotekama.

U bibliotekama su izvršene samo minimalne izmene, poštujući sledeće dve odluke:

- svi operatori Haskell-a koji imaju svoj direktan analogon medju ugradjenim funkcijama LL-a se implementiraju direktnim transliranjem u LL, a njihove definicije se brišu iz standardnih biblioteka.
- ako za neki operator ne postoji direktan analogon medju ugradjenim funkcijama LL-a, onda se ostavlja njegova definicija iz standardne biblioteke, umesto da se taj operator izražava kompozicijom ugradjenih funkcija LL-a.

U daljem tekstu će biti opisane potrebne izmene u modulima.

Modul Prelude. Iz modula je potrebno izbaciti deklaracije za operatore && i ||, kao i izbaciti definicije logičkih operatora (&&, ||, **not** i **otherwise**), jer se direktno transliraju u primene ugradjenih funkcija LL-a.

Modul PreludeBuiltin. Ovaj modul sadrži deklaracije (tipova) ugradjenih funkcija jezika Haskell. Na Sl. 3.16 se nalaze deklarisane funkcije i akcije koje je potrebno preduzeti.

U prvom delu su navedeni identifikatori koje treba zameniti identifikatorima odgovarajućih (opštijih) ugradjenih funkcija, kako bi se brojevi u Haskell-u predstavili na način na koji je to usvojeno u LL-u. U drugom delu su navedeni identifikatori koje je potrebno dodati i u trećem se nalaze identifikatori koje je potrebno izbaciti. U svim narednim modulima je potebno izvršiti naznačene izmene, i to tako što će biti izbačene sve definicije u kojima figurišu neki od identifikatora koji se zamenjuju novim, a umesto njih se ubacuje jedna definicija sa novim identifikatorom. Definicije koje sadrže identifikatore koji se izbacuju se takodje izbacuju iz svih narednih biblioteka.

Modul PreludeInteger. Ovaj modul ne postoji u definiciji jezika [Hudak, Peyton Jones, Wadler, 1992] već su ugradjene funkcije za rad sa celim brojevima neograničene preciznosti direktno podržane: **primEqInteger**, **primLeInteger**, **primLessInteger**, **primPlusInteger**, **primSubInteger**, **primMulInteger**, **primNegInteger**, **primQuotRemInteger**, **primIntegerToInt**, **primIntToInteger**.

```
Izmeniti:
primEqInt        primEqFloat     primEqDouble          -> primEq
primLeInt        primLeFloat     primLeDouble          -> primLe
primPlusInt      primPlusFloat   primPlusDouble        -> primPlus
primMulInt       primMulFloat    primMulDouble         -> primMul
primQuotRemInt                                         -> primQuotRem
primDivFloat     primDivDouble                         -> primDiv
primSinFloat     primSinDouble                         -> primSin
primCosFloat     primCosDouble                         -> primCos
primExpFloat     primExpDouble                         -> primExp
primSinhFloat    primSinhDouble                        -> primSinh
primCoshFloat    primCoshDouble                        -> primCosh
primPiFloat      primPiDouble                          -> primPi
primLogFloat     primLogDouble                         -> primLog
primAtanFloat    primAtanDouble                        -> primAtan
primAtanhFloat   primAtanhDouble                       -> primAtanH

Dodati:
primLess         primSub         primArrayCr      primArrayIndex

Izbaciti:
primNegInt       primNegFloat    primNegDouble    primNullBin      primIsNullBin
primAppendBin    primSqrtFloat   primSqrtDouble   primTanFloat     primTanDouble
primTanhFloat    primTanhDouble  primAsinFloat    primAsinDouble   primAcosFloat
primAcosDouble   primAsinhFloat  primAsinhDouble  primAcoshFloat   primAcoshDouble
primFloatRadix   primDoubleRadix primFloatDigits  primDoubleDigits primFloatMinExp
primDoubleMinExp primFloatMaxExp primDoubleMaxExp primDecodeFloat  primDecodeDouble
primEncodeFloat  primEncodeDouble
```

Sl. 3.16 Izmene u bibliotekama Haskell-a

Kako LL ne poseduje funkcije za rad sa brojevima neograničene preciznosti, ove operacije je potrebno implementirati u posebnom modulu.

Modul PreludeCore. U ovom modulu se nalaze definicije klasa raznih operatora Haskell-a i odgovarajući operatorski objekti u zavisnosti od od tipova operanada na koji se primenjuju. Iz ovog modula je pored izmena nastalih usled izmena u modulu **PreludeBuiltin**, potrebno izvršiti i sledeće izmene:

- izostaviti definiciju tipa podataka **Bool**, jer se logičke vrednosti direktno transliraju u ugradjene logičke vrednosti LL-a.
- izostaviti iz definicije klase **Ord** podrazumevajući slučaj za operator <, jer se on direktno translira u odgovarajuću ugradjenu funkciju LL-a.
- izostaviti iz definicije klase **Num** podrazumevajući slučaj za operator -, jer se on direktno translira u poziv odgovarajuće ugradjene funkcije LL-a.
- u definiciju klase **Floating** dodati podrazumevajuće slučajeve za definicije funkcija **acos**, **asin**, **acosh** i **asinh**, posredstvom postojećih operatora.

PreludeRatio, PreludeComplex, PreludeText. Moduli sadrži definicije operatora i struktura podataka potrebnih za implementaciju redom, racionalnih brojeva, kompleksnih brojeva i tekstova. Pored izmena nastalih izmenom biblioteke **PreludeBuiltin** nikakve izmene u ovim modulima nisu potrebne.

PreludeList. U ovom modulu se nalaze definicije osnovnih operacija za rad sa listama. Pored izmena nastalih izmenom biblioteke **PreludeBuiltin**, iz ovog modula je potrebno izbaciti definicije sledećih operatora i funkcija, jer se one transliraju direktno u ugradjene funkcije LL-a: **head**, **tail**, **++**, **genericLength**, **!!**, **drop** i **elem**.

PreludeArray. Modul sadrži definicije osnovnih operacija za rad sa nizovima. Pored izmena nastalih izmenom biblioteke **PreludeBuiltin** umesto definicije funkcija i operatora **array** i **!** potrebno je uneti definicije prikazane na Sl. 3.17. Cilj uslova navedenog u definiciji funkcije **array** je da ispita da li su indeksi liste **ivs** unutar zadatog intervala i da li je element svakog indeksa definisan tačno jednom.

Definiciju operatora // treba izostaviti, jer se on translira direktno u poziv odgovarajuće ugradjene funkcije LL-a.

```
array b ivs =
   if (and [inRange b i | i:=_ <- ivs]) &&
      ([hd b .. tl b] == qs[i | _:=i <- ivs])
      then MkArray b primArrayCr b ivs
      else error "array{PreludeArray}: indexes"

(!) (MkArray _ a) n = primArrayIndex a n
```

Sl. 3.17 Izmene u modulu PreludeArray

PreludeIO. Modul sadrži definicije operatora i struktura podataka potrebnih za implementaciju ulazno/izlaznih operacija. Pored izmena nastalih izmenom biblioteke **PreludeBuiltin** iz ovog modula je potrebno izbaciti definicije tipova podataka **Requests** i **Responses**.

3.4.5 Primeri

U ovom odeljku su prikazani neki primeri programa pisanih u Haskell-u i odgovarajuće translacije u LL. Na Sl. 3.18 do Sl. 3.20 se nalaze redom definicije funkcija za izračunavanje znaka broja **x**, poravnavanje binarnog stabla i brzo sortiranje liste.

```
module Main where

  main :: Int
  sign :: Int -> Int

  main = sign 5
  sign x | x > 0       = 1
         | x == 0      = 0
         | otherwise   = -1
```

```
(_letrec main
  (main . (sign (_quote 5)))
  (sign . (_lambda (x)
        (_if (_leNum (_quote 0) x)
             (_quote 1)
             (_if (_eqNum x (_quote 0))
                  (_quote 0)
                  (_if (_quote _true)
                       (_quote -1)
                       (_error(_quoteGreska_case))
) )      )    )    )
```

Sl. 3.18 Funkcija za izracunavanje znaka argumenta (Haskell i LL)

```
Module Main where
  data Tree a =
         Leaf a |
         Branch (Tree a) (Tree a)

 main :: [a]
 fringe :: (Tree a) -> [a]

 main = fringe (Leaf 2)
 fringe (Leaf x) = [x]
 fringe (Branch l r) =
         fringe l ++ fringe r
```

```
(_letrec main
   (main . (fringe (_tuple (_quote 1)
                           (_quote 0)
                           (_quote 2))))
   (fringe . (_lambda (y)
     (case y
        (0 .(_let
               (_cons z1 _nil)
               (z1 . (_select y (_quote 1)))
        )  )
        (1 .(_let
             (_append (fringe z1) (fringe z2))
             (z1 . (_select y (_quote 1)))
             (z2 . (_select y (_quote 2)))
        )  )
) ) )
```

Sl. 3.19 Funkcija za poravnanje specijalnog binarnog stabla (Haskell i LL)

```
module Main where
 main :: [a]
 qs   :: [a] -> [a]

 main      = qs [2,1,5,7]
 qs []     = []
 qs (x:xs) =
     qs [y | y<-xs,y<x]++
     [x] ++
     qs [y | y<-xs,y>=x]
```

```
(_letrec main
  (main .(qs (_cons (_quote 2))
             (_cons (_quote 1))
               (_cons (_quote 5))
                 (_cons (_quote 7))
                   (_cons _nil))))))

 (qs .(_lambda (z)
   (_case z
     (_nil . _nil)
     (_cons .
       (_let
         (_append
           (qs (_letrec (h z2)
                 (h .(_lambda (us)
                       (_case us
                         (_nil . _nil)
                         (_cons .(_let
                                   (_lambda (yy)
                                     (_if (_leNum yy z1)
                                          (_cons yy (h us1))
                                          (h us1)))
                                   (u .(_car us))
                                   (us1 .(_cdr us))
           )   ) )   ) ) )
           (_append
             (_cons z1 _nil)
             (qs (_letrec (h z2)
                   (h .(_lambda (us)
                         (_case us
                           (_nil . _nil)
                           (_cons .(_let
                                     (_lambda (yy)
                                       (_if (_leNum yy z1)
                                            (_cons yy (h us1))
                                            (h us1)))
                                     (u .(_car us))
                                     (us1 .(_cdr us))
             )   )   )   ) )   ) ) )
           (z1 . (_car z))
           (z2 . (_cdr z))
)) ) ) )
```

Sl. 3.20 Funkcija za "brzo" sortiranje (engl. *quick-sort*) liste (Haskell i LL)

Glava 4

Prevodjenje LL-a u jezike apstraktnih mašina

U ovoj glavi su data pravila za prevodjenje programa medjujezika LL u mašinske jezike nekoliko karakterističnih apstraktnih mašina za implementaciju čistofunkcionalnih programskih jezika: SECD, lenja SECD mašina, SK redukciona mašina, superkombinatorska mašina i G-mašina. Veća pažnja je posvećena prevodjenju onih konstrukcija LL-a koji nisu uobičajeni u drugim čistofunkcionalnim programskim jezicima i medjujezicima.

Opisana su potrebna proširenja mašinskih jezika apstraktnih mašina. Neke od mogućih optimizacija su kratko diskutovane.

Na Sl. 4.1 se nalazi LL program čiji će prevod u mašinske jezike različitih apstraktnih mašina biti prikazan u narednim odeljcima, radi ilustracije.

```
(_letrec (part (_quote 10))
  (part _lambda (n)
     (par n n)
  )
  (par _lambda (m n)
    (_if (_eq n (_quote 1))
         (_quote 1)
         (_if (_eq m (_quote 1))
              (_quote 1)
              (_if (_le m n)
                   (par m m)
                   (_if (_eq m n)
                        (_add (par m (_sub m (_quote 1))) (_quote 1))
                        (_add (par m (_sub n (_quote 1))) (par (_sub m n) n))
) )      )    )    )
```

Sl. 4.1 Izračunavanje broja particija broja 10 u LL-u

4.1 SECD mašina

SECD mašina sadrži 4 registra: S, E, C i D. Stanje u kome se mašina nalazi je jednoznačno odredjeno sadržajem sva četiri registra. Početno stanje SECD mašine je sledeće: S sadrži operande izraza (programa) čija se vrednost izračunava, C sadrži mašinski program SECD mašine koji će izračunati vrednost (tj. prevod originalnog LL programa), dok su E i D prazni. Registri S, E, C i D su po svojoj strukturi stekovi. U daljem tekstu će se oznakom *a.R* označavati operacija stavljanja elementa *a* na vrh regista (steka) *R*. Prazan stek će biti označen oznakom **nil**. SECD mašina koja se opisuje u ovom odeljku je zasnovana na SECD mašini opisanoj u [Henderson, 1980]. Skup komandi mašinskog jezika je proširen da bi na odgovarajući način podržao sve ugradjene funkcije medjujezika LL.

U naredna dva odeljka se navode pravila za prevodjenje LL izraza u komande SECD mašine i pravila izvršavanja tih komandi. Veća pažnja će biti posvećena komandama koje su nove u odnosu na [Henderson, 1980], dok se detaljniji opis prvobitnih komandi (LD, LDC, LDF, LDE, AP, AP0, RTN, DUM, RAP, UPD, SEL, JOIN, CAR, CDR, ATOM, CONS, EQ, ADD, SUB, MUL, DIV, REM, LEQ, STOP) nalazi u [Henderson, 1980] str. 163; 219.

4.1.1 Prevodilac LL-a u jezik SECD mašine

Pravila za prevodjenje LL konstrukcija u mašinski jezik SECD mašine će biti data funkcijom P koja preslikava izraze medjujezika LL (na osnovu vrednosti identifikatora koji učestvuju u izrazu) u odgovarajuću sekvencu mašinskih komandi SECD mašine. Funkcija P će biti definisana navodjenjem pravila sledećeg oblika:

$$P[\![e]\!] n = e'$$

pri čemu je *e* sintaksna konstrukcija medjujezika LL, *n* je lista identifikatora (engl. *namelist*) koja sadrži vrednosti identifikatora koji učestvuju u izrazu *e*, a *e'* predstavlja rezultujuću sekvencu naredbi mašinskog jezika SECD mašine. Funkcija P koristi pomoćnu funkciju D, kojom je realizovana direktiva LL-a **_from**. Pretpostavićemo takodje da postoji funkcija ***loc(x,n)*** koja pronalazi vrednost identifikatora *x* unutar liste imena *n* i funkcija ***find(x,l)*** čija je vrednost s-izraz koji sadrži definiciju identifikatora *x* iz biblioteke (fajla) *l*. Lista imena *n* se takodje predstavlja stekom i koriste se iste oznake za operacije.

Pri opisivanju rezultujuće sekvence naredbi SECD mašine *e'*, koristiće se operator ¦ koji označava konkatenaciju (dopisivanje) dve sekvence mašinskog koda SECD mašine. Takodje ćemo pretpostaviti (zbog jasnoće) da se rezultujući mašinski kod SECD mašine smešta u listu; tada je operator ¦ ekvivalentan sa ugradjenom funkcijom **_append** medjujezika LL, odnosno operatorom + + jezika SASL i Haskell.

Prevodjenje LL programa l u sekvencu odgovarajućih SECD komandi započinje pozivom funkcije **P'⟦l⟧** koji je jednog od sledeća dva oblika: **P⟦l⟧nil | (AP STOP)** i **P⟦l⟧nil | (STOP)**

Slede pravila za prevodjenje sintaksnih konstrukcija medjujezika LL:

P'⟦l⟧ = P⟦l⟧nil | (AP STOP)
ako je l program kome je potrebno naknadno zadati vrednosti argumenata

P'⟦l⟧ = P⟦l⟧nil | (STOP)
ako je l program kome nije potrebno naknadno zadati vrednosti argumenata

P⟦x⟧n = (LD i) pri čemu je x identifikator, a $i = loc(x,n)$

P⟦_nil⟧n = (LDC _nil)

P⟦(_quote e)⟧n = (LDC e)

P⟦(_lambda (x_1 ... x_k) e)⟧n = (LDF P⟦e⟧((x_1 ... x_k).n) |(RTN))

P⟦(e e_1 ... e_k)⟧n = (LDC _nil) | P⟦e_k⟧n | (CONS) | ... | P⟦e_1⟧n | (CONS) | P⟦e⟧n | (AP)

P⟦(_let e (x_1.e_1) ... (x_k.e_k))⟧n = (LDC _nil) | D⟦(x_k.e_k)⟧n | (CONS) | ... | D⟦(x_1.e_1)⟧n | (CONS) | (LDF P⟦e⟧((x_1 ... x_k).n) |(RTN) AP)

P⟦(_letrec e (x_1.e_1) ... (x_k.e_k))⟧n = (DUM LDC _nil) | D⟦(x_k.e_k)⟧m | (CONS) | ... | D⟦(x_1.e_1)⟧m | (CONS) | (LDF P⟦e⟧m |(RTN) RAP)
pri čemu je $m = (($x_1$ ... x_k).n)

D⟦(x.(_from l))⟧n = P⟦*find*(x,l)⟧n

D⟦(x.e)⟧n = P⟦e⟧n

P⟦(_delay e)⟧n = (LDE P⟦e⟧n | (UPD))

P⟦(_if e_1 e_2 e_3)⟧n = P⟦e_1⟧n | SEL P⟦e_2⟧n | (JOIN) P⟦e_3⟧n | (JOIN))

P⟦(_and e_1 e_2)⟧n = P⟦e_1⟧n | SEL P⟦e_2⟧n | (JOIN) (LDC _false JOIN))

P⟦(_or e_1 e_2)⟧n = P⟦e_1⟧n | SEL (LDC _true JOIN) P⟦e_2⟧n | (JOIN))

P⟦(_not e)⟧n = P⟦e⟧n | SEL (LDC _false JOIN) (LDC _true JOIN))

P⟦(_case e (x_1.e_1) ... (x_k.e_k))⟧ = P⟦e⟧n | SEL P⟦e_1⟧n | (JOIN) ... P⟦e_k⟧n | (JOIN))

P⟦(_tuple k e e_1 ... e_k)⟧n = P⟦e_k⟧n | ... | P⟦e_1⟧n | P⟦e⟧n | P⟦k⟧n | (TUP)

P⟦(_array k ((i_1 . e_1) ... (i_k . e_k)))⟧n = P⟦e_k⟧n | P⟦i_k⟧n | ... | P⟦e_1⟧n | P⟦i_1⟧n | P⟦k⟧n | (ARR)

P⟦(_update a k e)⟧n = P⟦e⟧n | P⟦k⟧n | P⟦a⟧n | (UPDT)

P⟦(_apply e e_1 ... e_k)⟧n

$= (\text{LDC \_nil}) \mid P[\![e_k]\!]n \mid (\text{CONS}) \mid \ldots \mid P[\![e_1]\!]n \mid (\text{CONS}) \mid \text{LDC } e \mid (\text{APLY})$

$P[\![(\text{\_foreign } e\ e_1 \ldots e_k)]\!]n$
$= (\text{LDC \_nil}) \mid P[\![e_k]\!]n \mid (\text{CONS}) \mid \ldots \mid P[\![e_1]\!]n \mid (\text{CONS}) \mid \text{LDC } e \mid (\text{SWT})$

$P[\![(\text{\_seq } e_1\ e_2)]\!]n = (\text{LDC \_nil}) \mid P[\![e_1]\!]n \mid (\text{CONS}) \mid (\text{LDF } P[\![e_2]\!](x.n)) \mid (\text{RTN})\ \text{AP})$
pri čemu je x identifikator koji nije slobodan u e_1 i e_2

$P[\![(o\ e_1\ e_2)]\!]n = P[\![e_2]\!]n \mid P[\![e_1]\!]n \mid (P[\![o]\!])$

P[\[_add]\]	= ADD	P[\[_sub]\]	= SUB
P[\[_mul]\]	= MUL	P[\[_quo]\]	= QUO
P[\[_div]\]	= DIV	P[\[_mod]\]	= MOD
P[\[_cons]\]	= CONS	P[\[_select]\]	= SLCT
P[\[_index]\]	= INDX	P[\[_append]\]	= APND
P[\[_member]\]	= MEMB	P[\[_nth]\]	= NTH
P[\[_rest]\]	= REST	P[\[_eq]\]	= EQ
P[\[_leq]\]	= LEQ	P[\[_le]\]	= LE
P[\[_eqNum]\]	= EQN	P[\[_leqNum]\]	= LEQN
P[\[_leNum]\]	= LESN	P[\[_eqStr]\]	= EQS
P[\[_leqStr]\]	= LEQS	P[\[_leStr]\]	= LESS

$P[\![(o\ e)]\!]n = P[\![e]\!]n \mid (P[\![o]\!])$

P[\[_car]\]	= CAR	P[\[_cdr]\]	= CDR
P[\[_tag]\]	= TAG	P[\[_len]\]	= LEN
P[\[_atom]\]	= ATOM	P[\[_number]\]	= NUM
P[\[_chr]\]	= CHR	P[\[_ord]\]	= ORD
P[\[_sin]\]	= SIN	P[\[_cos]\]	= COS
P[\[_exp]\]	= EXP	P[\[_sinH]\]	= SINH
P[\[_cosH]\]	= COSH	P[\[_log]\]	= LOG
P[\[_arcTan]\]	= ATAN	P[\[_arcTanH]\]	= ATANH
P[\[_force]\]	= AP0	P[\[_error]\]	= ERR

4.1.1.1 Komentari

Iako je **_nil** konstruktor podataka medjujezika LL, sa stanovišta SECD mašine je to konstanta, te se tako i prevodi.

Prevodi argumenata ugradjenih funkcija LL-a se u rezultujućem izrazu funkcije P javljaju u obrnutom redosledu, jer će se njihove vrednosti argumenata stavljati na stek. Tako će se pre primene naredbe SECD mašine vrednosti argumenata na steku nalaziti u originalnom redosledu - po redu navedenom u izvornom LL programu.

Pravilo za prevodjenje **let** izraza se zasniva na sledećoj ekvivalenciji (poznatoj iz λ računa):

$$(\_\text{let } e\ (x_1.e_1)\ \ldots\ (x_k.e_k)) \equiv ((\_\text{lambda } (x_1\ \ldots\ x_k)\ e)\ e_1\ \ldots\ e_k)$$

Pravilo za prevodjenje poziva ugradjene funkcije **_seq** se zasniva na ideji prikazanoj na Sl. 3.2, tj. na ekvivalenciji:

$$(\_\text{seq } e_1\ e_2) \equiv (\_\text{let } e_2\ (x.e_1))$$

pri čemu je x identifikator koji nije slobodan u e_1 i e_2.

Pravilo za prevodjenje primene ugradjene funkcije **_apply** je preuzeto iz [Budimac, Ivanović, 1991b].

Pravila za prevodjenje primene ugradjenih funkcija **_not**, **_con** i **_dis** se zasnivaju na sledećim ekvivalencijama:

$$\begin{aligned}(\_\text{not } e) &\equiv (\_\text{if } e\ \_\text{false}\ \_\text{true})\\ (\_\text{and } e_1\ e_2) &\equiv (\_\text{if } e_1\ e_2\ \_\text{false})\\ (\_\text{or } e_1\ e_2) &\equiv (\_\text{if } e_1\ \_\text{true}\ e_2)\end{aligned}$$

Pravilo za prevodjenje primene ugradjene funkcije **_case** je analogno pravilu za prevodjenje primene ugradjene funkcije **_if**. Pri tome, identifikatori x_i, i=1,...,k ne figurišu u rezultujućoj sekvenci SECD naredbi, a izbor odgovarajućeg argumenta se vrši isključivo na osnovu pozicije. Takvo rešenje je moguće, jer semantika ugradjene funkcije **_case** zahteva da uvek postoje sve grane ugradjene funkcije.

Ugradjene funkcije LL-a za rad sa listama **_append**, **_len**, **_member** itd. se prevode u odgovarajuće naredbe SECD mašine (APND, LEN, MEMB, ...) zbog efikasnosti, jer će se ugradjena naredba SECD mašine izvršavati efikasnije nego ekvivalent nižeg nivoa, koji se sastoji od sekvence postojećih naredbi.

LL izraz (**_from** *e*) se u toku prevodjenja zamenjuje definicijom odgovarajućeg identifikatora iz biblioteke funkcija *e*. Ekvivalent rezervisane reči **_from** stoga ne figuriše medju naredbama SECD mašine.

4.1.1.2 Primer

Na Sl. 4.2 se nalazi prevod LL programa za izračunavanje broja particija.

4.1.2 Implementacija SECD mašine

Svaka naredba SECD mašine se može opisati promenama stanja mašine. Dejstvo svake naredbe će biti opisano na sledeći način:

nar: S E C D ⟶ S' E' C' D'

```
( DUM LDC NIL LDF ( LD ( 0 . 1 ) LDC 1 EQ SEL ( LDC 1 JOIN ) ( LD ( 0 . 0 ) LDC 1 EQ
SEL ( LDC 1 JOIN ) ( LD ( 0 . 0 ) LD ( 0 . 1 ) LE SEL ( LDC NIL LD ( 0 . 0 ) CONS
LD ( 0 . 0 ) CONS LD ( 1 . 1 ) AP JOIN ) ( LD ( 0 . 0 ) LD ( 0 . 1 ) EQ SEL
( LDC NIL LD ( 0 . 0 ) LDC 1 SUB CONS LD ( 0 . 0 ) CONS LD ( 1 . 1 ) AP LDC 1 ADD
JOIN ) ( LDC NIL LD ( 0 . 1 ) LDC 1 SUB CONS LD ( 0 . 0 ) CONS LD ( 1 . 1 ) AP LDC
NIL LD ( 0 . 1 ) CONS LD ( 0 . 0 ) LD ( 0 . 1 ) SUB CONS LD ( 1 . 1 ) AP ADD JOIN )
JOIN ) JOIN ) JOIN ) RTN ) CONS LDF ( LDC NIL LD ( 0 . 0 ) CONS LD ( 0 . 0 ) CONS
LD ( 1 . 1 ) AP RTN ) CONS LDF ( LDC NIL LDC 10 CONS LD ( 0 . 0 ) AP RTN ) RAP AP
STOP )
```

Sl. 4.2 Broj particija broja 10 u mašinskom jeziku SECD mašine

pri čemu je **nar** naredba SECD mašine, **S**, **E**, **C** i **D** su sadržaji registara SECD mašine pre izvršenja naredbe, a **S'**, **E'**, **C'** i **D'** su sadržaji registara S, E, C i D posle izvršenja naredbe **nar**.

U opisu naredbi će se koristiti sledeće dve oznake: **[F (C.E)]** koja označava recept (engl. *recipe*) sastavljen od koda **C** i okoline **E**; i **[T *x*]**, koja označava izračunati recept (recepti su strukture podataka kojima se predstavlja "zadržano izračunavanje" u SECD mašini [Henderson, 1980], str. 220). U ovom odeljku će se koristiti i funkcija ***rh(x,y)*** koja zamenjuje glavu tačkastog izraza ***x***, izrazom ***y***. Takodje će se u opisivanju rezultata pojedinih komandi SECD mašine, zbog jednostavnosti, koristiti i pozivi ugradjenih funkcija medjujezika LL. Ti pozivi će se medjutim navoditi *iskošenim* slovima kako bi se istakla činjenica da se njima samo predstavlja vrednost definisana semantikom ugradjene funkcije, a ne LL izraz.

Na primer, oznaka *(_add x y)* samo označava vrednost dobijenu od *x* i *y* po semantičkim pravilima definisanim za funkciju **add** (u 2. glavi), a ne poziv funkcije **(_add *x y*)** medjujezika LL.

Slede pravila izvršavanja mašinskih naredbi SECD mašine:

LDC:	S E (LDC *x*.C) D	$\longrightarrow$ (*x*.S) E C D
LD:	S E (LD *i*.C) D	$\longrightarrow$ (*loc(i*,E).S) E C D
LDF:	S E (LDF *c'*.C) D	$\longrightarrow$ ((*c'*.E).S) E C D
LDE:	S E (LDE *c'*.C) D	$\longrightarrow$ ([F(*c'*.E)].S) E C D
RTN:	(*x*) E (RTN) (*s' e' c'*.D)	$\longrightarrow$ (*x*.*s'*) *e' c'* D
DUM:	S E (DUM.C) D	$\longrightarrow$ S (Ω.E) C D
		Ω je tzv. "lažna" okolina
UPD:	(*x*) *e'* (UPD) (([F(*c'*.*e'*)].S) E C D)	
		$\longrightarrow$ (*x*.S) E C D, i [F(*c'*.*e'*)] $\longrightarrow$ [T *x*]
AP:	((*c'*.*e'*) *v*.S) E (AP.C) D	$\longrightarrow$ nil (*v*.*e'*) *c'* (S E C.D)
RAP:	((*c'*.*e'*) *v*.S) (Ω.E) (RAP.C) D	$\longrightarrow$ nil *rh*(*e'*,*v*) *c'* (S E C.D)
AP0:	([F(*c'*.*e'*)].S) E (AP0.C) D	$\longrightarrow$ nil *e' c'* (([F(*c'*.*e'*)].s) E C.D)
	(*x*.S) E (AP0.C) D	$\longrightarrow$ (*x*.S) E C D
SEL:	(*x*.S) E (SEL *y z*.C) D	$\longrightarrow$ S E *(_if x y z)* (C.D)
CASE:	(*e*.S) E (CASE e_1 e_2 ... e_k.C) D, k≥1	
		$\longrightarrow$ S E (*_case e* e_1 ... e_k) (C.D)
JOIN:	S E (JOIN) (C.D)	$\longrightarrow$ S E C D

EQ: $(x\ y.S)\ E\ (EQ.C)\ D \longrightarrow ((eq\ x\ y).S)\ E\ C\ D$
(slično i za LEQ, LE, EQN, LEQN, LESN, EQS, LEQS i LESS)

ADD: $(x\ y.S)\ E\ (ADD.C)\ D \longrightarrow ((add\ x\ y).S)\ E\ C\ D$
(slično i za SUB, MUL, QUO, DIV i MOD)

SIN: $(x.S)\ E\ (SIN.C)\ D \longrightarrow ((\_sin\ x).S)\ E\ C\ D$
(slično i za COS, EXP, SINH, COSH, LOG, ATAN i ATANH)

CONS: $(x\ y.S)\ E\ (CONS.C)\ D \longrightarrow ((x.y).S)\ E\ C\ D$

CAR: $((x.y).S)\ E\ (CAR.C)\ D \longrightarrow (x.S)\ E\ C\ D$

CDR: $((x.y).S)\ E\ (CDR.C)\ D \longrightarrow (y.S)\ E\ C\ D$

TUP: $(k.e.e_1\ \ldots\ e_k.S)\ E\ (TUP.C)\ D \longrightarrow ([x.e_1\ \ldots\ e_k].S)\ E\ C\ D$

TAG: $([e.e_1 \ldots e_k].S)\ E\ (TAG.C)\ D \longrightarrow (e.S)\ E\ C\ D$

SLCT: $([e.e_1 \ldots e_k].i.S)\ E\ (SLCT.C)\ D \longrightarrow (e_i.S)\ E\ C\ D$

ARR: $(k.i_1.e_1\ \ldots\ i_k.e_k.S)\ E\ (ARR.C)\ D \longrightarrow ([e_1\ \ldots\ e_k].S)\ E\ C\ D$

UPDT: $([e_1,\ldots,e_i,\ldots,e_k].i.e'_i.S)\ E\ (UPDT.C)\ D \longrightarrow ([e_1,\ldots,e'_i,\ldots,e_k].S)\ E\ C\ D$

INDX: $([e_1,\ldots,e_k].i.S)\ E\ (INDX.C)\ D \longrightarrow (e_i.S)\ E\ C\ D$

APND: $(x\ y.S)\ E\ (APND.C)\ D \longrightarrow ((\_append\ x\ y).S)\ E\ C\ D$
(slično i za MEMB, NTH i REST)

LEN: $(x.S)\ E\ (LEN.C)\ D \longrightarrow ((\_len\ x).S)\ E\ C\ D$
(slično i za ATOM, NUL, CHR i ORD)

SWT: $(f\ (x_1\ \ldots\ x_k).S)\ E\ (SWT.C)\ D \longrightarrow (r.S)\ E\ C\ D$
pri čemu je $r=f(x_1,\ldots,x_k)$, rezultat primene strane funkcije f na argumente x_i, i=1,..k.

APLY: $(f.(x_1\ \ldots\ x_k).S)\ E\ (APLY.C)\ D \longrightarrow ((x_1\ \ldots\ x_k))\ \text{nil}\ P'[\![\ loc(f,E)\ ]\!]\ (S\ E\ C.D)$

ERR: $(e.S)\ E\ (ERR.C)\ D \longrightarrow S\ E\ (STOP)\ \text{nil}$
pri čemu se e ispisuje na ekranu

STOP: $S\ E\ (STOP)\ \text{nil} \longrightarrow S\ E\ (STOP)\ D$
$(x)\ E\ (STOP.C)\ (s'\ e'\ c'.D) \longrightarrow (r.s')\ e'\ c'\ D$

4.1.2.1 Komentari

Naredba STOP je definisana sa dva pravila, od kojih se prvo primenjuje kada je registar D prazan (tj. komanda STOP je poslednja naredba SECD programa), a drugo se primenjuje u ostalim slučajevima. Prvo pravilo formalno uvodi SECD mašinu u beskonačnu petlju (implementacija treba u tom slučaju da zaustavi rad mašine). Drugo pravilo za naredbu STOP je potrebno zbog postojanja naredbe APLY čije svako izvršavanje uvodi u SECD program po jednu (novu) naredbu STOP. Drugo pravilo za naredbu STOP je istovetno sa pravilom za naredbu RTN.

Oznaka $P'[\![\ loc(f,E)\ ]\!]$ u pravilu za APLY označava pronalaženje izraza f (koji je do tog trenutka poznat i nalazi se u okolini E) i njegovo prevodjenje funkcijom P' (koja u SECD program uvodi novu komandu STOP).

Alternativna pravila za naredbe APLY i STOP, bez čuvanja sadržaja registara S, E i C u registru D, su sledećeg oblika:

APLY: $(f\ (x_1 \ldots x_k).S)\ E\ (APLY.C)\ D \longrightarrow ((x_1 \ldots x_k).S)\ E\ (P'[\![\ loc(f,E)\ ]\!]\ C)\ D$

STOP: $S\ E\ (STOP)\ nil \longrightarrow S\ E\ (STOP.C)\ D$

$S\ E\ (STOP.C)\ D \longrightarrow S\ E\ C\ D$

Treći način realizacije komande APLY je prikazan u [Budimac, Ivanović, 1991b]. Ovaj način je primenljiv u svakoj apstraktnoj mašini i realizuje se rekurzivnim pozivom simulatora apstraktne mašine i implementacijom skupljača otpadaka (engl. *garbage collector*) od više nivoa.

Selektivno izračunavanje vrednosti argumenata logičkih funkcija **_and** i **_or** je osigurano prevodjenjem primene logičkih funkcija u odgovarajuće primene **if** izraza (te prevodjenjem u primene SEL i JOIN naredbi SECD mašine). Naredba SEL u zavisnosti od vrednosti vrha registra S u potpunosti odbacuje jednu svoju "granu", ne izračunavajući je.

Pravilo izvršavanja naredbe SWT je preuzeto iz [Ivanović, Budimac, 1990], gde je i detaljno opisano. Poseban problem pri realizaciji ovog pravila je kako tokom izvršavanja SECD programa pozvati i izračunati vrednost funkcije pisane u drugom (imperativnom) programskom jeziku, o čemu će nešto više reči biti u odeljku 5.3.6.

Navedena pravila za izvršavanje naredbi SECD mašine podrazumevaju da je mašinski program korektan, a uskladjenost tipova proverena - zbog toga podrazumevamo da će svaka naredba na vrhu steka pronaći argument kakav očekuje. U implementaciji mašinskih naredbi treba da se proverava broj i tipovi argumenata, ako se radi o implementaciji netipiziranih jezika.

Mašinski programi SECD mašine se mogu optimizovati na različite načine. Jedna od najpoznatijih mogućnosti je i tzv. optimizacija repa (engl. *tail (recursion) optimization*).

4.2 Lenja SECD mašina

Lenja SECD je po svojoj konstrukciji identična "vrednoj" SECD mašini, opisanoj u prethodnom odeljku. Lenja SECD mašina koja se opisuje u ovom odeljku je zasnovana na lenjoj SECD mašini opisanoj u [Henderson, Jones, Jones, 1983] str. 108. Skup konstrukcija izvornog jezika u čije prevode je potrebno "usaditi" naredbe za zadržavanje i forsiranje izračunavanja je, medjutim, proširen da bi podržao sve ugradjene funkcije medjujezika LL.

4.2.1 Prevodilac LL-a u jezik lenje SECD mašine

Pravila za prevodjenje LL kontrukcija u mašinski jezik lenje SECD mašine će biti data na isti način kao i u slučaju "vredne" SECD mašine. Prevodjenje LL konstrukcije l u sekvencu odgovarajućih naredbi lenje SECD mašine započinje pozivom funkcije **P'⟦*l*⟧** koji je oblika: **P⟦*l*⟧nil | (AP STOP)***.

Slede pravila za prevodjenje samo onih konstrukcija medjujezika LL, koja su izmenjena u odnosu ne vrednu SECD mašinu opisanu u prethodnom odeljku. Pravila prevodjenja primene ugradjenih funkcija **cons**, **let**, **letrec**, **car**, **cdr**, primene funkcije i obraćanja identifikatoru su preuzete iz [Henderson, Jones, Jones, 1983] str. 108, dok su ostala pravila nova:

P'⟦*l*⟧ = P⟦*l*⟧nil | (AP STOP)

P⟦*x*⟧*n* = (LD *i* AP0) pri čemu je *x* identifikator, a *i* = *loc(x,n)*

P⟦(*e* e_1 ... e_k)⟧*n* = (LDC NIL LDE P⟦e_k⟧*n* | (UPD) CONS ... LDE P⟦e_1⟧*n* | (UPD) CONS P⟦*e*⟧*n* AP)

P⟦(_let *e* (x_1.e_1) ... (x_k.e_k))⟧*n* = (LDC NIL LDE D⟦(x_k.e_k)⟧*n* | (UPD) CONS ... LDE D⟦(x_1.e_1)⟧*n* | (UPD) CONS LDF P⟦*e*⟧((x_1 ... x_k).*n*) |(RTN) AP)

P⟦(_letrec *e* (x_1.e_1) ... (x_k.e_k))⟧*n* = (DUM LDC NIL LDE D⟦(x_k.e_k)⟧*m* | (UPD) CONS ... LDE D⟦(x_1.e_1)⟧*m* | (UPD) CONS LDF P⟦*e*⟧*m* |(RTN) RAP)
pri čemu je *m* = ((x_1 ... x_k).*n*)

P⟦(_cons e_1 e_2)⟧*n* = (LDE P⟦e_2⟧*n* | (UPD) LDE P⟦e_1⟧*n* | (UPD) CONS)

P⟦(_car *e*)⟧*n* = P⟦*e*⟧*n* | (CAR AP0)

P⟦(_cdr *e*)⟧*n* = P⟦*e*⟧*n* | (CDR AP0)

P⟦(_tuple *k* *e* e_1 ... e_k)⟧*n* = (LDE P⟦e_k⟧*n* | (UPD) ... LDE P⟦e_1⟧*n* | (UPD) LDE P⟦*e*⟧ | (UPD) P⟦*k*⟧ | (TUP)

P⟦(_tag *e*)⟧*n* = P⟦*e*⟧*n* | (TAG AP0)

P⟦(_select e_1 e_2)⟧*n* = P⟦e_2⟧*n* | P⟦e_1⟧ | (SLCT AP0)

P⟦(_array *k* ((i_1 . e_1) ... (i_k . e_k)))⟧*n* = (LDE P⟦e_k⟧*n* | (UPD) P⟦i_k⟧*n* | ... LDE P⟦e_1⟧*n* | (UPD) P⟦i_1⟧*n* | P⟦*k*⟧*n* | (ARR)

P⟦(_update *a* k *e*)⟧*n* = (LDE P⟦*e*⟧*n* | (UPD) P⟦*k*⟧*n* | P⟦*a*⟧*n* | (UPDT AP0)

---

* [Henderson, Jones, Jones, 1983] str. 108, predlažu da se na prevod izraza *l* doda sekvenca **(LD (0 . 0) AP0 STOP)**, koja medjutim ne daje dobre rezultate.

$P[\![(\_\text{index } e_1\ e_2)]\!]n = P[\![e_2]\!]n \mid P[\![e_1]\!] \mid (\text{INDX AP0})$

$P[\![(\_\text{append } e_1\ e_2)]\!]n = (\text{LDE } P[\![e_2]\!]n \mid (\text{UPD}) \text{ LDE } P[\![e_1]\!]n \mid (\text{UPD}) \text{ APND})$
$P[\![(\_\text{member } e_1\ e_2)]\!]n = (\text{LDE } P[\![e_2]\!]n \mid (\text{UPD}) \text{ LDE } P[\![e_1]\!]n \mid (\text{UPD}) \text{ MEMB})$
$P[\![(\_\text{nth } e_1\ e_2)]\!]n = (\text{LDE } P[\![e_2]\!]n \mid (\text{UPD}) \text{ LDE } P[\![e_1]\!]n \mid (\text{UPD}) \text{ NTH})$
$P[\![(\_\text{rest } e_1\ e_2)]\!]n = (\text{LDE } P[\![e_2]\!]n \mid (\text{UPD}) \text{ LDE } P[\![e_1]\!]n \mid (\text{UPD}) \text{ REST})$

$P[\![(\_\text{apply } e\ e_1 \ldots e_k)]\!]n$
$= (\text{LDC NIL}) \mid (\text{LDE } P[\![e_k]\!]n \mid (\text{UPD CONS}) \mid \ldots \mid$
$(\text{LDE } P[\![e_1]\!]n \mid (\text{UPD CONS}) \mid \text{LDC } e \mid (\text{APLY})$
$P[\![(\_\text{foreign } e\ e_1 \ldots e_k)]\!]n$
$= (\text{LDC NIL}) \mid (\text{LDE } P[\![e_k]\!]n \mid (\text{UPD CONS}) \mid \ldots \mid$
$(\text{LDE } P[\![e_1]\!]n \mid (\text{UPD CONS}) \mid P[\![e]\!]n \mid (\text{SWT})$

4.2.1.1 Komentari

Naredbe lenje SECD mašine za zadržavanje izračunavanja LDE i UPD se ugradjuju samo u prevode onih konstrukcija medjujezika LL u kojima je potrebno zadržati izračunavanje. Na primer, te naredbe se ne ugradjuju u prevod primene funkcije **_add** jer je za izračunavanje te vrednosti potrebno odmah izračunati oba argumenta, pa zadržavanje ne bi imalo smisla. Naredba SECD mašine za forsiranje izračunavanja (AP0) se ugradjuje u prevod svih primena ugradjenih funkcija čiji argumenti mogu imati vrednost recepata.

Podsetimo se da u jezicima sa nestriktnom semantikom samo izračunavanje vrednosti elemenata (a ne indeksa) niza treba da bude lenjo. Zbog toga se u lenjoj SECD mašini generiše kod koji će zadržati samo izračunavanje elemenata niza.

4.2.1.2 Primer

Na Sl. 4.3 je prikazan prevod LL programa za izračunavanje broja particija.

```
( DUM LDC NIL LDE ( LDF ( LD ( 0 . 1 ) AP0 LDC 1 EQ SEL ( LDC 1 JOIN ) ( LD ( 0 . 0 )
AP0 LDC 1 EQ SEL ( LDC 1 JOIN ) ( LD ( 0 . 0 ) AP0 LD ( 0 . 1 ) AP0 LE SEL ( LDC NIL
LDE ( LD ( 0 . 0 ) AP0 UPD ) CONS LDE ( LD ( 0 . 0 ) AP0 UPD ) CONS LD ( 1 . 1 ) AP0 AP
JOIN ) ( LD ( 0 . 0 ) AP0 LD ( 0 . 1 ) AP0 EQ SEL ( LDC NIL LDE ( LD ( 0 . 0 ) AP0 LDC
1 SUB UPD ) CONS LDE ( LD ( 0 . 0 ) AP0 UPD ) CONS LD ( 1 . 1 ) AP0 AP LDC 1 ADD
JOIN ) ( LDC NIL LDE ( LD ( 0 . 1 ) AP0 LDC 1 SUB UPD ) CONS LDE ( LD ( 0 . 0 ) AP0
UPD ) CONS LD ( 1 . 1 ) AP0 AP LDC NIL LDE ( LD ( 0 . 1 ) AP0 UPD ) CONS LDE
( LD ( 0 . 0 ) AP0 LD ( 0 . 1 ) AP0 SUB UPD ) CONS LD ( 1 . 1 ) AP0 AP ADD JOIN )
JOIN ) JOIN ) JOIN ) RTN ) UPD ) CONS LDE ( LDF ( LDC NIL LDE ( LD ( 0 . 0 ) AP0 UPD )
CONS LDE ( LD ( 0 . 0 ) AP0 UPD ) CONS LD ( 1 . 1 ) AP0 AP RTN ) UPD ) CONS LDF ( LDC
NIL LDE ( LDC 10 UPD ) CONS LD ( 0 . 0 ) AP0 AP RTN ) RAP AP0 STOP )
```

Sl. 4.3 Broj particija broja 10 u mašinskom jeziku lenje SECD mašine

4.2.2 Implementacija lenje SECD mašine

Sve definicije naredbi SECD mašine opisane u prethodnom odeljku ostaju neizmenjene, jer se lenjo izračunavanje postiže umetanjem već postojećih naredbi **LDE, UPD** i **AP0** u prevod odgovarajućih LL konstrukcija.*

4.3 SK redukciona mašina

SK redukciona mašina redukuje kombinatorski graf kojim je predstavljen izraz čija se vrednost izračunava. Program medjujezika LL se prevodi u izraz kombinatorskog računa (koji se predstavlja grafom) i koji se potom redukuje po pravilima svojstvenim svakom kombinatoru.

SK redukciona mašina koja se opisuje u ovom odeljku je zasnovana na prvobitnoj SK mašini [Turner, 1979], ali je proširena novim ugradjenim kombinatorima da bi podržala sve ugradjene funkcije medjujezika LL.

U naredna dva odeljka se navode pravila za prevodjenje LL konstrukcija u kombinatore (komande) SK redukcione mašine i pravila za redukciju tih kombinatora. Pored ugradjenih kombinatora koji direktno implementiraju ugradjene funkcije medjujezika LL, implementiran je tzv. Turner-ov skup kombinatora [Turner, 1979] (ali je kombinator B' zamenjen kombinatorom B*): S, K, I, B, C, S', C' i B*.

Ukoliko zagrade u kombinatorskom izrazu nisu navedene podrazumeva se vezivanje operanada "ulevo", tj. $x\ y\ z$ je isto što i $((x\ y)\ z)$.

4.3.1 Prevodilac LL-a u izraz kombinatorskog računa (jezik SK mašine)

Pravila za prevodjenje LL konstrukcija u kombinatorski račun će biti definisana funkcijom P koja preslikava konstrukcije medjujezika LL u odgovarajući kombinatorski izraz. Kako rezultujući izraz ne sadrži identifikatore, lista imena nije potrebna pri prevodjenju, ni okolina pri izvršavanju. Funkcija P će biti definisana navodjenjem pravila sledećeg oblika:

$$P[\![e]\!] = e'$$

pri čemu e predstavlja sintaksnu konstrukciju medjujezika LL, a e' predstavlja rezultujući izraz kombinatorskog računa. Funkcije D i *find* imaju isto značenje kao i u slučaju SECD mašine.

---

* [Henderson, Jones, Jones, 1983] str. 109 redefinišu komandu **STOP**, koja bi, u slučaju kada je na vrhu registra **S** recept, trebalo da forsira njegovo dalje izračunavanje. To, medjutim, nije potrebno, jer će se ta aktivnost izvršiti komandom **AP0** koja se nalazi pre komande **STOP**.

Pri definisanju funkcije P, koristiće se i algoritam za apstrahovanje identifikatora iz λ izraza (engl. *bracket abstraction algorithm*). Za apstrahovanje identifikatora x iz izraza E, koristiće se uobičajena notacija $[x]$E.

Slede pravila za prevodjenje konstrukcija medjujezika LL:

$$
\begin{aligned}
&P[\![x]\!]n && = x\\
&P[\![\text{\_nil}]\!]n && = \text{\_nil}\\
&P[\![(\text{\_quote } e)]\!]n && = P[\![e]\!]\\
&P[\![(\text{\_lambda } (x_1 \ldots x_k)\ e)]\!] && = [x_1](\ldots([x_k]P[\![e]\!])\ldots)\\
&P[\![(e\ e_1 \ldots e_k)]\!] && = P[\![e]\!]\ P[\![e_1]\!] \ldots P[\![e_k]\!]\\
&P[\![(\text{\_let } e\ (x_1.e_1) \ldots (x_k.e_k))]\!] && = [x_1](\ldots([x_k]P[\![e]\!])\ldots)D[\![(x_1.e_1)]\!]\ldots D[\![(x_k.e_k)]\!]\\
&P[\![(\text{\_letrec } e\ (x_1.e_1) \ldots (x_k.e_k))]\!] &&\\
&\qquad = ([(x_1 \ldots x_k)]P[\![e]\!])(Y([(x_1 \ldots x_k)](D[\![(x_1.e_1)]\!]\ldots D[\![(x_k.e_k)]\!])))
\end{aligned}
$$

$$
\begin{aligned}
&D[\![(x.(\text{\_from } l))]\!] && = P[\![\mathit{find}(x,l)]\!]\\
&D[\![(x.e)]\!] && = P[\![e]\!]
\end{aligned}
$$

$$
\begin{aligned}
&P[\![(\text{\_delay } e)]\!] && = P[\![e]\!]\\
&P[\![(\text{\_force } e)]\!] && = \text{FORCE } P[\![e]\!]
\end{aligned}
$$

$$
\begin{aligned}
&P[\![(\text{\_if } e_1\ e_2\ e_3)]\!] && = \text{IF } P[\![e_1]\!]\ P[\![e_2]\!]\ P[\![e_3]\!]\\
&P[\![(\text{\_and } e_1\ e_2)]\!] && = \text{IF } P[\![e_1]\!]\ P[\![e_2]\!]\ \text{\_false}\\
&P[\![(\text{\_or } e_1\ e_2)]\!]n && = \text{IF } P[\![e_1]\!]\ \text{\_true}\ P[\![e_2]\!]\\
&P[\![(\text{\_not } e)]\!] && = \text{IF } P[\![e]\!]\ \text{\_false \_true}\\
&P[\![(\text{\_case } e\ (x_1.e_1) \ldots (x_k.e_n))]\!] && = \text{CASE } P[\![e]\!]\ P[\![e_1]\!] \ldots P[\![e_n]\!]
\end{aligned}
$$

$$
\begin{aligned}
&P[\![(\text{\_tuple } n\ e\ e_1 \ldots e_n)]\!] && = \text{TUP } P[\![n]\!]\ P[\![e]\!]\ P[\![e_1]\!] \ldots P[\![e_n]\!]\\
&P[\![(\text{\_array } n\ ((i_1.e_1) \ldots (i_n.e_n)))]\!] &&\\
&\qquad && = \text{ARR } P[\![n]\!]\ P[\![i_1]\!]\ P[\![e_1]\!] \ldots P[\![i_n]\!]\ P[\![e_n]\!]\\
&P[\![(\text{\_update } a\ n\ e)]\!] && = \text{UPDT } P[\![a]\!]\ P[\![n]\!]\ P[\![e]\!]
\end{aligned}
$$

$$
\begin{aligned}
&P[\![(\text{\_foreign } e\ e_1 \ldots e_k)]\!]n && = \text{SWT } P[\![e]\!]\ (P[\![e_1]\!] \ldots P[\![e_n]\!])\\
&P[\![(\text{\_apply } e\ e_1 \ldots e_k)]\!]n && = \text{APLY } e\ (P[\![e_1]\!] \ldots P[\![e_n]\!])\\
&P[\![(\text{\_error } e)]\!] && = \text{ERR } P[\![e]\!]
\end{aligned}
$$

$$
P[\![(o\ e_1\ e_2)]\!] = P[\![o]\!]\ P[\![e_1]\!]\ P[\![e_2]\!]
$$

$P[\![\text{\_add}]\!]$	= ADD	$P[\![\text{\_sub}]\!]$	= SUB
$P[\![\text{\_mul}]\!]$	= MUL	$P[\![\text{\_quo}]\!]$	= QUO
$P[\![\text{\_div}]\!]$	= DIV	$P[\![\text{\_mod}]\!]$	= MOD
$P[\![\text{\_cons}]\!]$	= CONS	$P[\![\text{\_select}]\!]$	= SLCT
$P[\![\text{\_index}]\!]$	= INDX	$P[\![\text{\_append}]\!]$	= APND
$P[\![\text{\_member}]\!]$	= MEMB	$P[\![\text{\_nth}]\!]$	= NTH
$P[\![\text{\_rest}]\!]$	= REST	$P[\![\text{\_eq}]\!]$	= EQ

P⟦ _leq ⟧	= LEQ	P⟦ _le ⟧	= LE
P⟦ _eqNum ⟧	= EQN	P⟦ _leqNum ⟧	= LEQN
P⟦ _leNum ⟧	= LESN	P⟦ _eqStr ⟧	= EQS
P⟦ _leqStr ⟧	= LEQS	P⟦ _leStr ⟧	= LESS
P⟦ _seq ⟧	= SEQ		

P⟦ (*o e*) ⟧ = P⟦ *o* ⟧ P⟦ *e* ⟧

P⟦ _atom ⟧	= ATOM	P⟦ _number ⟧	= NUM
P⟦ _car ⟧	= CAR	P⟦ _cdr ⟧	= CDR
P⟦ _tag ⟧	= TAG	P⟦ _len ⟧	= LEN
P⟦ _sin ⟧	= SIN	P⟦ _cos ⟧	= COS
P⟦ _exp ⟧	= EXP	P⟦ _sinH ⟧	= SINH
P⟦ _cosH ⟧	= COSH	P⟦ _log ⟧	= LOG
P⟦ _arcTan ⟧	= ATAN	P⟦ _arcTanH ⟧	= ATANH
P⟦ _chr ⟧	= CHR	P⟦ _ord ⟧	= ORD

Sledi algoritam apstrahovanja identifikatora, pri čemu važi da je x identifikator koja se apstrahuje, E, F i G su s-izrazi LL-a u kojima se ne pojavljuje identifikator x, a E_x, F_x i G_x su konstrukcije u kojima se identifikator x pojavljuje. Primenljivost navedenih pravila se ispituje redom, odozgo nadole:

$[x]x$	= I
$[x]E$	= K E
$[x]((E\ x)\ F_x)$	= S E $[x]F_x$
$[x]((E\ x)\ F)$	= S E F
$[x]((E\ F_x)\ G_x)$	= S' E $[x]F_x$ $[x]G_x$
$[x]((E\ (F\ G_x))$	= B* E F $[x]G_x$
$[x]((E\ F_x)\ G)$	= C' E $[x]F$ G
$[x](E\ x)$	= E
$[x](E_x\ F_x)$	= S $[x]E_x$ $[x]F_x$
$[x](E\ F_x)$	= B E $[x]F_x$
$[x](E_x\ F)$	= C $[x]E$ F

$[(x_1 \ldots x_k)]E$ = U($[x_1]([(x_2 \ldots x_k)]E)$

[nil]E = K E

4.3.1.1 Komentari

Algoritam za apstrahovanje identifikatora koji se navodi u prethodnom odeljku je algoritam Kennaway-a [1984] str. 2 u kome je pravilo za prevodjenje u kombinator B' zamenjeno pravilom za prevodjenje u kombinator B* i u kome je dodat slučaj apstrahovanja liste identifikatora iz proizvoljnog izraza E. Može se pokazati da ovaj algoritam proizvodi isti kombinatorski izraz kao i poboljšanje

Diller-ovog algoritma [1988] str. 98 opisano u [Budimac, Maćoš, Ivanović, 1993]; te kao osnovni Schönfinkel-ov algoritam prevodjenja u kombinatore S, K i I, optimizovan Curry & Feys-ovim, Turner-ovim i Scheevel-ovim optimizacijama koji polazni izraz prevode u primenu kombinatora S, K, I, B, C, S', C' i B*. Opis poslednjeg pomenutog algoritma sa optimizacijama se nalazi u [Turner, 1979;1981b], [Scheevel, 1986], [Peyton Jones, 1987] str. 260 i [Field, Harrison, 1988] str. 274. Alternativni algoritmi za prevodjenje λ izraza u kombinatorske termove se mogu pronaći na primer i u [Diller, 1988] str. 90, [Burton, 1982] i [Noshita, 1985].

Pravilo za prevodjenje _**let** izraza se zasniva na osnovu iste ekvivalencije kao i u slučaju SECD mašine (str. 90).

Pravilo za prevodjenje _**letrec** izraza se zasniva na smeštanju svih identifikatora i svih izraza u zasebne liste:

$$\_\textbf{letrec}\ e\ (x_1 \ldots x_k)\ (e_1 \ldots e_k)$$

i potom, upotrebi kombinatora Y (standardnog zatvorenog izraza λ računa) za eliminisanje rekurzije na "uobičajeni" način. Da bi prevod LL **letrec** izraza bio kompletan, treba da postoje i pravila za apstrahovanje listi identifikatora iz proizvoljnog izraza, što je učinjeno proširivanjem Kennaway-ovog algoritma. Primetimo da se time uvodi novi kombinator U, koji služi za "raspakivanje" strukturnih objekata (liste), tj. za primenu funkcije na svaki element strukturnog objekta (videti pravilo redukcije za kombinator U u sledećem odeljku). Ovaj način prevodjenja **letrec** izraza je dat na osnovu rada Scott-a i Strachey-a [1971] str. 15; 27, a njegov opis se nalazi i u [Diller, 1988] str. 70.

Alternativno pravilo za prevodjenje **letrec** izraza je sledeće:

$$P[\![(\_\text{letrec}\ e\ (x_1.e_1) \ldots (x_k.e_k))]\!] = Y\ [t](\_\text{tuple}\ k\ 0\ L_1 \ldots L_k)\ ,$$
$$L_i = (P[\![e_i]\!][(\_select\ t\ 1)/x_1]\ldots)[(\_select\ t\ k)/x_k]$$

i sledi na osnovu jednakosti:

$$t = (\_\text{tuple}\ k\ 0\ L_1 \ldots L_k),\ L_i = (P[\![e_i]\!][(\_select\ t\ 1)/x_1]\ldots)[(\_select\ t\ k)/x_k]$$

odakle sledi (na osnovu "uobičajene" primene kombinatora Y):

$$Y\ (\_\text{lambda}\ (t)\ (\_\text{tuple}\ k\ L_1 \ldots L_k))$$

pri čemu su _**tuple** i _**select** ugradjene funkcije medjujezika LL (i ugradjeni kombinatori SK mašine). Ovako opisano prevodjenje predstavlja varijantu gore-opisanog i zasnovano je na [Landin, 1966], a varijante te ideje su opisane i u [Field, Harrison, 1988] str. 132.

Primetimo da je u osnovi oba pravila za prevodjenje ista ideja: smeštanje identifikatora i odgovarajućih izraza u složene strukture podataka (listu, odnosno n-

torku) i zamene svakog identifikatora pristupom odgovarajućem elementu strukture podataka u koju su smešteni. Pristup se vrši ili primenom kombinatora U koji primenjuje funkciju na svaki element složene strukture podataka (kao u prvom slučaju) ili primenom funkcije za pristup složenoj strukturi podataka (kao u drugom slučaju).

Jedan način za prevodjenje **letrec** izraza (medjusobno rekurzivnih definicija) bez smeštanja u neku strukturu podataka je dato i u [Hudak, 1989] str. 372, ali podrazumeva ciljni (medju)jezik znatno višeg nivoa nego što je to LL.

Kako SK mašina realizuje nestriktnu semantiku, to LL ugradjena funkcija **_delay** nema smisla (jer je ionako svako izračunavanje inicijalno zadržano). Zbog toga se primene ugradjene funkcije **_delay** pri prevodjenju ignorišu.

Primene ugradjenih funkcija **_append**, **_len**, **_member** itd. se direktno prevode u primene odgovarajućih kombinatora (APND, LEN, MEMB), zbog efikasnosti (slično kao u slučaju SECD mašine). Kako se prevodjenjem u jezik SK mašine implementira nestriktna semantika izvornog jezika, potrebno je da implementacija kombinatora za rad sa listama (na primer, APND, MEMB, NTH i sl.) omogući lenjo izračunavanje vrednosti svojih argumenata (videti odeljak 5.3.5).

Moguće optimizacije SK redukcione mašine se na primer mogu pronaći u [Turner, 1981b].

4.3.1.2 Primer

Na Sl. 4.4 je prikazan prevod LL programa za izračunavanje broja particija.

```
( ( U ( B . U ) ( ( B* . K ) . K ) ( C . I ) . 10 ) Y U K U ( B . K ) ( ( S' . CONS )
( C . S ) . I ) ( ( C' . CONS ) ( ( B* B S ( ( C' . IF ) ( C . EQ ) . 1 ) . 1 ) ( S' .
B ) ( ( C' . IF ) ( C . EQ ) . 1 ) . 1 ) ( S ( ( B* S' . S ) ( S' C' . IF ) . LE ) ( C
. S ) . I ) ( S ( ( B* S' . S ) ( S' C' . IF ) . EQ ) ( ( C' C' . ADD ) ( C . S ) ( C
. SUB ) . 1 ) . 1 ) ( ( S' S' . S ) ( C C' B* . ADD ) ( C . SUB ) . 1 ) ( C ( ( C' . C'
) . S' ) . SUB ) . I ) )
```

Sl. 4.4 Broj particija broja 10 u mašinskom jeziku lenje SK mašine

4.3.2 Implementacija SK mašine

Svaki kombinator (naredba SK mašine) se može opisati načinom na koji transformiše sopstvene argumente tj. izraze koji se nalaze neposredno iza kombinatora. Redukovanje svakog kombinatora će biti opisano na sledeći način:

$$R[\![E]\!] \longrightarrow E'$$

pri čemu je E kombinatorski izraz pre primene kombinatora, a E' je rezultujući kombinatorski izraz. Redukcija kombinatorskog izraza E započinje na sledeći način:

$$R'[\![R[\![L]\!]]\!] \longrightarrow E'$$

Definicija i diskusija funkcije **R'** je data u nastavku.

Slično kao i u opisu SECD mašine, u opisivanju rezultata pojedinih kombinatora će se koristiti i pozivi ugradjenih funkcija medjujezika LL, ali samo da bi se jednostavnije objasnili rezultati primene kombinatora.

Redukovanje izraza se vrši sve dok se ne dobije izraz koji se više ne može redukovati.

Slede pravila redukcije kombinatora (naredbi SK mašine), pri čemu su E, E_1, ... E_n, F, G, H, x, y i z proizvoljni kombinatorski izrazi:

$R'[\![\text{CONS } E\ F]\!] \longrightarrow (R[\![E]\!] . R[\![F]\!])$
$R'[\![\text{TUP } n\ E\ E_1 \ldots E_n]\!] \longrightarrow [R[\![E]\!] . R[\![E_1]\!] \ldots R[\![E_n]\!]]$
$R'[\![E]\!] \longrightarrow R[\![E]\!]$, inače

$R[\![\text{I } E]\!] \longrightarrow E$
$R[\![\text{K } E\ F]\!] \longrightarrow E$
$R[\![\text{S } E\ F\ G]\!] \longrightarrow (E\ G)\ (F\ G)$
$R[\![\text{B } E\ F\ G]\!] \longrightarrow E\ (F\ G)$
$R[\![\text{C } E\ F\ G]\!] \longrightarrow E\ G\ F$
$R[\![\text{S' } E\ F\ G\ H]\!] \longrightarrow E\ (F\ H)\ (G\ H)$
$R[\![\text{B}^* E\ F\ G\ \text{H}]\!] \longrightarrow E\ (F\ (G\ H))$
$R[\![\text{C' } E\ F\ G\ H]\!] \longrightarrow E\ (F\ H)\ G$

$R[\![\text{Y } E]\!] \longrightarrow E\ (\text{Y } E)$
$R[\![\text{U } E\ (\text{CONS } F\ G)]\!] \longrightarrow E\ F\ G$

$R'[\![\text{ARR } n\ I_1\ E_1 \ldots I_n\ E_n]\!] \longrightarrow [E_1, \ldots, E_n]$

$R[\![\text{CAR } (\text{CONS } E\ F)]\!] \longrightarrow E$
$R[\![\text{CDR } (\text{CONS } E\ F)]\!] \longrightarrow F$
$R[\![\text{TAG } (\text{TUP } n\ E\ E_1 \ldots E_n)]\!] \longrightarrow E$
$R[\![\text{SLCT } (\text{TUP } n\ E\ E_1 \ldots E_n)\ i]\!] \longrightarrow E_i$
$R[\![\text{UPDT } [E_1,\ldots,E_i,\ldots,E_n]\ i\ E'_i]\!] \longrightarrow [E_1,\ldots,E'_i,\ldots,E_n]$
$R[\![\text{INDX } [E_1, \ldots, E_n]\ i]\!] \longrightarrow E_i$

$R[\![\text{FORCE } E]\!] \longrightarrow R[\![E]\!]$
$R[\![\text{SEQ } E\ F]\!] \longrightarrow R[\![F]\!]$, pri čemu i $R'[\![E]\!]$
$R[\![\text{IF } x\ E\ F]\!] \longrightarrow (\_if\ R[\![x]\!]\ E\ F)$
$R[\![\text{CASE } x\ E_1 \ldots E_n]\!] \longrightarrow (\_case\ x\ E_1 \ldots E_n)$

$R[\![\text{SWT } E\ (E_1 \ldots E_n)]\!] \longrightarrow R$

pri čemu je R rezultat primene strane funkcije E na listu argumenata $E_1,\ldots,E_n$.

R⟦APLY E (E_1 ... E_n)⟧ ⟶ R⟦P⟦E⟧(E_1,...,E_n)⟧
R⟦ERR E⟧ ⟶ ERR R⟦E⟧
pri čemu se R⟦E⟧ ispisuje na ekranu.

R⟦EQ x y⟧ ⟶ (_eq R⟦x⟧ R⟦y⟧)
slično i za LEQ, LE, EQN, LEQN, LESN, EQS, LEQS i LESS
R⟦ADD x y⟧ ⟶ (_add R⟦x⟧ R⟦y⟧)
slično i za SUB, MUL, QUO, DIV i MOD
R⟦APND x y⟧ ⟶ (_append R⟦x⟧ R⟦y⟧)
slično i za MEMB, NTH i REST
R⟦SIN x⟧ ⟶ (_sin R⟦x⟧)
slično i za COS, EXP, SINH, COSH, LOG, ATAN i ATANH
R⟦LEN x⟧ ⟶ (_len R⟦x⟧)
slično i za ATOM, NUM, CHR i ORD

R⟦E⟧ ⟶ E, u svim ostalim slučajevima

4.3.2.1 Komentari

Mašina prekida sa radom kada se kombinatorski izraz više ne može redukovati. Primetimo da pored pravila za ERR E, tu spadaju i pravila koje se primenjuju "u svim ostalim slučajevima", koji obuhvata i izraze oblika CONS E F i TUP n E E_1 ... E_n.

CONS E F i TUP n E E_1 ... E_n se ne redukuju da bi se omogućilo lenjo izračunavanje i beskonačne strukture podataka. Kako bi se medjutim kao rezultat redukovanja dobio konkretan rezultat, čitav proces redukovanja je potrebno ostvariti interakcijom redukovanja (poziva funkcije R) i dodatne funkcije R' koja će posebno redukovati komponente izraza CONS E F i TUP n E E_1 ... E_n.

Umesto definicije **R'** prikazane u prethodnom odeljku, koja konstruiše strukturu podataka, u mnogim drugim implementacijama se struktura štampa, bez konstruišanja.

Pri redukovanju primene ugradjenog kombinatora SWT na argumente, argumenti E_1 do E_n se ne redukuju unapred. Implementacija strane funkcije E treba da forsira izračunavanje svojih argumenata. Ovakva koncepcija je "kompatibilna" sa koncepcijom SK redukcione mašine. Ako se usvoji drugačija koncepcija po kojoj se argumenti strane funkcije redukuju pre prosledjivanja, tada se kao argumenti ne mogu prosledjivati beskonačne strukture podataka. Neki dalji aspekti realizacije stranih funkcija u okolini lenje apstraktne mašine su dati u [Budimac, Maćoš, 1993].

Ovde prikazana SK redukciona mašina nije potpuno lenja (engl. *fully lazy*) jer se u pojedinim slučajevima neki izrazi mogu izračunati i više od jednom. Razlog tome su pravila za redukciju u kojima se selektuje jedan od argumenata. Ovaj nedostatak se može ispraviti implementacijom tzv. preusmeravajućih čvorova (engl. *indirection nodes*) ili/i prvo redukovanjem željenog argumenta, pa tek onda

njegovim selektovanjem. Ovaj drugi pristup se može definisati sledećim pravilima redukcije:

$R[\![I\ x]\!] \longrightarrow R[\![x]\!]$
$R[\![CAR\ (CONS\ x\ y)]\!] \longrightarrow R[\![x]\!]$
$R[\![NTH\ x\ n]\!] \longrightarrow R[\![(\_nth\ x\ n)]\!]$
$R[\![TAG\ (TUP\ n\ t\ e_1\ \ldots\ e_n)]\!] \longrightarrow R[\![t]\!]$
$R[\![SLCT\ (TUP\ n\ t\ e_1\ \ldots\ e_n)\ i]\!] \longrightarrow R[\![e_i]\!]$
$R[\![INDX\ [e_1, \ldots, e_n]\ i]\!] \longrightarrow R[\![e_i]\!]$
$R[\![IF\ x\ y\ z]\!] \longrightarrow R[\![(\_if\ R[\![x]\!]\ y\ z)]\!]$
$R[\![CASE\ e\ e_1\ \ldots\ e_n]\!] \longrightarrow R[\![(\_case\ e\ (x_1.e_1)\ \ldots\ (x_n.e_n))]\!]$

$R[\![K\ x\ y]\!] \longrightarrow R[\![x]\!]$
$R[\![CAR\ (CONS\ x\ y)]\!] \longrightarrow R[\![y]\!]$

4.4 Superkombinatorska mašina

Superkombinatorska (redukciona) mašina redukuje superkombinatorski graf kojim je predstavljen izraz čija se vrednost izračunava. Izraz LL jezika se prevodi u superkombinatorski izraz (koji se predstavlja grafom) i koji se potom redukuje po pravilima svojstvenim svakom superkombinatoru.

Superkombinatorska mašina koja se opisuje u ovom odeljku je prvobitna superkombinatorska mašina [Hughes, 1982] opisana u [Peyton Jones, 1987] str. 220; [Field, Harrison, 1988] str. 305. Superkombinatorska mašina je proširena istim skupom ugradjenih kombinatora za implementaciju LL-a kao i SK mašina iz prethodnog odeljka.

4.4.1 Prevodilac LL-a u superkombinatore

Pravila za prevodjenje LL konstrukcija u superkombinatore će biti definisana funkcijom P koja preslikava sintaksne konstrukcije medjujezika LL u odgovarajuće superkombinatorske izraze. Prevodjenje se sastoji u eliminisanju slobodnih identifikatora iz tela anonimnih funkcija medjujezika LL. Jedna verzija postupka (koja se često naziva i *lambda lifting*) je definisana u ovom odeljku.

Rezultat prevodjenja je superkombinatorski izraz zapisan u obliku **_letrec** izraza medjujezika LL. Superkombinator se u LL-u predstavlja izrazom sledećeg oblika: **((komb** x_1 **...** x_n**) telo)**, pri čemu je **komb** ime dodeljeno kombinatoru u toku prevodjenja, a x_1, ..., x_n su argumenti kombinatora (argumenti **lambda** izraza prošireni odgovarajućim brojem novih argumenata), a **telo** je telo kombinatora (prethodno telo **lambda** izraza).

Funkcija P će biti definisana navodjenjem pravila sledećeg oblika:

$$P[\![e]\!] = e' ; f := f$$

pri čemu e predstavlja sintaksnu konstrukciju medjujezika LL, e' predstavlja transformisanu konstrukciju medjujezika LL, f predstavlja stari rezultujući izraz (superkombinatorski) izraz, a f' je novi rezultujući izraz, nastao posle primene pravila P. Kako su superkombinatorski izrazi f i f' predstavljeni s-izrazom, to se u njihovom gradjenju može koristiti već poznati operator | koji označava konkatenaciju dva s-izraza. Pri primeni pravila koji ne utiču na izmenu superkombinatorskog izraza, deo iza znaka ; neće biti navodjen.

Iz opisa funkcije P se vidi da ona: preslikava originalnu LL konstrukciju u rezultujuću, sve vreme je "smanjujući"; i kreira novi (superkombinatorski) izraz posle prepoznavanja odredjenih LL konstrukcija. f i f' se mogu posmatrati i kao (ulazni i izlazni) atribut LL izraza e u opisu ovog načina prevodjenja atributivnim gramatikama.

U navodjenju pravila za prevodjenja, usvajamo konvenciju da se oznakama e, e_1, ...,e_n obeležavaju sintaksne konstrukcije LL-a, oznakama E, E_1, ...,E_n obeležavaju LL izrazi koji ne sadrže nijedan **lambda** izraz, a oznakom L LL izraz koji sadrži bar jedan **lambda** izraz pri čemu L nije sâm **lambda** izraz. Oznakom ϵ se predstavlja prazan LL izraz.

U prevodu LL konstrukcije u superkombinatore se koriste i sledeće funkcije: S - koja sve sukcesivne **lambda** izraze pretvara u jedan **lambda** izraz; funkcija Q koja pretvara LL konstrukciju u superkombinatorski izraz, ali na jednom nivou ispod tekućeg; i funkcija P_{SK} koja prevodi LL izraz u kombinatore SK redukcione mašine.

Prevodjenje LL izraza l započinje pozivom funkcije $\mathbf{P'}[\![l]\!]$, koja je sledećeg oblika $\mathbf{P}[\![\mathbf{S}[\![l]\!]]\!]$.

$$\mathbf{P'}[\![l]\!] = \mathbf{P}[\![\mathbf{S}[\![l]\!]]\!]$$

$$\mathbf{S}[\![(\_\text{letrec } e\ (x_1 . e_1) \ldots (x_n . e_n))]\!] = (\_\text{letrec } e\ (x_1 . \mathbf{S}[\![e_1]\!]) \ldots (x_n . \mathbf{S}[\![e_n]\!]))$$

$$\mathbf{S}[\![(\_\text{let } e\ (x_1 . e_1) \ldots (x_n . e_n))]\!] = (\_\text{let } e\ (x_1 . \mathbf{S}[\![e_1]\!]) \ldots (x_n . \mathbf{S}[\![e_n]\!]))$$

$$\mathbf{S}[\![(f\ x_1 \ldots x_n)]\!] = (\mathbf{S}[\![f]\!]\ \mathbf{S}[\![x_1]\!] \ldots \mathbf{S}[\![x_n]\!])$$

$$\mathbf{S}[\![(\_\text{lambda } (x_1 \ldots x_k)\ (\_\text{lambda } (x_1 \ldots x_p) \ldots (\_\text{lambda } (x_1 \ldots x_n)\ L) \ldots))]\!] = (\_\text{lambda } (x_1 \ldots x_k\ x_1 \ldots x_p \ldots x_1 \ldots x_n)\ \mathbf{S}[\![L]\!])$$

$$\mathbf{S}[\![E]\!] = E$$

$$\mathbf{P}[\![(\_\text{letrec } e\ (y_1 . e_1) \ldots (y_n . e_n)]\!] = \mathbf{P}[\![(\_\text{letrec } e\, \mathbf{P}[\![(y_1 . e_1)]\!] \ldots \mathbf{P}[\![(y_n . e_n)]\!]]\!] \ ; f := \textbf{nil}$$

$$\mathbf{P}[\![(\_\text{let } e\ (y_1 . e_1) \ldots (y_n . e_n)]\!] = \mathbf{P}[\![(\_\text{let } e\, \mathbf{P}[\![(y_1 . e_1)]\!] \ldots \mathbf{P}[\![(y_n . e_n)]\!]]\!] \ ; f := \textbf{nil}$$

$$\mathbf{P}[\![(x . E)]\!] = \epsilon \ ; f := ((\$x)\ E[\$x/x]) \mid f[\$x/x]$$

$$\mathbf{P}[\![(x . (\_\text{lambda } (x_1 \ldots x_m)\ E)]\!] = \epsilon\ ; f := ((\$x\ x_1 \ldots x_m)\ E[\$x/x]) \mid f[\$x/x]$$

$$\mathbf{P}[\![(x . (\_\text{lambda } (x_1 \ldots x_m)\ L)]\!] = \mathbf{P}[\![(x . (\_\text{lambda } (x_1 \ldots x_m)\ \mathbf{Q}[\![L]\!])]\!]$$

$$\mathbf{P}[\![(\_\text{letrec } e)]\!] = \epsilon$$

$$;f := P_{SK}[\![(\_letrec \$Prg (\$Prg . (e[\$y_1/y_1];\$y_2/y_2;\ldots;\$y_n/y_n])) \mid f]\!]$$

pri čemu su y_i, i=1,...,n identifikatori uvedeni **letrec** izrazom koji se prevodi, a $\$y_i$ uvedeni kombinatori pri prevodu **letrec** izraza.

$$P[\![(\_let\ e)]\!] = \epsilon$$
$$;f := P_{SK}[\![(\_letrec \$Prg (\$Prg . (e[\$y_1/y_1;\$y_2/y_2;\ldots;\$y_n/y_n]) \mid f]\!]$$

pri čemu su y_i, i=1,...,n identifikatori uvedeni **let** izrazom koji se prevodi, a $\$y_i$ uvedeni kombinatori pri prevodu **let** izraza.

$$Q[\![(x\ .\ e)]\!] = (x\ .\ Q[\![e]\!])$$
$$Q[\![(\_letrec\ e\ e_1\ \ldots\ e_n)]\!] = (\_letrec\ Q[\![e]\!]\ Q[\![e_1]\!]\ \ldots\ Q[\![e_n]\!])$$
$$Q[\![(\_let\ e\ e_1\ \ldots\ e_n)]\!] = (\_let\ Q[\![e]\!]\ Q[\![e_1]\!]\ \ldots\ Q[\![e_n]\!])$$
$$Q[\![(\_lambda(x_1\ \ldots\ x_m)\ E)]\!] = \$\alpha\ v_1\ \ldots\ v_n$$
$$;f := ((\$\alpha\ v_1\ \ldots\ v_n\ x_1\ \ldots\ x_m)\ E) \mid f$$

ako su $v_1,\ldots,v_n$ slobodni identifikatori izraza E, a $\$\alpha$ je identifikator koji nije slobodan u E.

$$Q[\![(\_lambda(x_1\ \ldots\ x_m)\ L)]\!] = Q[\![(\_lambda(x_1\ \ldots\ x_m)\ Q[\![L]\!])]\!]$$
$$Q[\![(f\ x_1\ \ldots\ x_n)]\!] = (Q[\![f]\!]\ Q[\![x_1]\!]\ \ldots\ Q[\![x_n]\!])$$
$$Q[\![E]\!] = E$$

4.4.1.1 Komentari

Sukcesivne **lambda** izraze je moguće zameniti samo jednim **lambda** izrazom zbog osobine superkombinatorske mašine da redukuje samo one kombinatore (**lambda** izraze) čiji su svi argumenti poznati u vreme redukovanja superkombinatora. Zbog toga je više sukcesivnih **lambda** izraza ekvivalentno jednom, spojenom. Funkcije P i Q realizuju isti algoritam prevodjenja LL konstrukcija u superkombinatore. Razlika izmedju njih se pravi jer se za definicije na gornjem nivou LL izraza (unutar prvog **let** ili **letrec** izraza) zna da nisu u okviru nijednog **lambda** izraza te da nemaju slobodnih identifikatora. Zbog toga je u većini slučajeva takve definicije (sa gornjeg nivoa) moguće pretvoriti u superkombinatore na jednostavniji način nego ostale.

Funkcija P_{SK} je uvedena u prethodnom odejku i u superkombinatorskom okruženju se koristi bez izmena. Pošto u superkombinatorskom izrazu ne postoje **lambda** izrazi, funkcija P_{SK} neće proizvesti kombinatore S, I, S', B, B*, C i C' (koji nastaju apstrahovanjem identifikatora iz **lambda** izraza). Funkcija P_{SK} će medjutim proizvesti kombinatore U, K i Y kao rezultat prevodjenja **letrec** izraza. Peyton Jones [1987] str. 232 predlaže drugačiji pristup u realizaciji **letrec** izraza, tako što ih posmatra kao tekstualni opis cikličnih grafova i sve **letrec** izraze predstavlja grafom. Takav pristup je efikasniji, ali zavisi od implementacije

superkombinatorske mašine i ne može se lako opisati. Kreiranje cikličnog grafa na osnovu **letrec** izraza će medjutim biti prikazano u odeljku implementacije G mašine.

Pri odredjivanju identifikatora superkombinatora, usvojena je konvencija predložena u [Peyton Jones, 1987] po kojoj svi identifikatori superkombinatora počinju znakom **$**. Usvaja se takodje konvencija po kojoj superkombinatori nastali od lokalnih definicija u **letrec** izrazu nose imena identifikatora uvedenih u tim definicijama ispred kojih se dodaje znak $. Superkombinatori uvedeni **lambda** izrazima dobijaju proizvoljna imena $α, koja moraju biti različita od svih postojećih idenifikatora superkombinatora.

Neke od mogućih optimizacija superkombinatorske mašine su opisane u [Peyton Jones, 1987] str. 228;243 i primenljive su i na prevodjenje opisano ovde.

4.4.1.2 Primer

Na Sl. 4.5 je prikazan prevod LL izraza sa Sl. 4.1, ali bez primene funkcije P_{SK} za transformisanje **letrec** izraza. Izrazi ovakvog oblika (sa očuvanim **letrec** izrazima) predstavljaju polaznu tačku za prevodioce superkombinatora u apstraktne mašine zasnovane na superkombinatorima (G mašina, TIM mašina i sl.).

```
(_letrec $Prg
    ($Prg (part (_quote 10)))
    (($part n) ($par n n))
    (($par m n)
          (_if (_eq n (_quote 1))
               (_quote 1)
               (_if (_eq m (_quote 1))
                    (_quote 1)
                    (_if (_le m n)
                         ($par m m)
                         (_if (_eq m n)
                              (_add ($par m (_sub m (_quote 1))) (_quote 1))
                              (_add ($par m (_sub n (_quote 1))) ($par (_sub m n) n))
)    )    )    )    )    )
```

Sl. 4.5 Broj particija nekog broja kao superkombinatorski izraz

4.4.2 Implementacija superkombinatorske mašine

Superkombinatorska (redukciona) mašina se od SK (redukcione) mašine razlikuje samo po načinu na koji se apstrahuju identifikatori iz **lambda** izraza LL jezika. Kod SK mašine se identifikatori apstrahuju pretvaranjem u kombinatore I, K i S (i njihove optimizovane varijante B, C, S', B*, C') a kod superkombinatorske po pretvaranju svih **lambda** izraza u superkombinatore (po gornjem algoritmu).

Implementacija superkombinatorske mašine se od implementacije SK mašine razlikuje samo po tome što se uvode pravila za redukovanje superkombinatora, a izbacuju pravila za redukovanje kombinatora I, S, B, C, S', B*, C' (kombinatori K, U i Y su potrebni zbog **letrec** izraza).

Pravilo za redukovanje superkombinatora je oblika:

$$R'[\![(\$x\ x_1 \ldots x_n)\ e)\ ]\!]\ v_1 \ldots v_k \longrightarrow e'$$

pri čemu je $(\$x\ x_1 \ldots x_n)\ e)$ superkombinator čije ime je $\$x$, argumenti $x_1 \ldots x_n$, a telo e; $v_1,\ldots,v_k$ su vrednosti na koje kombinator treba da bude primenjen, a e' je rezultujući superkombinatorski izraz. U opisu funkcije R' će se koristiti i funkcija R, koja je sledećeg oblika:

$$R\ [\![e]\!]\ x\ v = e'$$

pri čemu je e superkombinatorski izraz, x je jedan od argumenata superkombinatora (istovremeno i jedan od identifikatora koji se možda pojavljuje u e) i v je vrednost kojom treba zameniti svaku pojavu identifikatora x u izrazu e.

Slede pravila redukcije:

$$R'[\![(\$x\ x_1 \ldots x_n)\ e)\ ]\!]\ v_1 \ldots v_k \longrightarrow (\$x\ x_1 \ldots x_n)\ e),\ k<n$$

$$R'[\![(\$x\ x_1 \ldots x_n)\ e)\ ]\!]\ v_1 \ldots v_n \longrightarrow R[\![\ldots R[\![R[\![e]\!] x_1\ v_1]\!] x_1\ v_1 \ldots]\!] x_n\ v_n$$

$$\begin{aligned} R[\![x]\!]\ x\ v &= v \\ R[\![x]\!]\ y\ v &= y,\ x \neq y \\ R[\![(f\ x_1 \ldots x_n)]\!] &= (R[\![f]\!]\ x\ v\ R[\![x_1]\!]\ x\ v \ldots R[\![x_n]\!]\ x\ v) \end{aligned}$$

4.4.2.1 Komentari

Primetimo da je pravilom $R[\![e]\!]\ x\ v$ u stvari implementirana uobičajena notacija $e[v/x]$. Pravilo R je medjutim znatno jednostavnije od moguće opšte implementacije notacije $e[v/x]$, koja bi trebala da uzme u obzir i mogućnost da izraz e može da bude **lambda** izraz (što kod superkombinatora nije slučaj). Funkcija R je dakle specijalni slučaj realizacije notacije $e[v/x]$.

4.5 G mašina

G mašina sadrži 4 registra: S, G, C i D. Stanje u kome se mašina nalazi je jednoznačno odredjeno sadržajem sva četiri registra. Početno stanje G mašine je sledeće: C sadrži mašinski program G mašine, a S, G i D su prazni (ovakvo stanje se inicijalizuje prvom komandom svakog G programa **begin**). Registri S, C i D su po svojoj strukturi stekovi, a registar G sadrži graf. G mašina koja se opisuje u ovom odeljku je zasnovana na G mašini uvedenoj u [Johnsson, 1984; Augustsson, 1984] i opisanoj u [Peyton Jones, 1987] str. 293; [Field, Harrison, 1988] str. 387. Skup naredbi je proširen da bi podržao sve ugradjene funkcije medjujezika LL.

U naredna dva odeljka se navode pravila za prevodjenje LL konstrukcija u naredbe G mašine i pravila izvršavanja tih naredbi. Nešto veća pažnja će biti posvećena naredbama koje su nove u odnosu na prvobitne, dok se detaljniji opis originalnih naredbi (eval, unwind, return, jump, jfalse, push, pushglob, pop, slide, update, alloc, car, cdr, add, mkap, cons) nalazi u [Peyton Jones, 1987] str. 293;322.

4.5.1 Prevodilac LL-a u jezik G mašine

Ulazni podatak prevodioca medjujezika LL u jezik G mašine je superkombinatorski izraz nastao na osnovu LL programa. Pogodan superkombinatorski izraz za prevodjenje se dobija primenom algoritma iz prethodnog odeljka bez primene funkcije P_{SK} nad rezultujućim izrazom.

Pravila za prevodjenje superkombinatorskog izraza u mašinski jezik G mašine će biti definisana funkcijom P. U definiciji funkcije P se koriste pomoćne funkcije R i C koje prevode telo superkombinatora na osnovu konteksta; kao i funkcija U koja prevodi ugradjene superkombinatore. Funkcije R i C će biti definisane pravilima sledećeg oblika:

$$R[\![e]\!]\rho d = e'$$

pri čemu *e* predstavlja sintaksnu konstrukciju medjujezika LL, ρ predstavlja funkciju koja za zadati identifikator izračunava indeks odgovarajućeg argumenta u tekućem kontekstu (segmentu steka), pri čemu najdalji element konteksta ima broj 0 (poslednji argument kombinatora ima broj 1, pretposlednji ima broj 2 itd). **d** je dužina tekućeg konteksta umanjena za 1. Oznaka $\rho[x=n]$ označava proširenje funkcije ρ slučajem $\rho(x)=n$. Izraz $d\text{-}\rho(x)$ izračunava poziciju identifikatora **x** računajući od vrha steka. Prevodjenje superkombinatorskog izraza *l* u sekvencu odgovarajućih naredbi G mašine započinje pozivom funkcije $P''[\![l]\!]$. Podsetimo se da je *l* oblika:

(_letrec $prg e_1 ... e_k)

pri čemu su e_i, i=1,...,k oblika:

(($x x_1 ... $x_{n'}$) e)

a **n'** zavisi od e_i.

Sekvenca mašinskih naredbi G mašine je predstavljena s-izrazom, a pri opisivanju rezultujuće sekvence kodova G mašine *e'*, koristiće se i operator | koji označava konkatenaciju (dopisivanje) dve sekvence mašinskog koda G mašine. Za grupisanje izraza u kojima učestvuju ρ i *d* unutar s-izraza će se koristiti uglaste zagrade, kako se takvi izrazi ne bi mešali sa sintaksom s-izraza.

Prevodjenje superkombinatorskog izraza *l* počinje pozivom $P''[\![l]\!]$. Slede pravila za prevodjenje svih izraza:

$P''[\![l]\!] = (\text{begin}) \mid P'[\![l]\!] \mid U[\![\_\text{cons}]\!]\ U[\![\_\text{add}]\!]\ U[\![\_\text{sub}]\!] \ldots$

$P'[\![l]\!] = (\text{pushg \$Prg eval print end}) \mid P[\![e_1]\!] \ldots P[\![e_k]\!]$

$P[\![(\$x\ x_1 \ldots x_n)\ e)]\!] = ((\$x\ \text{n}) \mid R[\![e]\!][x_1=\text{n}, x_2=\text{n-1}, \ldots, x_n=1]\text{n}$

$R[\![e]\!]\rho d = C[\![e]\!]\rho d \mid (\text{updat } [d+1] \text{ pop } d \text{ unwin})$

$C[\![x]\!]\rho d = (\text{push } [d-\rho(x)]\ )$, x je identifikator
$C[\![(\_\text{quote } x)]\!]\rho d = (\text{pushc } C[\![x]\!]\rho d)$
$C[\![\_\text{nil}]\!]\rho d = (\text{pushc \_nil})$
$C[\![\$x]\!]\rho d = (\text{pushg } \$x)$, $\$x$ je superkombinator ili ugradjena funkcija

$C[\![(e_1\ e_2)]\!]\rho d = C[\![e_2]\!]\rho d \mid C[\![e_1]\!]\rho[d+1] \mid (\text{mkap})$
$C[\![(\_\text{delay } e)]\!]\rho d = C[\![e]\!]\rho d$

$C[\![(\_\text{case } e\ (x_1.e_1) \ldots (x_n.e_n))$
$= C[\![(\_\text{case } e\ e_1 \ldots e_n)]\!]$

$C[\![(\_\text{let } e\ (x_1.e_1) \ldots (x_n.e_n))]\!]\rho d$
$= D[\![(x_1.e_1)]\!]\rho d \mid D[\![(x_2.e_2)]\!]\rho[x_1=d+1][d+1] \mid \ldots \mid$
$D[\![(x_n.e_n)]\!]\rho[x_1=d+1,\ldots,x_{n-1}=d+n\text{-}1][d+n\text{-}1] \mid$
$C[\![e]\!]\rho[x_1=d+1,\ldots,x_{n-1}=d+n\text{-}1, x_n=d+n][d+n] \mid$
$(\text{slide } n)$

$C[\![(\_\text{letrec } e\ (x_1.e_1) \ldots (x_n.e_n))]\!]\rho d$
$= C_L[\![((x_1.e_1) \ldots (x_n.e_n))]\!]\rho'[d+n] \mid$
$C[\![e]\!]\rho'[\text{d+n}] \mid (\text{slide } n)$

pri čemu je

$\rho' = \rho[x_1=d+1, x_2=\text{d+2}, \ldots, x_n=d+n]$
$C_L[\![((x_1.e_1) \ldots (x_n.e_n))]\!]\rho d = (\text{alloc } n) \mid D[\![(x_1.e_1)]\!]\rho d \mid (\text{update } n)$
$\mid D[\![(x_2.e_2)]\!]\rho d \mid (\text{update } [n\text{-}1])$
....
$\mid D[\![(x_n.e_n)]\!]\rho d \mid (\text{update } 1)$

$D[\![(x.(\_\text{from } l))]\!]n = P[\![find(\text{x,l})]\!]n$
$D[\![(x.e)]\!]n = P[\![e]\!]n$

$U[\![k]\!] = (k) \mid C[\![k]\!]$, k je ugradjena funkcija medjujezika LL

$C[\![\_\text{if}]\!]\rho d = ((\text{push 0 eval jfalse L1 push 1 jump L2 label L1 push 2 label L2 update 4 pop 3 unwind}))$

C⟦ _and ⟧ρd = ((push 0 eval jfalse L1 push 1 eval label L1 updat 3 pop 2 return))
C⟦ _or ⟧ρd = ((push 0 eval jtrue L1 push 1 eval label L1 updat 3 pop 2 return))
C⟦ _not ⟧ρd = ((eval jtrue L1 _true jump L2 label L1 _false label L2 updat 1 return))

C⟦ _case ⟧ρd = ((case L_1 L_2 ... L_{d+1} label L_1 push L_1 jump L_{d+1} ... label L_{d+1} update [d+2] pop d unwin))

C⟦ _nth ⟧ρd = ((push 1 eval push 1 eval nth updat 3 pop 2 unwin))

C⟦ _seq ⟧ρd = ((push 1 eval print push 1 eval updat 3 pop 2 unwin))
C⟦ _force ⟧ρd = ((eval))

C⟦ _apply ⟧ = ((aply updat 1 unwin))
C⟦ _foreign ⟧ = ((swt updat 1 unwin))

C⟦ _cons ⟧ρd = ((cons updat 1 return))
C⟦ _car ⟧ρd = ((eval car updat 1 unwin))
C⟦ _cdr ⟧ρd = ((eval cdr updat 1 unwin))

C⟦ _tuple ⟧ρd = ((tup updat 1 return))
C⟦ _tag ⟧ρd = ((eval tag updat 1 unwin))
C⟦ _select ⟧ρd = ((push 1 eval push 1 slct updat 3 pop 2 unwin))
C⟦ _array ⟧ρd = ((arr updat 1 return))
C⟦ _update ⟧ρd = ((push 1 eval push 2 eval updt updat 4 return))
C⟦ _index ⟧ρd = ((push 1 eval push 1 indx updat 3 pop 2 unwin))

C⟦ *o* ⟧ρd = ((push 1 eval push 1 eval CC⟦ *o* ⟧ updat 3 pop 2 return))
ako je *o* ugradjena funkcija LL-a od dva argumenta

CC⟦ _add ⟧	= add	CC⟦ _sub ⟧	= sub
CC⟦ _mul ⟧	= mul	CC⟦ _quo ⟧	= quo
CC⟦ _div ⟧	= div	CC⟦ _mod ⟧	= mod
CC⟦ _append ⟧	= apnd	CC⟦ _member ⟧	= memb
CC⟦ _rest ⟧	= rest	CC⟦ _eq ⟧	= eq
CC⟦ _leq ⟧	= leq	CC⟦ _le ⟧	= le
CC⟦ _eqNum ⟧	= eqn	CC⟦ _leqNum ⟧	= leqn
CC⟦ _leNum ⟧	= lesn	CC⟦ _eqStr ⟧	= eqs
CC⟦ _leqStr ⟧	= leqs	CC⟦ _leStr ⟧	= less
CC⟦ _seq ⟧	= seq		

C⟦ *o* ⟧ρd = ((eval CC⟦ *o* ⟧ updat 1 return))

ako je *o* ugradjena funkcija LL-a jednog argumenta

CC⟦ _len ⟧	= len	CC⟦ _atom ⟧	= atom
CC⟦ _number ⟧	= num	CC⟦ _sin ⟧	= sin
CC⟦ _cos ⟧	= cos	CC⟦ _exp ⟧	= exp
CC⟦ _sinH ⟧	= sinh	CC⟦ _cosH ⟧	= cosh
CC⟦ _log ⟧	= log	CC⟦ _arcTan ⟧	= atan
CC⟦ _arcTanH ⟧	= atanh	CC⟦ _ord ⟧	= ord
CC⟦ _chr ⟧	= chr	CC⟦ _error ⟧	= err

4.5.1.1 Komentari

Program G mašine je predstavljen s-izrazom na sledeći način:

```
(begin
        (pushg $Prg eval print end)
        ($x n (kod za $x)) ...
        (_cons (kod za _cons)) ...
)
```

Imena i arnost uvedenih superkombinatora i imena ugradjenih superkombinatora se nalaze na prvom nivou u s-izrazu, dok se odgovarajući kod nalazi na drugom nivou. Takvom organizacijom je olakšano interpretiranje koda G mašine. Ukoliko se program G mašine dalje prevodi u mašinski program konkretne mašine, može se odabrati i drugačija reprezentacija, više nalik ciljnom asembleru.

Prevod primene logičkih operatora na argumente je nastao na osnovu ekvivalencije uvedene na strani 91.

Umesto prvobitnog (i ovde prikazanog) prevoda primene ugradjenog superkombinatora **_if**, primena se može prevesti i na sledeći način:

C⟦ _if ⟧ρd = ((push 0 eval jtrue L1 (push 1 join) (push 2 join) update 4 pop 3 unwind))

pri čemu su naredbe G mašine **jtrue** i **join** analogne naredbama SECD mašine **SEL** i **JOIN** (strana 92). Ovakav način je nešto jednostavniji za implementaciju simulatora G mašine, jer se dve grane uslovnog izraza **_if** nalaze na jednom nivou ispod tekućeg. Na isti način se mogu prevoditi i logičke ugradjene funkcije.

U prevodu primene nekih ugradjenih funkcija LL-a se na kraju sekvence G naredbi nalazi komanda **return** a nekih **unwin**. Intuitivno, **return** se nalazi na kraju prevoda primena svih onih funkcija čiji rezultat ne može da bude funkcija, a **unwin** se nalazi na kraju prevoda primene onih funkcija čiji rezultat može da bude funkcija.

Kao i u slučaju implementacije prevodioca LL-a u mašinski jezik SK mašine i ovde nije prikazana "potpuno lenja" verzija G mašine. Da bi se dobila takva verzija, potrebno je kod primene ugradjenih funkcija navedenih na strani 104

izračunavati selektovani argument. Tako bi na primer prevodi primene funkcija **_car** i **_if** imali sledeći oblik:

$C[\![\_if]\!]\rho d$ = ((push 0 eval jfalse L1 push 1 jump L2 label L1 push 2 label L2 **eval** update 4 pop 3 unwind))

$C[\![\_car]\!]\rho d$ = ((eval CAR **eval** update 1 unwind))

Nad kodom G mašine se mogu primeniti i mnoge druge optimizacije i to je jedna od njenih osnovnih prednosti. Sve optimizacije navedene na primer u [Peyton Jones, 1987] str. 338 se mogu primeniti (ili jednostavno prilagoditi) i na kod koji je dobijen prevodjenjem definisanim u ovom odeljku.

4.5.1.2 Primer

Na Sl. 4.6 se nalazi prevod izraza medjujezika LL sa Sl. 4.1.

```
(begin
   (pushg $Prg eval print end)
   (($Prg) (pushg $Prg pushc 10 mkap))
   (($part n) (pushg $par push 1 mkap push 1 mkap))
   (($par m n) (pushg _if pushg _eq push 2 mkap pushc 1 mkap mkap pushc 1 mkap pushg _if
               pushg _eq push 1 mkap pushc 1 mkap mkap pushc 1 mkap pushg _if pushg _le
               push 1 mkap push 2 mkap mkap pushg $par push 1 mkap push 1 mkap pushg
               _if pushg _eq push 1 mkap push 2 mkap mkap pushg _add pushg $par push
               1 mkap pushg sub push 1 mkap pushc 1 mkap mkap pushg _add pushg $par
               push 1 mkap pushg sub push 2 oushc 1 mkap mkap mkap pushg $par pushg
               _sub push 1 mkap push 2 mkap mkap mkap mkap mkap)))
```

Sl. 4.6 Broj particija nekog broja u mašinskom jeziku G mašine

4.5.2 Implementacija G mašine

Svaka naredba G mašine se može opisati promenom stanja mašine. Dejstvo svake naredbe će biti opisano na sledeći način:

nar: S G C D ⟶ S' G' C' D'

pri čemu je **nar** naredba G mašine, **S**, **G**, **C** i **D** su sadržaji registara G mašine pre izvršenja naredbe, a **S'**, **G'**, **C'** i **D'** je sadržaj registara S, G, C i D posle izvršenja naredbe **nar**. U daljem tekstu će se oznakom *a.R* označavati operacija stavljanja elementa *a* na vrh regista (steka) *R* (R je S, C ili D). Prazan stek i prazan graf će biti označavani oznakom **nil**.

U opisu komandi će se koristiti i sledeće oznake mogućih čvorova grafa (sadržaja registra G):

$(e_1.e_2)$ čvor koji označava izraz (cons e_1 e_2) (tj. neizračunatu listu);

$[t.e_1...e_n]$	čvor koji označava izraz (tuple n t e_1 ... e_n) (tj. neizračunatu označenu n-torku);
$[e_1,...,e_n]$	čvor koji označava niz sa neizračunatim elementima;
$(e_1@e_2)$	čvor koji označava primenu izraza (funkcije, kombinatora) e_1 na izraz e_2;
$(k;C)$	čvor koji označava superkombiantor od **k** argumenata čije telo je C;
hole	čvor sa (još uvek) nedefinisanom sadržinom;
c, c_1, c_2	čvor(ovi) koji označavaju konstante (brojeve ili identifikatore);
x, x_1, x_2	bilo koji čvor(ovi);

G[n=(e_1.e_2)] označava graf (sadržina registra G) u kome je čvor **n** tipa **(e_1.e_2)**. Oznakom **G[n=G n']** će biti označen graf u kome je sadržina čvora **n** ista kao i sadržina čvora **n'**. Ako se neka od gornje dve oznake pojavljuje sa leve strane pravila za opis komande G mašine, a nijedna oznaka se ne nalazi sa desne uz registar G, onda se oznaka sa leve strane podrazumeva i sa desne strane.

U opisivanju rezultata pojedinih komandi G mašine će se, zbog jednostavnosti, koristiti i pozivi ugradjenih funkcija medjujezika LL, sa istim značenjem kao u prethodnim odeljcima.

Slede pravila izvršavanja mašinskih naredbi G mašine:

begin: S G (begin.C) D ⟶ nil nil C nil

push: $(e_0.e_1. \ldots e_k.S)$ G (push k.C) D ⟶ $(e_k.e_0.e_1. \ldots e_k.S)$ G C D
pushc: S G (push c.C) D ⟶ (n.S) G[n=c] C D
pushg: S G (pushglob f.C) D ⟶ (n.S) G[n=c] C D
pop: $(e_1. \ldots e_k.S)$ G (pop k.C) D ⟶ S G C D
slide: $(e_0.e_1. \ldots e_k.S)$ G (slide k.C) D ⟶ $(e_0$.S) G C D
updat: $(e_0.e_1. \ldots e_k.S)$ G (updat k.C) D ⟶ $(e_1. \ldots e_k.S)$ G[e_k=G e_0] C D
alloc: S G (alloc k.C) D ⟶ $(e_1. \ldots e_k.S)$ G[e_1=hole,...,e_k=hole] C D
mkap: $(e_1.e_2.S)$ G (mkap.C) D ⟶ (e.S) G[e=$(e_1@e_2)$] C D
eval: (e.S) G[e=$(e_1@e_2)$] (eval.C) D ⟶ (e.nil) G (unwin.nil) (S.C.D)
(e.S) G[e=(0;C')] (eval.C) D ⟶ (e.nil) G (C'.nil) (S.C.D)
(e.S) G[e=c] (eval.C) D ⟶ (e.S) G C D
(e.S) G[e=$(e_1.e_2)$] (eval.C) D ⟶ (e.S) G C D
(e.S) G[e=$[t.e_1...e_n]$] (eval.C) D ⟶ (e.S) G C D
(e.S) G[e=$[e_1,...,e_n]$] (eval.C) D ⟶ (e.S) G C D
(e.S) G[e=$(k;C)$] (eval.C) D ⟶ (e.S) G C D
unwin: (e.nil) G[e=c] (unwin.nil) (S.C.D) ⟶ (e.S) G C D
(e.nil) G[e=$(e_1.e_2)$] (unwin.nil) (S.C.D) ⟶ (e.S) G C D
(e.S) G[e=$(e_1@e_2)$] (unwin.nil) (S.C.D) ⟶ (e_1.e.S) G (unwin.nil) D
$(e_0.e_1. \ldots .e_k.S)$ G[e_0=$(k;C)$,e_i=$(e_{i-1}@n_i)$] (unwin.nil) D, $1 \le i \le k$ ⟶ $(n_1.n_2. \ldots .n_k.e_k.S)$ G C D

$(e_0.e_1. \ldots .e_i.\text{nil})$ G[$e_0=(k;C')$] (unwin.nil) (S.C.D), i<k
$\longrightarrow$ (e_i.S) G C D

return: $(e_0.e_1. \ldots .e_k.\text{nil})$ G (return.nil) (S.C.D)
$\longrightarrow$ (e_k.S) G C D

print: (e.S) G[$e=(e_1.e_2)$] (print.C) D
$\longrightarrow$ (($e_1.e_2$).S) G (eval.print.eval.print.C) D

(e.S) G[$e=[t.e_1 \ldots e_k)$] (print.C) D
$\longrightarrow$ ([$t.e_1 \ldots e_k$).S) G (eval.print....eval.print.C) D

jump: S G (jump L. ... .label L.C) D $\longrightarrow$ S G C D

jfalse: (e.S) G[e=_true] (jfalse L.C) D $\longrightarrow$ S G C D

(e.S) G[e=_false] (jfalse L. ... .label L.C) D
$\longrightarrow$ S G C D

jtrue: (e.S) G[e=_false] (jtrue L.C) D $\longrightarrow$ S G C D

(e.S) G[e=_true] (jtrue L. ... .label L.C) D
$\longrightarrow$ S G C D

label: S G (label L.C) D $\longrightarrow$ S G C D

cons: $(e_1.e_2$.S) G (cons.C) D $\longrightarrow$ (e.S) G[$e=(e_1.e_2)$] C D

car: (e.S) G[$e=(e_1.e_2)$] (car.C) D $\longrightarrow$ (e_1.S) G C D

cdr: (e.S) G[$e=(e_1.e_2)$] (car.C) D $\longrightarrow$ (e_2.S) G C D

tup: $(k.e.e_1 \ldots .e_k$.S) G (tup.C) D $\longrightarrow$ (e'.S) G[$e'=[e.e_1 \ldots e_k])$] C D

tag: (e.S) G[$e=[t.e_1 \ldots e_k]$] (tag.C) D $\longrightarrow$ (t.S) G C D

slct: (e.i.S) G[$e=[t.e_1 \ldots e_k]$],i=c] (slct.C) D
$\longrightarrow$ (e_i.S) G C D

arr: $(k.i_1.e_1 \ldots i_k.e_k$.S) G (arr.C) D $\longrightarrow$ (e'.S) G[$e'=[e_1 \ldots e_k]$] C D

updt: $(a.i.e'_i$.S) G[$a=[e_1,\ldots,e_i,\ldots,e_k]$,$i$=c] (updt.C) D
$\longrightarrow$ (e'.S) G[$e'=[e_1,\ldots,e'_i,\ldots,e_n]$] C D

indx: (e.i.S) G[$e=[e_1 \ldots e_k]$,i=c] (indx.C) D $\longrightarrow$ (e_i.S) G C D

swt: $(e.e_1 \ldots .e_n$.S) G[$e=c,e_1=x_1,\ldots,e_n=x_n$] (swt.C) D
$\longrightarrow$ (r.S) G[$r=x$] C D

pri čemu je r rezultat primene (strane) funkcije e na izraze $e_1,\ldots,e_n$

aply: $(e.e_1 \ldots .e_n$.S) G[$e=c,e_1=x_1,\ldots,e_n=x_n$] (aply.C) D
$\longrightarrow$ ($e_1 \ldots .e_n$) G P'[[c]] (S.C.D)

err: (e.S) G[$e=x$] (err.C) D $\longrightarrow$ (e.S) G (end) nil

end: S G (end) nil $\longrightarrow$ S G (end) nil

S G (end.C) (S'.C'.D) $\longrightarrow$ S' G C' D

apnd: ($e_1.e_2$.S) G[$e_1=x_1,e_2=x_2$] (apnd.C) D $\longrightarrow$ (e.S) G[$e=(\_append\ x_1\ x_2)$]C D
slično i za memb, nth i rest.

case: (e.S) G[e=nil] (case L_1 L_2. label L_1) D $\longrightarrow$ S G C D

(e.S) G[$e=(x_1.x_2)$] (case L_1 L_2. ... .label L_2.C) D $\longrightarrow$ S G C D

(e.S) G[$e=[i.x_1 \ldots x_n)$] (case L_1 $L_{2 \ldots L}$n. ... .label L_i.C) D
$\longrightarrow$ S G C D

add: ($e_1.e_2$.S) G[$e_1=c_1,e_2=c_2$] (add.C) D $\longrightarrow$ (e.S) G[$e=(\_add\ c_1\ c_2)$] C D
slično i za sub, mul, quo, div i mod.

sin: $(e.S)\ G[e=c]\ (\text{sin.C})\ D \longrightarrow (e'.S)\ G[e'=(\_sin\ e)]\ C\ D$
slično i za cos, exp, sinh, cosh, log, atan, atan i atanh

eq: $(e_1.e_2.S)\ G[e_1=x_1,e_2=x_2]\ (\text{eq.C})\ D \longrightarrow (e.S)\ G[e=(\_eq\ x_1\ x_2)]\ C\ D$
slično i za le, leq, eqn, leqn, lesn, eqs, leqs i less

len: $(e.S)\ G[e=x]\ (\text{len.C})\ D \longrightarrow (e'.S)\ G[e'=(\_len\ x)]\ C\ D$
slično i za atom, num, ord i num

4.5.2.1 Komentari

U prvobitnoj G mašini se preporučuje po jedna verzija naredbe **push** za svaki tip konstanti koji postoji u jeziku koji se implementira. Za tako nešto nema potrebe, ukoliko je unutrašnja reprezentacija programa LL-a "kompatibilna" sa unutrašnjom reprezentacijom struktura G mašine. U tom slučaju je dovoljna samo jedna G instrukcija (**pushc**), jer je konstanta već predstavljena na odgovarajući način.

Komanda **jtrue** nije neophodna (pored postojanja originalne **jfalse**), ali je uvedena zbog kraćeg zapisa prevoda primene funkcije **_or**.

U prikazanim pravilima za prevodjenje se predlaže uključivanje prevoda svih ugradjenih funkcija uz svaki program, što se u aktualnoj implementaciji može izbeći.

Izrazi oblika CONS *E F* i TUP *E* $E_1 \ldots E_n$ se (kao i u SK i superkombinatorskoj mašini) ne redukuju (u pravilu za instrukciju **eval**) da bi se omogućilo lenjo izračunavanje i beskonačne strukture podataka. Kako bi se medjutim kao rezultat redukovanja dobio "konkretan" rezultat, redukovanje pojedinih komponenata treba posebno forsirati naredbom **print**. Naredba **print** u prvobitnoj mašini štampa vrednosti sa vrha steka, dok se ovde predlaže samo redukovanje izraza doklegod je to moguće. Izračunata vrednost (koja se nalazi na vrhu steka) se može po završetku redukovanja, odštampati odjednom. Odluka zavisi od implementacije G mašine.

Implementacija komande **aply** u G mašini je analogna implementaciji iste komande u SECD mašini. Realizacija komande **end** u G mašini je analogna realizaciji komande **STOP** u SECD mašini (upravo zbog komande **aply**).

Mašina zaustavlja rad kada naidje na komandu **end**.

Glava 5

Mogućnosti implementacije LL-a

U ovoj glavi su predstavljeni različiti aspekti implementacije LL-a: translatora izvornih jezika u LL, prevodilaca LL-a u jezike apstraktnih mašina i simulatora apstraktnih mašina. Prikazana je jedna moguća unutrašnja struktura podataka i predloženi su potrebni moduli za realizaciju čitavog sistema za medjujezik LL. Dve analize izvršavanja LL programa na apstraktnoj i konkretnoj mašini su date na kraju glave.

Pri svim razmatranjima u ovoj glavi podrazumevaju se jednostavne (mikro) računarske arhitekture sa ograničenom unutrašnjom memorijom (a čiji tipičan predstavnik su mikro-računari zasnovani na mikro-procesorima serije 80x86 pod operativnim sistemom MS-DOS*).

5.1 Struktura podataka

Osnovna struktura podataka za implementaciju LL-a je binarno stablo sa čvorovima promenljive veličine i zasnovana je na strukturi opisanoj u [Ivanović, Budimac, 1989; Budimac, Ivanović, 1989;1990a].

Čvor binarnog stabla je prikazan strukturom promenljivog sloga u jeziku Modula-2 na Sl. 5.1, i njime su implementirani svi tipovi podataka LL programa. Brojevi su implementirani kao polja **IntVal** i **ReaVal** promenljivog dela sloga, identifikatori (uključujući i logičke konstante) su implementirani kao pokazivači na tabelu simbola (polje **SymVal**), a složeni tipovi podataka su implementirani kao dva pokazivača na istovetnu strukturu (polja **Head** i **Tail**).

Elementi nizova treba da zauzmu sukcesivne memorijske lokacije i to je u principu moguće ostvariti na dva načina:

---

* MS-DOS je zaštićeno ime firme Microsoft, SAD.

- rezervisanjem posebnog memorijskog prostora za potrebe nizova, što može da bude neekonomično rešenje,

- zauzimanjem potrebnog broja sukcesivnih lokacija u zajedničkom prostoru rezervisanom za sve strukture podataka, što komplikuje realizaciju skupljača otpadaka (engl. *garbage collector*).

```
TypeSE   = (IntSE, ReaSE, SymSE, ConsSE, TuplSE)

SExp     = POINTER TO Node;
Node     = RECORD
             (* podaci potrebni za
                skupljanje otpadaka *)
             CASE T : TypeSE OF
                 IntSE : IntVal    : LONGINT |
                 ReaSE : ReaVal    : REAL    |
                 SymSE : SymVal    : SymbTabEntry |
                 ConsSE,
                 TuplSE : Head,
                          Tail     : SExp
             END
           END;
```

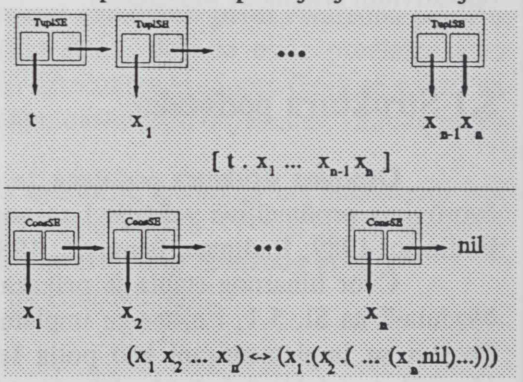
Sl. 5.1 Unutrašnja struktura podataka LL-a

Nizove je na računarima sa ograničenim memorijskim resursima gotovo nemoguće efikasno implementirati, i zbog toga neće biti dalje razmatrani.

Iako u LL-u ne postoji podela brojeva na cele i realne, oni moraju biti posebno predstavljeni u unutrašnjoj memoriji. Primetimo takodje da se liste i n-torke predstavljaju na istovetan način, što nije neuobičajena praksa u implementaciji funkcionalnih programskih jezika [Peyton Jones, 1987] str. 184. Na taj način je pristup pojedinim elementima n-torke sporiji, ali je organizacija memorije ujednačena. Nasuprot tome, n-torke je moguće predstaviti i jedinstvenim blokom, čime bi se omogućio brži pristup pojedinim elementima, ali bi se binarno stablo sastojalo od čvorova različitih veličina, što bi komplikovalo upravljanje memorijom.

Na Sl. 5.2 je prikazan način predstavljanja predloženom strukturom označene n-torke i liste.

Predloženom strukturom se ne predstavljaju samo podaci medjujezika LL, već i izvorni kod medjujezika LL. Strukture podataka potrebne za implementaciju unutrašnjih struktura apstraktnih mašina, kao i zapisa mašinskih programa tih mašina, se mogu uz mala prilagodjenja realizovati već opisanom strukturom **SExp**, na sledeći način.



Sl. 5.2 Predstavljanje n-torke i tačkastog s-izraza

Registri (vredne i lenje) SECD mašine su stekovi, koji se prirodno mogu realizovati s-izrazima [Ivanović, Budimac, 1989], te se mašinski program SECD mašine i sva četiri registra takodje mogu predstaviti s-izrazom [Henderson, 1980; Budimac, Ivanović, 1990a]. Pored toga za implementaciju lenje SECD mašine je potreban i recept - koji sadrži kôd koji treba da se izvrši i vrednosti identifikatora koji učestvuju u tom kodu. Struktura **SExp** može podržati i implementaciju recepta

ako se proširi novom oznakom **ReciSE**, koja ima istu strukturu kao i **ConsSE** i **TuplSE**.

Apstraktne mašine zasnovane na tranformaciji grafa (SK, superkombinatorska i G mašina) predstavljaju kombinatorski ili superkombinatorski izraz u obliku grafa, te ga zatim redukuju. Strukture podataka potrebne za implementaciju (super)kombinatorskog grafa, kao i zapisa kombinatorskog izraza, mogu realizovati strukturom **SExp**, na sledeći način.

Uz "obične" identifikatore, u (super)kombinatorskom grafu je potrebno razlikovati još i "promenljive" (**VarSE**) i (super)kombinatore (**ComSE**). Simbole koji predstavljaju promenljive je potrebno razlikovati od ostalih simbola u toku prevodjenja LL programa u kombinatorski term, a simboli koji predstavljaju kombinatore (mašinske komande SK i superkombinatorske mašine) su potrebni i tokom prevodjenja i tokom redukovanja grafa. Podatak da je neki simbol kombinator se čuva uz pojavu simbola (a ne u tabeli simbola) zbog toga što u superkombinatorskoj mašini ne postoji unapred poznati skup superkombinatora.

U grafovima postoji i čvor specijalne namene, koji predstavlja primenu kombinatorskog izraza na podatke, zbog čega se uvodi novi čvor (čvor primene) sa oznakom **ApplSE**.

U superkombinatorskoj mašini postoji i potreba za predstavljanjem superkombinatora koji je u prethodnoj glavi zapisivan kao: (($x x_1 ... x_n) e). On se može predstaviti listom od dva elementa, pri čemu prvi element lista sa imenom superkombinatora i nazivima argumenata, a drugi element je telo superkombinatora. Potencijalno efikasnija reprezentacija superkombinatora bi se dobila na sledeći način: ([$x . x_1 ... x_n] . e), uz efikasniju reprezentaciju n-torke nego što je ovde predloženo.

Za implementaciju G mašine se moraju implementirati i čvorovi grafa tipa **hole** (prazni, nedefinisani čvorovi) i tipa (k;C) koji označavaju superkombinator sa **k** argumenata i telom **C**. Kako se čvorovi tipa **ReciSE** ne koriste u okruženju G mašine, takvi čvorovi se mogu smatrati čvorovima tipa **hole** u okruženju G mašine. Superkombinatori (čvorovi tipa (k;C) se u okruženju G mašine mogu predstaviti na način sugerisan u prethodnoj glavi kao: (**$x k (C)**), pri čemu se broju argumenata pristupa kao drugom elementu navedene liste, a telu kao trećem. Da se radi o čvoru tipa (k;C) može se zaključiti ispitivanjem prvog elementa navedene liste (da li je kombinator).

```
TypeSE   = (IntSE, ReaSE, SymSE, ConsSE, TuplSE)

SExp     = POINTER TO Node;
Node     = RECORD
             (* podaci potrebni za
                skupljanje  otpadaka *)
             CASE T : TypeSE OF
                 IntSE : IntVal   : LONGINT |
                 ReaSE : ReaVal   : REAL  |
                 ComSE,
                 VarSE
                 SymSE : SymVal   : SymbTabEntry |
                 ApplSE,
                 ReciSE,
                 ConsSE,
                 TuplSE : Head,
                          Tail    : SExp
             END
           END;
```

Sl. 5.3 Konačna struktura implementacije LL-a

Na Sl. 5.3 je prikazana konačna struktura podataka **SExp** kojom se realizuju strukture podataka LL-a, izvorni kod LL programa, unutrašnje strukture svih pet opisanih mašina kao i mašinski programi apstraktnih mašina. Sledi rezime tipova čvorova u strukturi **SExp**:

- čvorovi tipa **ApplSE**, **ReciSE**, **VarSE** i **ComSE** se ne koriste u predstavljanju programa LL-a i njegovih struktura podataka,
- čvorovi tipa **ApplSE**, **VarSE** i **ComSE** se ne koriste u predstavljanju programa SECD mašine i njenih struktura podataka,
- čvorovi tipa **ReciSE** se ne koriste u predstavljanju programa SK i superkombinatorske mašine i njenih struktura podataka. Čvorovi tipa se **ConsSE** i **TuplSE** ne moraju koristiti unutar ovih mašina, jer se i jedna i druga struktura podataka mogu predstaviti pozivom odgovarajućih konstruktora podataka, čijim se izračunavanjem delovi složenih struktura odmah ispisuju i nikad ne konstruišu*.
- čvorovi tipa **VarSE** se ne koriste u predstavljanju programa G mašine i njenih struktura podataka, uz napomenu da se čvor tipa **ReciSE** koristi kao čvor tipa **hole** koji je potreban u implementaciji G mašine. Napomena za čvorove tipa **ConsSE** i **TuplSE** iz prethodnog pasusa, važi i za G mašinu.

5.2 Potrebni moduli za realizaciju LL sistema

Implementacija čitavog sistema medjujezika LL opisanog u ovoj tezi može da bude kompaktna i modularna, za šta postoje mogućnosti odabirom zajedničke unutrašnje strukture podataka za predstavljanje LL-a, njegovih struktura podataka, apstraktnih mašina i mašinskih programa.

Na Sl. 5.4 se nalazi prikaz jedne moguće raspodele modula za implementaciju sistema. Veze izmedju modula su izražene implicitno, od gore ka dole, pri čemu važi da moduli koji se nalaze niže na slici zavise od (dela) modula koji se nalaze iznad njih.

Sledi kratak opis svakog od modula:

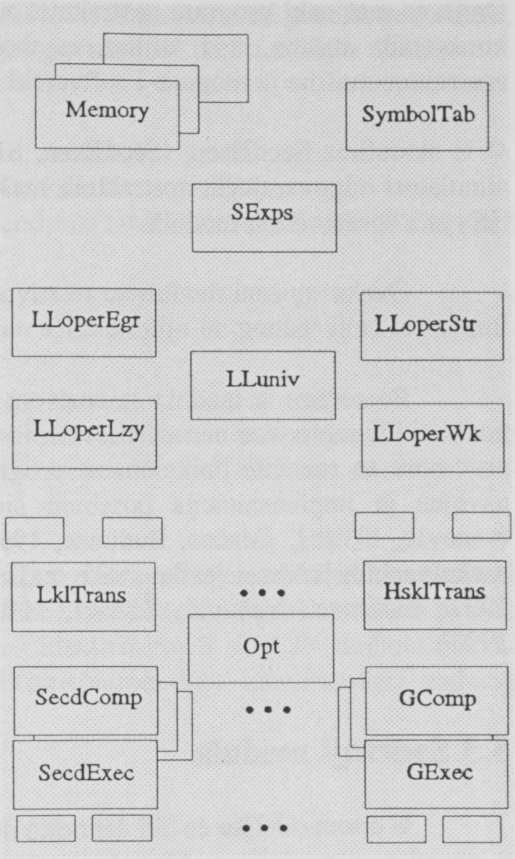
- U modulu **Memory** je definisan čvor osnovne strukture podataka (**SExp**), celokupan memorijski prostor kao skup čvorova (hrpa, engl. *heap*) i realizovane osnovne procedure za "zauzimanje" čvora i skupljanje otpadaka.
- U modulu **SymbolTab** je definisana tabela simbola i osnovne procedure.
- U modulu **SExps** su definisani s-izraz i osnovne procedure. Realizovane su opšte operacije koje su nezavisne od implementacije medjujezika LL. Ovaj modul je nastao na osnovu [Budimac, Ivanović, 1989].

---

* To medjutim nije slučaj sa implementacijama LL-a opisanim u ovoj tezi.

Modul **SExps** jedini direktno koristi identifikatore iz prethodna dva modula.

- U modulima **LLoperEgr** i **LLoperLzy** se implementiraju ugradjene funkcije LL-a čija se semantika razlikuje u zavisnosti od toga da li se posredstvom LL-a implementira jezik (redom) striktne ili nestriktne semantike. Ova dva modula implementiraju iste operacije, ali na različite načine.
- U modulima **LLoperWk** i **LLoperStr** se implementiraju ugradjene funkcije čija se semantika razlikuje u zavisnosti da li se posredstvom LL-a implementira (redom) netipizirani ili tipizirani jezik. U modulu **LLoperWk** se nalaze definicije ugradjenih funkcija u kojima se pre njihove primene proverava uskladjenost tipova podataka. U modulu **LLoperStr** se nalaze definicije ugradjenih funkcija u kojima se pre primene ne proverava uskladjenost tipova podataka.



Sl. 5.4 Struktura modula

- U modulu **LLuniv** su implementirane ostale ugradjene funkcije LL-a, koje ne zavise ni od (ne)striktne semantike ni od (ne)tipiziranosti izvornog jezika.

Svih pet nabrojanih modula koriste procedure iz modula **SExps** i oni će nadalje jednim imenom biti nazivani 'operatorskim' modulima.

- U modulima **LklTrans**, **IswTrans**, **SaslTrans**, **HsklTrans** itd. se implementiraju translatori za (redom) LispKit LISP, ISWIM, SASL, Haskell itd. u medjujezik LL.
- U modulu **Opt** su implementirane različite analize, provere i optimizacije koje se mogu izvršiti nad LL programom. Modul koristi procedure iz modula **SExps** i operatorskih modula.
- U modulima **SecdComp**, **lSecdComp**, **SKComp** itd. su implementirani prevodioci medjujezika LL u mašinske jezike (redom) apstraktnih mašina SECD, lenja SECD, SK itd. Ovi moduli takodje koriste procedure iz modula **SExps** i operatorskih modula.

Dalje se mašinski program apstraktnih mašina može translirati u mašinske jezike konkretnih mašina, nad kojima se mogu vršiti dalje optimizacije. Medjutim, apstraktne mašine je moguće i softverski simulirati.

o U modulima **SecdExec**, **lSecdExec**, **SKExec** itd. su implementirani programski simulatori odgovarajućih apstraktnih mašina. Moduli koriste procedure iz modula **SExps** i operatorskih modula.

Ovako opisani moduli su relativno nezavisni i omogućavaju da promena u implementaciji jednog od njih ne utiče na ostale module.

Procedure iz modula sa nazivom ***Trans**, ***Comp**, ***Exec** i **Opt** se mogu pozivati i kombinovati na različite načine u programima koji implementiraju jezičke procesore za različite funkcionalne programske jezike. Princip koji je dosada bio usvojen je implementacija posebnih simulatora apstraktnih mašina [Budimac, Ivanović, 1992b], [Maćos, Budimac, 1993] i implementacija prevodilaca izvornih funkcionalnih jezika u jezike svake mašine ponaosob [Ivanović, Budimac, Putnik, 1991; Budimac, Ivanović, Živkov, 1992; Budimac, Ivanović, Živkov, Maćoš, 1993].

5.3 Sadržaji modula

U ovom odeljku će biti dat nešto detaljniji spisak procedura implementiranih u navedenim modulima. Moguće implementacije nekih procedura će biti date u programskom jeziku Modula-2.

5.3.1 Modul Memory

Na Sl. 5.5 se nalazi spisak nekih od procedura definisanih u modulu **Memory**.

```
.....
PROCEDURE NewSE(VAR p: SExp);
PROCEDURE NotEnoughStorageSE(): BOOLEAN;
PROCEDURE MarkSE(p: SExp);
PROCEDURE CollectSE(VAR Deleted: CARDINAL);
PROCEDURE InitStorageSE;
PROCEDURE StructuredSE(t: TypeSE): BOOLEAN;
.....
```

Sl. 5.5 Neke operacije modula **Memory**

Procedura **InitStorageSE** kreira čitav memorijski prostor kao niz čvorova tipa **SExp**. Ova procedura treba da bude izvršena pre bilo koje druge. Procedura **NewSE** uzima prvi slobodan čvor iz niza slobodnih čvorova. Procedura **MarkSE(p)** markira sve čvorove povezane sa **p**. **CollectSE** sve nemarkirane čvorove proglašava slobodnim, a poziv funkcije **StructuredSE** ima vrednost TRUE, ako je tip čvora strukturni (**ReciSE**, **ConsSE**, **ApplSE** i **TuplSE** u predloženoj implementaciji). Procedurama **MarkSE** i **CollectSE** je implementiran tzv. "mark&sweep" skupljač otpadaka.

Konačna struktura čvora **SExp**, implementacija nekih od gornjih procedura, kao i njihov uticaj na brzinu izvršavanja programa, biće dati u odeljku 5.4.2).

5.3.2 Modul SymbolTab

U modulu **SymbolTab** je implementirana tabela simbola **SymbolTable**, čiji su elementi **SymbolTableEntry**.

```
TYPE
   SymbolTable;
   SymbolTableEntry;

PROCEDURE MakeNullSymTab(VAR ST: SymbolTable);
PROCEDURE ToCons (S: ARRAY OF CHAR; VAR ST: SymbolTable): SymbolTableEntry;
PROCEDURE GetSymbolTableName(E: SymbolTableEntry): ARRAY OF CHAR;
```

Sl. 5.6 Operacije definisane modulom **SymbolTab**

Na Sl. 5.6 se nalaze deklaracije tri potrebne procedure nad tabelom simbola za realizaciju LL-a i pratećih programa. Procedura **MakeNullSymTab** kreira praznu tabelu simbola, funkcija **ToCons** pronalazi identifikator **S** u tabeli simbola **ST**, ili ga u nju smešta. Funkcija **GetSymbolTableName** daje identifikatora zadatog elementa tabele simbola.

5.3.3 Modul SExps

U modulu je implementiran apstraktni tip podataka s-izraz.

Na Sl. 5.7 se nalaze deklaracije nekih od procedura definisanih u modulu **SExps**. Procedura **ReadSE** učitava s-izraz iz fajla **Fi**, a procedura **DispSE** prikazuje s-izraz **P** na ekranu. Procedura **ReHeSE** zamenjuje "glavu" s-izraza **E** izrazom **R**, procedura **SetTypeSE** postavlja tip čvora izraza **E** na **t**, a procedura **IsTypeSE** ispituje da li je **t** tip s-izraza **E**. Procedura **CreateSE** kreira "prazan" s-izraz, a procedura **ConsSE** kreira tačkasti s-izraz (tipa **ConsSE**) od s-izraza **P** i **Q**. Procedure **HeadSE** i **TailSE** pristupaju glavi i repu složenog s-izraza **P**, a procedura **ValcSE** pretvara brojni s-izraz **P** u prirodni broj.

```
......
PROCEDURE ReadSE(Fi: File): SExp;
PROCEDURE DispSE(P : SExp);
PROCEDURE ReHeSE(E: SExp; R: SExp);
PROCEDURE SetTypeSE(E: SExp; t: TypeSE);
PROCEDURE IsTypeSE(E: SExp; t: TypeSE): BOOLEAN;
PROCEDURE CreateSE() : SExp;
PROCEDURE HeadSE(P : SExp) : SExp;
PROCEDURE TailSE(P : SExp) : SExp;
PROCEDURE ConsSE(P,Q : SExp) : SExp;
PROCEDURE ValcSE(P : SExp) : CARDINAL;
......
```

Sl. 5.7 Neke operacije modula **SExps**

Na Sl. 5.8 se nalazi definicija procedure **GetTerm** za učitavanje s-izraza **E** iz fajla **Fi**. Od konstanti s-izraza procedura **GetTerm** prepoznaje identifikatore (medju kojima po prefiksu razlikuje kombinatore), cele i realne brojeve, posebni

```
PROCEDURE GetList(Fi: File; tpe: TypeSE; VAR E : SExp);
BEGIN
  NewSE(E); E^.T := tpe;
  GetTerm(Fi, E^.Head);
  IF Symbol = '.' THEN
     GetSymbol(Fi); GetTerm(Fi, E^.Tail)
  ELSE
     IF Symbol = ')' THEN
        E^.Tail := CreateSE();
     ELSIF NOT EOL THEN
        GetList(Fi, tpe, E^.Tail);
     ELSE
        WriteLn; WriteString('*** Error (GetList)');
        HALT;
     END
  END
END GetList;

PROCEDURE GetTerm(Fi: File; tpe: TypeSE; VAR E:SExp);
VAR Num: REAL;
BEGIN
  IF SymbolType = SCons THEN  (* identifikator *)
     NewSE(E);
     IF Symbol = "_C" THEN    (* kombinator *)
        E^.T := CombSE;
        GetSymbol(Fi);
        IF SymbolType <> SCons THEN
           WriteString('*** Error (GetTerm)'); HALT;
        END;
     ELSE
        E^.T := SymSE     (* "obican" identifikator *)
    END;
    E^.SymVal := ToCons(Symbol, ST);
    GetSymbol(Fi);
  ELSIF SymbolType = SNum THEN  (* broj *)
     NewSE(E);
     IF TypeH = IntSE THEN          (* ceo broj *)
        E^.T := IntSE;
        E^.IntVal := SymbolI;
     ELSE                           (* realan broj *)
        E^.T := ReaSE;
        E^.ReaVal := SymbolR;
     END;
     GetSymbol(Fi);
  ELSIF Symbol = '(' THEN
     GetSymbol(Fi);
     GetList(Fi, tpe, E);
     GetSymbol(Fi);
  END;
END GetTerm;
```

Sl. 5.8 Sintaksni analizator

simbol (na osnovu koga gradi liste. Da li će procedura graditi tačkaste izraze ili čvorove primene, zavisi od parametra procedure **tpe** (**ConsSE** ili **ApplSE**) - pri učitavanju LL programa i programa SECD mašine, procedura **GetTerm** treba da gradi tačkaste izraze, dok pri učitavanju (super) kombinatorskih grafova procedura treba da gradi čvorove primene. Procedura ne gradi čvorove ostalih tipova, jer oni postoje samo u internoj reprezentaciji s-izraza.

Procedura **GetSymbol** učitava jedan simbol iz fajla **Fi** i odredjuje mu tip: identifikator, broj (ceo ili realni) ili razdvajač.

5.3.4 Moduli LLoperWk i LLoperStr

U ova dva modula se definišu ugradjene funkcije LL-a čija se implementacija razlikuje po tome da li se pre primene funkcije proverava uskladjenost tipova argumenata (modul **LLoperWk**) ili ne (modul **LLoperStr**) (na primer, aritmetički i relacijski operatori). U oba modula su definisane iste ugradjene funkcije.

U principu bi se takve ugradjene funkcije mogle implementirati jedinstvenim procedurama uz dodatni parametar koji bi odredjivao da li se uskladjenost tipova proverava ili ne. Na taj način bi se medjutim potpuno izgubila prednost one implementacije u kojoj se ne ispituje uskladjenost tipova argumenata, te se zbog toga brže izvršava. Na Sl. 5.9 su prikazane definicije procedure **DivSE** iz oba modula.

```
(* DivSE iz modula LLoperWk *)
PROCEDURE DivSE(E1, E2 : SExp): SExp;
VAR
  E:SExp;
BEGIN
  IF IsTypeSE(E1, IntSE) AND IsTypeSE(E2, IntSE) THEN
     NewSE(E); SetTypeSE(E, IntSE);
     E^.IntVal := E1^.IntVal DIV E2^.IntVal;
  ELSE
     WriteLn; WriteString('*** Error (DivSE)'); HALT;
  END;
  RETURN E
END DivSE;

(* DivSE iz modula LLoperStr *)
PROCEDURE DivSE(E1, E2 : SExp): SExp;
VAR
  E:SExp;
BEGIN
  NewSE(E); SetTypeSE(E, IntSE);
  E^.IntVal := E1^.IntVal DIV E2^.IntVal;
  RETURN E
END DivSE;
```

Sl. 5.9 Operacije iz modula **LLoperWk** i **LLoperStr**

5.3.5 Moduli LLoperEgr i LLoperLzy

U ova dva modula su implementirane ugradjene funkcije LL-a koji se razlikuju po striktnosti semantike (ugradjene funkcije za rad sa listama, na primer).

Na Sl. 5.10 se nalaze dve verzije implementacije ugradjene funkcije **_nth**, u zavisnosti od toga da li je lista E (čiji se n-ti element traži) već izračunata ili nije (na osnovu [Budimac, Maćoš, 1993]). U drugom slučaju je "rep" pre pristupanja potrebno prethodno izračunati. U implementaciji funkcije **nth** iz modula **LLoperLzy** se podrazumeva da je lista neizračunata i predstavljena na način prikazan na Sl. 5.11b (u SK, super-kombinatorskoj i G mašini). Pozivi funkcija **_2of3** i **_3of3** za izraz oblika **(cons x y)** imaju vrednosti redom **x** i **y**, a poziv funkcije **Eval(t)** ima vrednost izračunatog (kombinatorskog) izraza **t**.

5.3.6 Modul LLuniv

U ovom modulu se implementirane one ugradjene funkcije jezika koje ne zavise ni od tipiziranosti ni od striktnosti izvornog čisto-funkcionalnog programskog jezika. Kao ilustracija neka posluži implementacija ugradjene funkcije **_foreign**, zasnovana na korutinama i simulatoru (interpretatoru) apstraktne mašine [Ivanović, Budimac, 1990; Budimac, Ivanović, 1991a; Budimac, Maćoš, 1993].

Da bi se pozivanje stranih funkcija realizovalo posredstvom korutina potrebno je kao korutine realizovati:

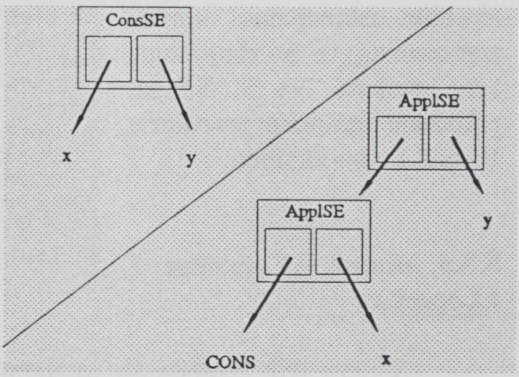
- simulator apstraktne mašine,
- stranu funkciju u programskom jeziku u kome je realizovan simulator apstraktne mašine (u našem slučaju Modula-2).

```
(* procedura iz modula LLoperEgr *)
PROCEDURE NthSE(E, n: SExp): SExp;
VAR i, upper: CARDINAL;
BEGIN
  upper := ValcSE(n) - 1;
  FOR i := 1 TO upper DO
      E := TailSE(E);
  END;
  RETURN CopySE(HeadSE(E));
END NthSE;

(* procedura iz modula LLoperLzy *)
PROCEDURE Nth(E, n: SExp): SExp;
VAR i, upper: CARDINAL;
    t: SExp;
BEGIN
  upper := ValcSE(n) - 1;
  FOR i := 1 TO upper DO
      t := _3of3(E);
      E := Eval(t);
  END;
  RETURN CopySE(_2of3(E));
END Nth;
```

Sl. 5.10 Dve implementacije operatora _nth

te uspostaviti "komunikacione kanale" izmedju njih.

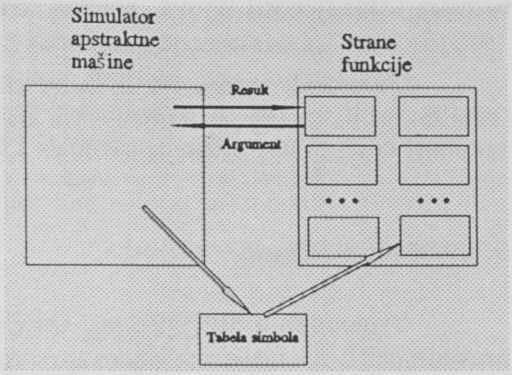


Sl. 5.11 Izračunati i zadržani tačkasti s-izraz

Na slici Sl. 5.12 je prikazana željena struktura korutina, pri čemu je **Argument** globalna promenljiva tipa SExp, kojom se prosledjuju argumenti stranoj funkciji, dok je **Result** globalna promenljiva tipa SExp, kojom se prosledjuje rezultat primene strane funkcije na argumente. Poziv odgovarajuće korutine se vrši na osnovu male tabele simbola, u kojoj su sadržana imena svih stranih funkcija i adrese odgovarajućih korutina.

Svaka strana funkcija treba da bude realizovana na način prikazan na Sl. 5.13a. Komanda apstraktnih mašina SWT, treba da bude realizovana na način prikazan na Sl. 5.13b. U implementaciji simulatora treba obezbediti odgovarajuće ažuriranje registara (ili grafa) pre i posle poziva procedure SWT za implementaciju istoimene mašinske komande. Tako na primer u simulatoru apstraktne SECD mašine treba uzeti odgovarajuće vrednosti sa steka S pre poziva procedure, a rezultat poziva procedure SWT ponovo vratiti na stek S, istovremeno ažurirajući stek C (po pravilu za komandu SWT na strani 93). Za detaljniji uvid u implementaciju stranih funkcija posredstvom SECD mašine videti [Ivanović, Budimac, 1990], a posredstvom SK

redukcione mašine videti [Budimac, Maćoš, 1993].



Sl. 5.12 Struktura korutina

5.3.7 Moduli *Trans

U ovim modulima su implementirani translatori izvornih čisto-funkcionalnih programskih jezika u medjujezik LL, po pravilima sadržanim u trećoj glavi ove teze (ili sličnim). Preporučuje se upotreba nekog od automatskih generatora leksičkih i sintaksnih analizatora ("kompajlera kompajlera"). Zadavanjem odgovarajućih semantičkih akcija se bez dodatnih aktivnosti može realizovati translacija izvornog programa u medjujezik LL.

Translator izvornog jezika učitava fajl sa izvornim programom, a kao rezultat daje rezultujući LL program (s-izraz). Taj s-izraz se dalje može ispisati u rezultujući fajl, proslediti prevodiocu u jezik neke apstraktne mašine ili proslediti nekoj proceduri za optimizaciju.

```
a)
PROCEDURE ForFun;
BEGIN
   LOOP
     Uzmi odgovarajući broj argumenata iz
          promenljive Argument;
     Izvrši željenu akciju;
     Stavi rezultat u promenljivu Result;
     Prebaci kontrolu simulatoru;
   END
END ForFun;

b)
PROCEDURE SWT(ImeFunkcije: ARRAY OF CHAR): SExp;
BEGIN
   Postavi argumente u promenljivu Argument;
   Pronadji adresu korutine za ImeFunkcije u
       tabeli simbola;
   Prebaci kontrolu odgovarajućoj korutini;
   RETURN Result;
END SWT;
```

Sl. 5.13 Implementacija operatora _foreign

U dodatku ove teze je prikazana kompletna implementacija translatora za programski jezik ISWIM, po pravilima iz odeljka 3.2, a implementirana pomoću kompajlera kompajlera COCO-2 [Dobler, Pirklbauer, 1990; Dobler, 1991]. U semantičkim akcijama se koriste neke od procedura iz prethodno pomenutih modula, koje će u dodatku biti ukratko opisane.

5.3.8 Modul Opt

U modulu se implementiraju procedure za optimizacije, analize i transformacije LL programa, kao što su na primer: semantička analiza [Putnik, Budimac, Ivanović, 1992a], analiza zavisnosti [Putnik, Ivanović, Budimac, 1992b],

provera uskladjenosti tipova, analiza striktnosti, analiza protoka podataka [Bloss, 1989] i ostale transformacije programa [Kelsey, 1989].

Parametar svake od ovih procedura je s-izraz (LL program), a njihov rezultat je ili s-izraz sa izmenjenim (optimizovanim) LL programom ili logička vrednost koja govori o (ne)ispravnosti LL programa.

5.3.9 Moduli *Comp

U modulima SECDComp, lSECDComp, SKComp itd. su implementirani prevodioci LL-a u mašinske jezike apstraktnih mašina. Prevodioci kao svoj argument uzimaju s-izraz sa LL programom (moguće optimizovanim i analiziranim), a kao svoj rezultat daju s-izraz koji sadrži rezultujući program u jeziku odgovarajuće apstraktne mašine.

Kao ilustrativne primere moguće implementacije prevodioca, navodimo na Sl. 5.14 jedan primer realizacije jednostavnog algoritma za apstrahovanje promenljivih iz LL izraza oblika (_lambda (x_1 ... x_k) e).

Algoritam je jednostavan i može se (po ugledu na pravila navedena na strani 99) izraziti na sledeći način:

```
PROCEDURE Ba (x, e: SExp): SExp;
BEGIN
 IF AtomSE(e) AND EqSE(x, e) THEN
    RETURN QcSE('I')
 ELSIF AtomSE (e) THEN
    RETURN ConsSE (QcSE ('K'), e);
 ELSE
    RETURN ConsSE
              (ConsSE(QcSE ('S'),
                   Ba (x, HeadSE (e))),
              Ba (x, TailSE (e)))
 END
END Ba;
```

Sl. 5.14 Implementacija jednostavnog alg. za apstrahovanje

[x]x = I
[x]y = K y, $x \neq y$
[x]E F = S [x]E [x]F

Implementacija algoritma za apstrahovanje identifikatora predloženog u odeljku 4.3.1 je složenija i obuhvata ispitivanje pripadnosti identifikatora koji se apstrahuje u nekom od podizraza. Na Sl. 5.15 je prikazana globalna struktura implementacije prevodioca LL-a u mašinski jezik SECD mašine.

Procedura **Compile** je implementacija pravila P' za prevodjenje LL-a u jezik SECD mašine (strana 89), dok je procedura **Comp** implementacija pravila P iz istog odeljka. Procedura **Comp** ima tri argumenta: **E** je LL izraz koji se prevodi, **N** je lista imena u odnosu na koju se izraz **E** prevodi, a **C** je sekvenca komandi SECD mašine, koja treba da se doda na rezultat prevodjenja izraza **E** (čime se implementira operator | iz pravila P).

Gornje dve procedure su realizovane na osnovu [Budimac, Ivanović, 1990a] [Ivanović, Budimac, Putnik, 1991; Maćoš, Budimac, 1992; 1993].

5.3.10 Moduli *Exec

U modulima SECDExec, ISECDExec, SKExec itd. se implementiraju simulatori (interpretatori) odgovarajućih apstraktnih mašina. Simulatori kao svoj argument uzimaju s-izraz sa mašinskim program apstraktne mašine, ukoliko je potrebno zahtevaju unošenje argumenata, te potom simuliraju izvršavanje programa. Rezultat izvršavanja se ispisuje na ekranu ili zapisuje u fajl.

```
PROCEDURE Comp(E,N,C:SExp):SExp;
BEGIN
  IF (kw = KeywordSE[ _quote ]) THEN
     ....
  ELSIF kw = KeywordSE[ _add ] THEN
     ....
  ELSIF kw = KeywordSE[ _le ]  THEN
     RETURN Comp(HeadSE(TailSE(E)),
                 N,
                 Comp(HeadSE(TailSE(TailSE(E))),
                      N,
                      ConsSE(QsSE('LE'),C)))
  ELSIF kw = KeywordSE[ _car ]  THEN
     RETURN Comp(HeadSE(TailSE(E)),
                 N,
                 ConsSE(QsSE('CAR'),C))
     ....
  END
END Comp;

PROCEDURE Compile(E:SExp):SExp;
BEGIN
  RETURN Comp(E, CreateSE(), QuoteSE('(AP STOP)'));
END Compile;
```

Sl. 5.15 Struktura implementacije prevodioca u jezik SECD mašine

Na Sl. 5.16a se nalazi implementacija funkcije **Eval(t)** koja izračunava kombinatorski izraz **t** (odnosno mašinski program SK redukcione mašine). Na Sl. 5.16b se nalazi implementacija pravila za redukovanje ugradjenog kombinatora ADD.

Definicijom funkcije **Eval** su implementirana pravila za redukovanje kombinatorskog izraza R, definisana u odeljku 4.3.2. Kombinatorski izraz je predstavljen grafom (s-izrazom), čijim elementima se pristupa pomoću steka **Spine**. Procedura **OneRed** u zavisnosti od kombinatora koji se nalazi na vrhu steka redukuje kombinatorski izraz po pravilima za redukciju R.

Procedura **RedAdd** redukuje kombinatorski izraz oblika ADD *E F*, koristeći proceduru **AddSE** (definisanu u modulima **LLoperWk** i **LLoperStr**).

Gornje dve procedure su preuzete iz [Maćoš, Budimac, 1993].

5.4 Analiza performansi izvršavanja LL programa

Performanse izvršavanja LL programa je moguće analizirati sa više različitih aspekata, te na osnovu rezultata popravaljati implementaciju, unutrašnje strukture podataka, te eventualno predložiti i drugačije ugradjene operacije medjujezika, koje je moguće efikasnije implementirati.

Kao ilustraciju mogućih analiza, u narednim odeljcima navodimo samo dve od izvršenih analiza, od kojih je prva vezana za upravljanje memorijom na računarskoj arhitekturi sa segmentiranom memorijom, a druga za različite tehnike implementacije i optimizacije jedne apstraktne računarske arhitekture. Odnos izmedju implementacije LL-a posredstvom SECD i SK mašine je kratko prikazan u

```
a)
PROCEDURE Eval (VAR G: SExp): SExp;
VAR c: SExp;
BEGIN
   INC(STop);              (* pamćenje trenutnog vrha steka *)
   Spine[STop].no := SBase;
   SBase := STop;

   UnWind(Graph);                    (* razvijanje grafa *)
   c := Spine[STop].ex;

   WHILE IsTypeSE(c, Combinator) AND (* redukovanje grafa *)
       (GetSymbolTableEntry(c) <>
              KeywordSE[ cons ]) DO
       OneRed;
       c := Spine[STop].ex;
   END;

   STop := SBase;                  (* restauracije steka *)
   SBase := Spine[STop].no;
   DEC(STop);
   RETURN Graph;
END Eval;

b)
PROCEDURE RedAdd;
VAR p1, p2: SExp;
BEGIN
   p1 := TailSE(Spine[STop-1].ex);  (* 1. argument *)
   p2 := TailSE(Spine[STop-2].ex);  (* 2. argument *)
   UpdateSE(Spine [STop - 2].ex,
            AddSE(Eval(p1),Eval(p2)));  (* rezultat *)
   DEC(STop, 2)
 END RedAdd;
```

Sl. 5.16 Delovi simulatora SK mašine

[Maćoš, Budimac, 1992].

Pre prikaza izvršenih analiza kratko opisujemo skup programa uz pomoć kojih su analize izvršene.

5.4.1 Testni programi

Programi korišćeni za testiranje implementacija LL-a su odabrani tako da obuhvate standardne programe za testiranje performansi, programe koji intenzivno koriste strukture podataka, programe koji koriste beskonačne strukture podataka i programe koji koriste funkcije višeg reda. Programi koji koriste beskonačne strukture podataka se odlikuju velikim iskorišćenjem stekova (za praćenje stanja izvršavanja programa) i malim iskorišćenjem "hrpe"*. Programi koji rukuju

* Hrpom se u SECD i G mašinama predstavljaju sadržaji registara mašine, a u SK i super-kombinatorskoj mašini (super)kombinatorski grafovi koji se redukuju. Stekovima se eksplicitno predstavlja stanje redukovanja (super)kombinatorskog grafa u SK i super-kombinatorskoj mašini, a implicitno u SECD i G mašini (internim stekom implementacionog jezika).

funkcijama višeg reda se odlikuju velikim iskorišćenjem hrpe i malim iskorišćenjem stekova. Na taj način odabrani skup testnih programa bolje odslikava sve aspekte implementacije (medju)jezika, nego što je to slučaj sa mnogim drugim analizama (na primer [Koopman, Lee, Siewiorek, 1992; King, 1991]).

Standardnim testnim programima nazivamo programe za izračunavanje n-tog fibonačijevog broja (**fib**), za izračunavanje broja poziva funkcije prilikom izračunavanja fibonačijevih brojeva (**nfib**), za izračunavanje Takeučijeve funkcije (**tak**) i za pronalaženje rešenja problema postavljanja kraljica na šahovskoj tabli (**queen**). Program koji intenzivno rukuje strukturama podataka je interpretator mini-BASIC programa (**basic**) [Ivanović, Stojković, Budimac, Putnik, 1985]. Programi koji koriste beskonačne strukture podataka su programi za izračunavanje n-tog elementa beskonačne liste (**nthLazy**), prvih n elemenata beskonačne liste (**nthFirst**) i n-tog prostog broja algoritmom Eratostenovog sita koje je predstavljeno beskonačnom listom (**nthPrime**). Program koji koristi funkcije višeg reda izračunava proizvod elemenata na dijagonali matrice koja je predstavljena funkcijom višeg reda (**diag**).

5.4.2 Upravljanje memorijom

Implementacija translatora izvornih funkcionalnih jezika u LL, prevodioca LL-a u jezike apstraktnih mašina i simulatora apstraktnih mašina se zasnivaju na istoj strukturi podataka - čvoru **SExp** (Sl. 5.3). Zauzimanje tih čvorova, popunjavanje i kasnije oslobadjanje kad više nisu potrebni, predstavljaju verovatno najznačajniju (a sigurno najčešću) aktivnost tokom izvršavanja bilo kog dela čitavog sistema. Čvorovi koji su dodeljeni programu se smeštaju u (logički) deo memorije koji se naziva hrpom. Osnovna karakteristike hrpe su da je njena veličina ograničena samo fizičkom količinom raspoložive memorije i da se u njoj memorijski prostor zauzima tokom izvršavanja programa, posebnim zahtevom.

U modulu **Memory** je implementirana hrpa i osnovne operacije za upravljanje hrpom: inicijalizacija, zauzimanje jednog čvora, ispitivanje da li je hrpa popunjena i implementacija skupljača otpadaka. U implementaciji koja će biti razmatrana u ovom odeljku, skupljač otpadaka je implementiran posredstvom dve procedure: za markiranje svih čvorova dostupnih iz programa i za oslobadjanje svih nemarkiranih (tj. programu nedostupnih) čvorova.

Ovde prikazana analiza je dopuna analize prikazane u [Budimac, Ivanović, 1992c] i prikazuje tri karakteristične implementacije hrpe. Računar na kome je ispitivan uticaj implementacije hrpe na performanse programa koji se izvršava, je mikro-računar zasnovan na mikro-procesorima klase 80x86, sa operativnim sistemom MS-DOS. Ovaj operativni sistem podržava samo segmentiranu memoriju u kome je najveći kontinualni blok od 64kb, što ograničava implementaciju hrpe. U daljem tekstu će biti prikazane tri moguće implementacije i prikazani rezultati merenja izvršavanja istih programa na sve tri implementacije hrpe.

Operacije koje će se razmatrati su prikazane na Sl. 5.5, a njihovo značenje je objašnjeno prilikom opisivanja modula **Memory**.

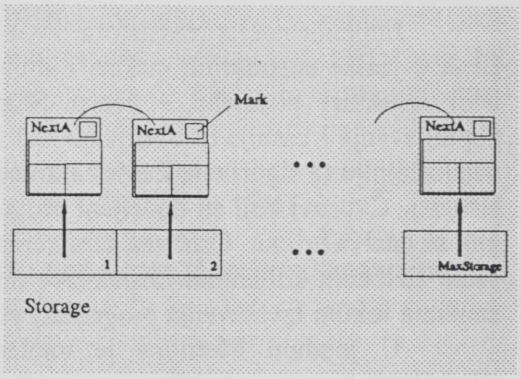
```
CONST
    MaxStorage = 15000;
TYPE
    TypeSE = ....
    SExp = POINTER TO Node;
    Node = RECORD
             Mark: BOOLEAN;
             NextA: [0..MaxStorage];
             CASE T : TypeSE OF
              ....
              ConsSE, ReciSE,
              ApplSE: Head,
                      Tail: [0..MaxStorage]
             END
           END;
VAR
  Storage:   ARRAY[1..MaxStorage] OF SExp;
  NextAvail: [1..MaxStorage]
```

Sl. 5.17 Statička implementacija hrpe

5.4.2.1 Prvi način - statička implementacija

Hrpa je implementirana nizom pokazivača na čvorove i prikazana je na Sl. 5.17, pri čemu su delovi koji su istovetni sa delovima prikazanim na Sl. 5.3 izostavljeni.

Ukupna dužina niza ne sme da prelazi 64 Kb (zbog ograničenja mikroprocesora i operativnog sistema), te se zbog toga u niz smeštaju pokazivači, a ne sâmi elementi. U takav niz se može smestiti oko 16400 "dalekih" pokazivača (engl. *far pointers*), koji su veličine 4 bajta. Nadalje ćemo pretpostaviti da niz ima 15000 elemenata, kako je na slici i prikazano.



Sl. 5.18 Struktura hrpe.

Veličina jednog čvora je 8 bajtova (po 1 bajt za polje tipa BOOLEAN i TypeSE, po dva bajta za polje tipa [0..MaxStorage] i po 4 bajta za pokazivače ili polja tipa REAL i LONGINT).

Na Sl. 5.18 je prikazana struktura hrpe. Polje **Mark** se koristi pri markiranju dostižnih čvorova i postavlja se na vrednost TRUE, ako je čvor dostižan iz neke od promenljivih programa, dok inicijalno ima vrednost FALSE. Polje **NextA** uvek "pokazuje" na sledeći slobodni čvor u nizu (ovim poljem se implementira tzv. lista slobodnih čvorova). Promenljiva **NextAvail** uvek pokazuje na prvi slobodni čvor.

Na Sl. 5.19 je prikazana implementacija četiri karakteristične procedure modula **Memory**.

Procedura **InitStorageSE** kreira "prazne" čvorove i postavlja lanac slobodnih čvorova. Lanac je povezan od zadnjeg ka prvom čvoru. Funkcija **NotEnoughMemory()** (koja nije prikazana na slici) ispituje da li je **NextAvail** ispod unapred odredjene granice, u kom slučaju ima vrednost TRUE. Procedura **NewSE** zauzima čvor sa indeksom **NextAvail**, koji potom ažurira da pokazuje na sledeći prvi slobodan čvor. Primetimo da je u slučaju kada je **NextAvail**<0 moguće automatski pozvati skupljanje otpadaka, ali bi u tom slučaju ovoj proceduri morale

```
PROCEDURE InitStorageSE;
VAR i: CARDINAL; p: SExp;
BEGIN
   FOR i := 1 TO MaxStorage DO
      NEW(p);
      Storage[i] := p;
      p^.NextA := i-1;
   END;
   NextAvail := MaxStorage
END InitStorageSE;

PROCEDURE NewSE(VAR p: SExp);
BEGIN
   p := Storage[NextAvail];
   NextAvail := p^.NextA;
   p^.Mark := FALSE;
   IF NextAvail <= 0 THEN
      WriteString(" Storage!"); HALT;
   END;
END NewSE;
```

```
PROCEDURE MarkSE(p: SExp);
BEGIN
   p^.Mark := TRUE;
   IF StructuredSE(p) THEN
      MarkSE(Storage[p^.Head];
      MarkSE(Storage[p^.Tail]
   END
END MarkSE;

PROCEDURE CollectSE;
VAR last, i: [0..MaxStorage];
BEGIN
  last := 0;
  FOR i := 1 TO MaxStorage DO
     IF NOT Storage[i]^.Mark THEN
        Storage[i]^.NextA := last;
        last := i;
     ELSE
        Storage[i]^.Mark := FALSE
     END;
  END;
  NextAvail := last
END CollectSE;
```

Sl. 5.19 Realizacija procedura iz modula **Memory** - 1. način

biti poznate sve promenljive korisničkog programa koje treba sačuvati (markirati). Procedura **MarkSE** markira sve čvorove dostupne iz promenljive **p**, a procedura **CollectSE** nemarkirane čvorove vraća u listu slobodnih čvorova.

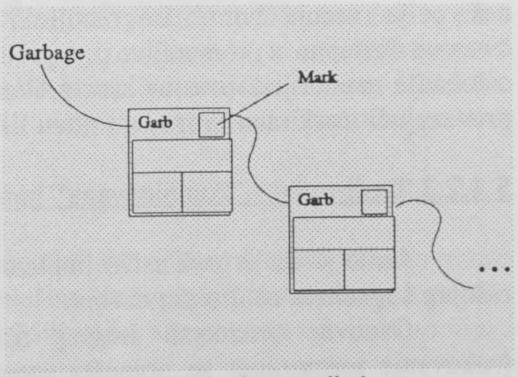
5.4.2.2 Drugi način - dinamička implementacija

Hrpa je implementirana povezanom listom čvorova i prikazana je na Sl. 5.20, pri čemu su delovi koji su istovetni sa delovima prikazanim na Sl. 5.3 izostavljeni.

```
TYPE
    TypeSE = ....
    SExp = POINTER TO Node;
    Node = RECORD
              Mark: BOOLEAN;
              Garb: SExp;
              ....
           END;
 VAR
   Garbage:   SExp;
```

Sl. 5.20 Dinamička implementacija hrpe

U ovom slučaju ne postoji unapred kreirani skup čvorova koji se kasnije samo dodeljuju, već se svaki čvor kreira tek onda kada je potreban i vezuje u povezanu listu sa ostalim čvorovima. Ovo povezivanje je potrebno da bi se tokom skupljanja otpadaka moglo pristupiti svakom kreiranom čvoru. Veličina jednog čvora je 14 bajtova.



Sl. 5.21 Dinamička implementacija hrpe

Na Sl. 5.21 je prikazana dinamička struktura hrpe. Polje **Mark** se koristi pri markiranju dostižnih

čvorova i postavlja se na vrednost TRUE, ako je čvor dostižan iz neke od promenljivih, dok inicijalno ima vrednost FALSE. Polje **Garb** pokazuje na sledeći zauzeti čvor u nizu. Promenljiva **Garbage** pokazuje na početak liste zauzetih čvorova. Broj čvorova je ograničen samo veličinom raspoložive memorije i pri 640 kb memorije, moguće je kreirati oko 32000 čvorova.

```
PROCEDURE NotEnoughStorage(): BOOLEAN;
BEGIN
   IF NOT Available(10000) THEN
      RETURN TRUE
   ELSE
      RETURN FALSE
   END:
END NotEnoughStorage;

PROCEDURE NewSE(VAR p: SExp);
BEGIN
   NEW(p);
   p^.Mark := TRUE;
   p^.Garb := Garbage;
   Garbage := p;
END NewSE;

PROCEDURE MarkSE(p: SExp);
BEGIN
   p^.Mark := TRUE;
   IF StructuredSE(p) THEN
      MarkSE(p^.Head);
      MarkSE(p^.Tail)
   END
 END MarkSE;
```

```
PROCEDURE CollectSE;
VAR curr, behind, delete: SExp;
BEGIN
  curr := Garbage;
  behind := curr;
  WHILE curr <> NIL DO
     IF NOT curr^.Mark THEN
        IF curr <> Garbage THEN
           delete := curr;
           curr := curr^.Garb;
           DISPOSE(delete);
           behind^.Garb := curr
        ELSE
           delete := curr;
           curr := curr^.Garb;
           behind := curr;
           DISPOSE(delete);
           Garbage := curr
        END
     ELSE
        curr^.mark := FALSE;
        behind := curr;
        curr := curr^.Garb
     END
  END
END CollectSE;
```

Sl. 5.22 Realizacija procedura iz modula **Memory** - 2. način

Na Sl. 5.22 je prikazana implementacija četiri karakteristične procedure modula **Memory**. Procedura **InitStorageSE** (koja nije prikazana na slici) postavlja promenljivu **Garbage** na NIL. Funkcija **NotEnoughStorageSE()** poziva standardnu Modula-2 proceduru **Available(n)** koja ispituje da li postoji **n** slobodnih bajtova. Procedura **NewSE** kreira novi čvor pozivom standardne procedure **NEW**, popunjava neka polja i vezuje čvor u listu postojećih čvorova. Procedura **MarkSE** markira sve čvorove dostupne iz promenljive **p**, a procedura **CollectSE** obilazeći povezanu listu, oslobadja memorijski prostor zauzet čvorovima koji nisu markirani, istovremeno prevezujući markirane čvorove u novu listu.

5.4.2.3 Treći način - "kombinovana" implementacija

Jasno je da je prvi način implementacije hrpe komplikovaniji, efikasniji i manjeg kapaciteta od drugog načina.

Osnovne prednosti drugog načina su jednostavnost implementacije (rukovanje memorijom se prepušta standardnim procedurama implementacionog jezika) i veća količina čvorova koje je moguće kreirati. Prednost prvog načina je efikasnost, pogotovu brzina zauzimanja čvora, koja je uvek konstantna.

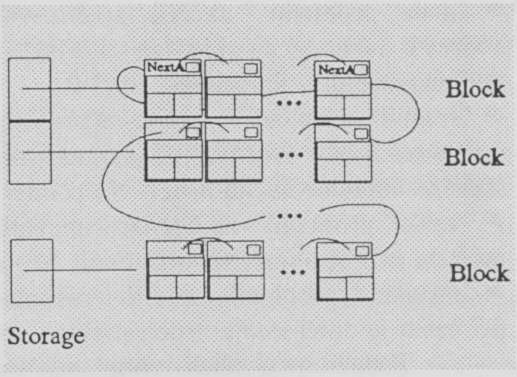
Treći način kombinuje prva dva utoliko što se umesto pojedinačnih čvorova kreiraju blokovi koji se ponaosob posmatraju kao nizovi. Ovakav način bi trebao da kombinuje dobre strane prethodna dva pristupa. Struktura je prikazana na Sl. 5.23.

```
CONST
   MxBl = 2500;
   MaxStorage = 1000;
TYPE
   TypeSE = ....
   NA = RECORD
           St: [0..MaxStorage];
           Bl: [0..MxBl]
        END;
   SExp     = POINTER TO Node;
   Node     = RECORD
                 Marked : BOOLEAN;
                 NextA: NA;
                 ....
              END
   Block    = POINTER TO ARRAY[1..MxBl] OF Node;
VAR
   Storage : ARRAY[1..MaxStorage] OF Block;
   NextAvail : NA;
```

Sl. 5.23 Kombinovana implementacija - 3. način

Veličina jednog čvora je 13 bajtova, a moguće je kreirati oko 31500 čvorova. Polje **Mark** ima istu ulogu kao u prethodne dve implementacije, a polje **NextA** uvek "pokazuje" na sledeći slobodni čvor u nizu (ovim poljem se realizuje tzv. lista slobodnih čvorova). Promenljiva **NextAvail** uvek pokazuje na prvi slobodni čvor u hrpi. "Adresa" sledećeg slobodnog čvora (polje **NextA** i promenljiva **NextAvail**) se sada sastoji od rednog broja bloka u kome se čvor nalazi i rednog broja čvora u okviru bloka.

Na Sl. 5.24 je prikazana struktura ove implementacije hrpe. Na Sl. 5.25 su prikazane dve procedure iz implementacije. Procedura **InitStorageSE** dodeljuje blokove dokle god je to moguće (tj. dokle god ima slobodne memorije). Promenljiva **RealMaxStorage** sadrži indeks poslednjeg alociranog bloka. Procedura **CollectSE** vraća u listu slobodnih čvorova sve nemarkirane čvorove. Procedura je analogna odgovarajućoj proceduri iz prvog opisanog načina implementacije, s tim što je adresiranje nešto komplikovanije.



Sl. 5.24 Implementacije hrpe

5.4.2.4 Analiza

U tabeli na Sl. 5.26 su prikazani rezultati merenja brzine izvršavanja testnih programa simuliranjem apstraktne SECD mašine, sa sve tri implementacije hrpe. Svi testni programi su pozivani sa takvim vrednostima argumenata, da su formirane četiri grupe:

```
PROCEDURE InitStorageSE;
VAR i,j: CARDINAL;
BEGIN
  i := 1;
  WHILE (i <= MaxStorage) AND
     Available(TSIZE(Node)*MaxBlock)DO
     NEW(Storage[i]);
     FOR j := 1 TO MaxBlock DO
      IF j=1 THEN
        Storage[i]^[j].NextA.St := i-1;
        Storage[i]^[j].NextA.Bl := MxBl
      ELSE
        Storage[i]^[j].NextA.St := i;
        Storage[i]^[j].NextA.Bl := j-1
      END;
     END;
     INC(i)
  END;
  RealMaxStorage := i - 1;
  NextAvail.St := RealMaxStorage;
  NextAvail.Bl := MxBl;
END InitStorageSE;
```

```
PROCEDURE CollectSE;
VAR
   i: [1..MaxStorage];
   j: [1..MaxBlock];
   zadnji: NA;
BEGIN
   zadnji.St := 0;
   zadnji.Bl := MxBl;

   FOR i := 1 TO RealMaxStorage DO
     FOR j := 1 TO MxBl DO
       IF (Storage[i]^[j].Mark THEN
         Storage[i]^[j].NextA := zadnji;
         zadnji.St := i;
         zadnji.Bl := j;
        ELSE (* va'e)i slogovi *)
         IF Storage[i]^[j].Mark THEN
            Storage[i]^[j].Mark := FALSE
         END;
        END;
     END;
   END;
   NextAvail := zadnji;
END CollectSE;
```

Sl. 5.25 Neke procedure za implementaciju hrpe

- "mali" problemi - za čije izračunavanje je potrebno ukupno manje od 15000 čvorova (tako da se skupljač otpadaka ne pozove nijednom ni na jednoj od tri implementacije). Na primer, **fib(12)**.
- "srednji" problemi - za čije izračunavanje je potrebno ukupno više od 15000, a manje od 30000 čvorova (tako da se skupljač otpadaka pozove samo pri testiranju statičke implementacije hrpe). Na primer, **fib(14)**.
- "veliki" problemi - za čije izračunavanje je potrebno više od 100000 čvorova, ali tako da je broj obraćanja hrpi mali. Na primer, **fib(20)**, **nth**, **nthPrime**, **nthFirst**.
- "ogromni" problemi - za čije izračunavanje je potrebno više od 100000 čvorova, pri čemu je opterećenje hrpe relativno veliko. Na primer, **basic** i **diag**.

Podaci su u tabeli (zbog veće opštosti) prikazani medjusobnim odnosom, umesto apsolutnim veličinama. Najbolje vreme je obeleženo oznakom ✓, a ostala vremena su data procentualnom razlikom u odnosu na najbolje vreme.

U prvom delu tabele su prikazana vremena potrebna za zauzimanje jednog čvora, za skupljanje otpadaka po jednom čvoru i markiranje jednog čvora. Podaci su dobijeni izolovanim merenjem sve tri aktivnosti. Pri računanju vremena potrebnog za skupljanje otpadaka, nije pravljena razlika izmedju markiranih i nemarkiranih čvorova.

Statičkoj implementaciji je za sve tri aktivnosti potrebno najmanje vremena, dok su velika usporenja u trećoj, kombinovanoj implementaciji, uzrokovana jedino komplikovanijim označavanjem lanca slobodnih čvorova. Prosečno vreme markiranja je isto u sve tri implementacije, jer se pri markiranju ne koristi reprezentacija hrpe. Vreme zauzimanja čvora u statičkoj i kombinovanoj implementaciji je uvek konstantno, dok bi se u dinamičkoj implementaciji

Osnovni podaci	Statička implementacija	Dinamička implementacija	Kombinovana implementacija
Veličina čvora	8 bajtova	14 bajtova	13 bajtova
Maks. broj čvorova	15000	32000	31500
t_{cr}	✓	+37.5%	+8.5%
t_{co}	✓	+529%	+102%
t_m	✓	✓	✓
Odnosi vremena			
$C = t_{cr} / t_{co}$	17	5.5	8.9
$M = t_{cr} / t_m$	4	5.5	4.25
Brzina izvršavanja			
"mali" problemi	✓	+13%	+1%
"srednji" problemi	+5%	+27%	✓
"veliki" problemi	✓	+74%	+4%
"ogromni" problemi	-	+145%	✓

Legenda:
t_{cr} - prosečno vreme potrebno za kreiranje jednog čvora
t_{co} - prosečno vreme potrebno za "skupljanje" jednog čvora
t_m - prosečno vreme potrebno za markiranje jednog čvora

Sl. 5.26 Poredjenje različitih realizacija skupljača otpadaka

povećavalo sa povećanjem segmentiranosti memorije (odnosno brojem aktiviranja sakupljača otpadaka*).

U drugom delu tabele su dati odnosi izmedju vremena potrebnog za kreiranje i skupljanje čvora, odnosno za kreiranje i markiranje čvorova. Podaci su dobijeni na osnovu apsolutnih vrednosti dobijenih u prethodnom merenju.

U trećem delu tabele su data vremena izvršavanja po "veličini" testnih programa. Statička implementacija hrpe omogućava najbrže izvršavanje "malih" i "velikih" zadataka, dok je nešto sporija od kombinovane implementacije pri izvršavanju "srednjih" zadataka, zbog aktiviranja skupljača otpadaka, koji kombinovanoj implementaciji nije potreban. "Ogromni" zadaci ne mogu da se rešavaju pri statičkoj implementaciji hrpe, zbog malog broja čvorova raspoloživog u jednom trenutku.

* Mada se segmentiranost memorije ne povećava značajno prolascima skupljača otpadaka, zbog poznate pojave da kasnije kreirane ćelije ranije postaju nepotrebne.

Kombinovana implementacija hrpe je samo neznatno sporija u izvršavanju "malih" i "velikih" zadataka. I pored duplo češćeg pozivanja skupljača otpadaka, statička implementacija je i dalje brža od kombinovane.

Može se zaključiti da je statička implementacija najbolja na računarima sa nesegmentiranom memorijom i kada je moguće kreirati dovoljno velike nizove. Kombinovana implementacija je najbolja na računarima sa segmentiranom memorijom, dok je jedina prednost dinamičke implementacije jednostavnost njene realizacije, te stoga može poslužiti samo kao prototip.

Pokažimo kako implementacija hrpe utiče na performanse rezultujećeg programa (translatora, prevodioca ili simulatora apstraktne mašine). Označimo sa Δt maksimalno moguće usporenje izvršavanja rezultujućeg programa koje zavisi od implementacije hrpe, sa n broj raspoloživih čvorova implementacije (kapacitet) i sa m^1, m^2, ... brojeve čvorova markiranih u prolascima skupljača otpadaka. Tada sledeća formula odslikava ciklus rada hrpe: kreiranje n čvorova, markiranje m^1 čvorova, sakupljanje kroz n čvorova; kreiranje $n\text{-}m^1$ čvorova itd.:

$$\Delta t = n\, t_{cr} + m^1\, t_m + n\, t_{co} + (n\text{-}m^1)t_{cr} + m^2\, t_m + n\, t_{co} + \ldots =$$
$$= t_{cr}(n + n - m^1 + n - m^2 \ldots) + t_m(m^1 + m^2 + \ldots) + t_{co}(n + n + \ldots)$$

Označimo sa g broj pozivanja skupljača otpadaka, a sa $m = \max\{m^1, m^2, \ldots, m^g\}$. Tada:

$$\Delta t \leq t_{cr}(n + g\, n - g\, m) + t_m\, g\, m + t_{co}\, g\, n \qquad (1)$$

pri čemu se g može shvatiti kao mera proporcionalnosti sa veličinom problema koji se rešava i jednak je $g = \lfloor (u\text{-}n)/(n\text{-}m) \rfloor + 1$, a u je ukupan broj čvorova koje je potrebno kreirati tokom izvršavanja programa. Eksperimenti pokazuju da je čest slučaj da $m^1 = m^2 = \ldots = m^g$ pogotovo kod simulatora apstraktnih mašina koji izvšavaju programe koji ne koriste intenzivno funkcije višeg reda. Inače je opšti slučaj $m^1 < m^2 < \ldots < m < \ldots m^g$. Formula (1) (i njen transformisani oblik (2) nešto kasnije) mogu poslužiti kao jednostavna procena efikasnosti implementacije hrpe uvek kada je količina potrebnih podataka u hrpi u svakom trenutku približno iste veličine.

Formula (1) se može napisati i u sledećem obliku (koristeći odnose vremena t_{cr} sa t_m i t_{co} (M i C)).

$$\Delta t \leq t_{cr}[n + g\, n - g\, m + (g\, m)/M + (g\, n)/C] \qquad (2)$$

Dalja aproksimacija procene efikasnosti implementacije se može dobiti pretpostavkom da $M=C=1$. Jedini uslov koji treba da bude ispunjen da bi aproksimacija bila valjana je da $t_{cr} > t_m$ i $t_{cr} > t_{co}$, što je gotovo uvek slučaj (a i eksperimentalni rezultati potvrdjuju). Tada

$$\Delta t \leq t_{cr} n(1 + 2g) \quad (3)$$

pri čemu se mora imati u vidu da je aproksimacija (3) tačnija što su **M** i **C** bliži jedinici.

5.4.3 Analiza implementacije simulatora SK redukcione mašine

Osnovnu implementaciju SK redukcione mašine (opisanu u odeljku 4.3) je moguće poboljšati na nekoliko načina: uvodjenjem novih kombinatora [Turner, 1981b], daljim prevodjenjem u specijalizovanije apstraktne mašine [Scheevel, 1986] ili pogodnim odabirom internih struktura podataka za implementaciju SK redukcione mašine [Koopman, Lee, Siewiorek, 1992].

Ovaj odeljak se bavi ispitivanjem performansi simulatora SK redukcione mašine, u zavisnosti od nekoliko različitih optimizacija odnosno izmena implementacije simulatora.

5.4.3.1 Kombinator B' ili B*?

Prvobitni Turner-ov [1979] skup kombinatora pored kombinatora S, K, I, B, C, S'i C' sadrži kombinator B', čije je pravilo redukcije:

$$R[\![B'\ E\ F\ G\ H]\!] \quad = E\ F\ (G\ H)$$

a koji je uveden optimizacijom kombinatora B sledećeg oblika: **O**$[\![$ **B** $(E\ F)\ G]\!]$ -> **B'** $E\ F\ G$. Scheevel [1986] je tvrdio da kombinator B' ne samo da ne smanjuje veličinu kombinatorskog izraza, nego i narušava mogućnost daljih optimizacija (pod)izraza oblika **B** $E\ F$. Zbog toga je sugerisao uvodjenje novog kombinatora B*, čije je pravilo redukcije:

$$R[\![B^*\ E\ F\ G\ H]\!] \quad = E\ (F\ (G\ H))$$

a koji je uveden sledećom optimizacijom **O**$[\![$ **B** E (**B** $F\ G$) $]\!]$ -> **B*** $E\ F\ G$.

Mišljenja o tome koji kombinator postiže bolje rezultate u implementaciji SK apstraktne mašine su još uvek podeljena. Tako na primer [Peyton Jones, 1987] str. 272 tvrdi da eksperimenti pokazuju da B* poboljšava performanse apstraktne mašine u odnosu na B', dok Diller [1988] str. 102 tvrdi suprotno.

U ovom odeljku će biti prikazani rezultati merenja brzine izvršavanja testnih programa na simulatoru SK mašine sa B' i B* kombinatorima (Sl. 5.27). Za svaki testni program su mereni (apsolutni) broj redukcija i veličina kombinatorskog izraza. Pod brojem redukcija se podrazumeva broj primena svakog pravila R i R' iz odeljka 4.3.2, dok se izraz ne svede na oblik koji se više ne može redukovati. Veličina kombinatorskog izraza je data veličinom fajla u kome se nalazi smešten kombinatorski izraz, predstavljen s-izrazom. Eksperimenti su pokazali da je broj

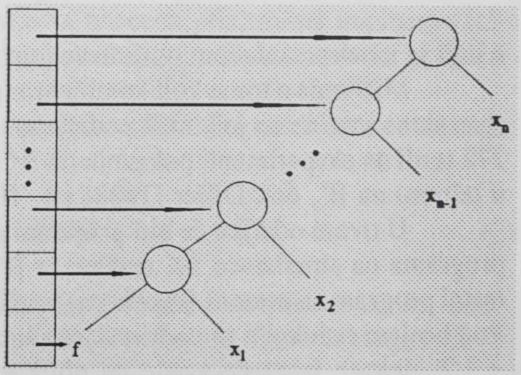
redukcija potrebnih za izvršavanje programa konzistentno proporcionalan sa vremenom potrebnim za izvršavanje, te se podaci dati za broj redukcija prikazani u tabeli odnose i na vreme izvršavanja testnih programa. Najbolji rezultati su označeni znakom ✓, dok su lošiji rezultati navedeni procentualnom razlikom u odnosu na najbolji rezultat.

Testni program	Redukcije		Velicina	
	B'	B*	B'	B*
Fib(21)	+0.001%	✓	+8.05%	✓
NFib(21)	+3.56%	✓	+13.54%	✓
Tak(12,9,3)	+21.4%	✓	+14.37%	✓
Queen(8)	+19.8%	✓	+24.41%	✓
nthPrime(13)	+8.33%	✓	+14.71%	✓
nth(59)	+5.99%	✓	+15.43%	✓
nthFirst(670)	+4.99%	✓	+8.67%	✓
Basic	+20.6%	✓	+27.47%	✓
Diag(22)	+12.9%	✓	+29.03%	✓

Sl. 5.27 Poredjenje SK mašine sa kombinatorima B' i B*

Iz tabele sa slike se vidi da je apstraktna SK mašina zasnovana na B* kombinatoru umesto na kombinatoru B' značajno efikasnija a i kombinatorski izraz sa B* je kraći nego što je to sa kombinatorom B'. Veće poboljšanje i u broju potrebnih redukcija i u veličini kombinatorskog izraza se može uočiti kod onih testnih programa koji sadrže veći broj definicija funkcija sa tri i više argumenata (**Basic**, **Queen** itd). Upravo zbog takvih funkcija su kombinatori C', B', S' i B* i uvedeni.

Kako su testni programi korišćeni za analizu raznovrsni i sveobuhvatni, dobijeni rezultati se mogu smatrati relevatnim i pouzdanim.



Sl. 5.28 Pomoćni stek

5.4.3.2 Pomoćni stek

Kombinatorski izraz se predstavlja u obliku grafa i sve redukcije izraza se realizuju kao redukcije grafa. U toku redukcije nekog podizraza (podgrafa), koristi se

pomoćni stek, kao što je prikazano na Sl. 5.28.

Kad se u procesu redukovanja dostigne čvor primene (tipa **ApplSE**), napreduje se po levom podgrafu sve dok postoje čvorovi primene, tj. dok se ne naidje na čvor koji predstavlja kombinator (na slici **f**). Istovremeno sa napredovanjem po levim potomcima grafa, svi čvorovi primene se pamte u steku, kako bi kasniji pristup argumentima kombinatora **f** bio brži. Na Sl. 5.29 se nalazi opšti oblik procedure **Eval** koja koristi pomoćni stek. Procedura **UnWind** koja se poziva unutar procedure **Eval**, postavlja pokazivače na čvorove primene u pomoćni stek.

U ovom odeljku će biti prikazano kako implementacija pomoćnog steka utiče na performanse SK redukcione mašine. Biće testirane tri karakteristične implementacije: lokalni pomoćni stek, globalni stek sa dva tipa elemenata i dva globalna steka. Sve tri realizacije stekova će biti statičke, zbog veće efikasnosti.

```
PROCEDURE Eval (VAR Graph: SExp): SExp;
VAR c: SExp;
BEGIN
  - inicijalizacija steka;
  UnWind(Graph);          (* "razvijanje" steka *)

  c := Stek[Vrh];      (* kombinator je na vrhu *)

  WHILE IsTypeSE(c, Combinator) AND
       (GetSymbolTableEntry(c)<>KeywordSE[cons])
          OneRed;           (* izvrsi redukciju *)
          c := Stek[Vrh];
  END;

  - restauracija steka

  RETURN Graph;
END Eval;
```

Sl. 5.29 Oblik procedure **Eval**

Implementacija lokalnog steka (procedure **Eval**) je najjednostavnija, jer ne zahteva ni posebnu inicijalizaciju, ni posebnu restauraciju steka. Na taj način, medjutim, dolazi do gubljenja memorijskog prostora, jer se pri svakom pozivu procedure **Eval** alocira novi lokalni stek (verovatno veći nego što je to potrebno za redukciju jednog podizraza).

Umesto lokalnih stekova je moguće uvesti jedan globalni, pri čemu svaki novi poziv procedure **Eval** koristi novi segment steka, neposredno iznad tekućeg. U tom slučaju se uvodi i dodatna promanljiva **Baza** koja sadrži osnovu (indeks prvog elementa) tekućeg segmenta. **Baza** prethodno korišćenog segmenta se takodje čuva u istom steku, da bi se prethodno stanje moglo restaurirati. Potreba da se u steku čuvaju dva tipa podataka, komplikuje implementaciju u višem programskom jeziku. Stek se realizuje kao niz promenljivih slogova, koji u zavisnosti od polja **Vrsta** čuvaju prirodni broj ili pokazivač na čvor (**SExp**). U realizaciji mašine je potrebno ispitivati polje **Vrsta**, da bi se utvrdilo koji podatak se nalazi na nekom mestu u steku. Na Sl. 5.30 se nalaze delovi procedure **Eval** za inicijalizaciju steka (postavljanje tekuće baze u stek i odredjivanje nove baze) i restauracije steka.

```
INC(Vrh);
Stek[Vrh].no := Baza;
Baza := Vrh;
....
Vrh := Baza;
Baza := Stek[Vrh].no;
DEC(Vrh);
```

Sl. 5.30 Inicijalizacija steka

Ideju sa globalnim stekom je moguće realizovati i sa dva globalna steka, odvojivši na taj način čuvanje baza segmenata steka od čuvanja pokazivača na

čvorove primene. I u ovom slučaju dolazi do trošenja memorijskih resursa više nego što bi to bilo inače potrebno, ali je implementacija stekova i njihovih elemenata jednostavnija.

Na Sl. 5.31 su prikazani rezultati merenja izvršavanja testnih programa sa sve tri implementacije steka. Meren je (i u tabeli prikazan) broj redukcija u sekundi - apsolutni broj redukcija se različitim implementacijama steka ne smanjuje. Broj redukcija u sekundi je proporcionalan vremenu utrošenom na izvršavanje programa, pa se podaci iz tabele mogu i tako tumačiti. Najbolji rezultati su kao i ranije dati oznakom ✓, dok su ostali dati procentualnom razlikom u odnosu na najbolja. Pri testiranju lokalni stekovi su imali po 10 elemenata, 1 globalni stek je imao 6000 elemenata, a kod testiranja implementacije sa dva globalna steka, stek sa pokazivačima je imao 6000, a test sa bazama 3000 elemenata. Merenja su izvršena na implementaciji simulatora SK mašine izvršenoj JPI Modula-2 prevodiocem, ver. 1.17.

Test	Lokalni stek	1 globalni stek	2 globalna steka
Fib(21)	+3.27%	+17.87%	✓
NFib(21)	+3.04%	+17.20%	✓
Tak(12,9,3)	+3.37%	+29.37%	✓
Queen(8)	+4.07%	+19.95%	✓
nthPrime(60)	-	+24.22%	✓
nth(600)	-	+26.66%	✓
nthFirst(740)	-	+14.75%	✓
Diag(22)	+3.46%	+16.59%	✓

Sl. 5.31 Merenja uticaja implementacije pomoćnog steka na performanse SK mašine

Iz tabele se vidi da najbolje rezultate postiže implementacija sa dva steka i da je konzistento bolja i od implementacije sa lokalnim stekovima (za oko 3-4%) i od implementacije sa jednim globalnim stekom (od 15 do 30%). Razlika u vremenu je veća kod onih testnih programa koji intenzivno opterećuju stekove i intenzivno pozivaju rekurzivne funkcije (kao što su **Tak** i programi za rukovanje beskonačnim strukturama podataka).

Rezultati merenja prikazanih u ovom odeljku zavise od implementacije prevodioca jezika u kome je implementiran simulator i njegovu efikasnost dodeljivanja prostora lokalnim promenljivima i njegovog oslobadjanja. Potreba za ispitivanjem oznake promenljivog sloga u globalnom steku i pristupanje jednom ili drugom polju u zavisnosti od oznake, degradira performanse simulatora. Moguće je da bi neki drugi prevodilac implementacionog jezika pokazao drugačije rezultate. Eksperimenti su takodje pokazali da je stek od 3000 elemenata dovoljan za sve testne programe. Odvajanje elemenata steka po tipovima u dva zasebna (globalna) steka

poboljšava performanse izvršavanja simulatora. Eksperimenti pokazuju da je u najvećem broju slučajeva u veličini stekova dovoljan odnos 3:1 u korist steka sa pokazivačima.

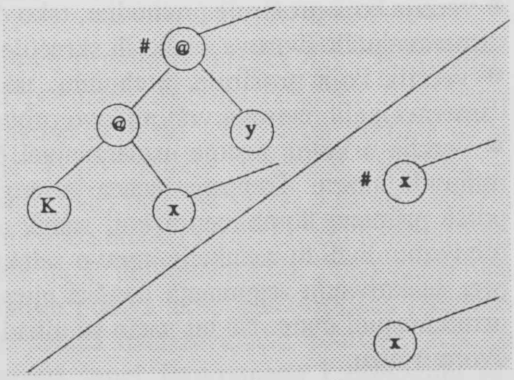
Pomoćni stek je moguće implementirati i na četvrti način, jednim stekom, ali tako da se nikad ne ispituje oznaka (odnosno tip) elementa niti da se oznaka elementa pamti u steku. Ovo je moguće ostvariti jer se u slučaju korektno implementiranog simulatora SK mašine u tekućem segmentu steka u bazi segmenta uvek nalazi broj (baza prethodnog segmenta) dok su sve ostalo pokazivači. Prilikom odabira ovakve realizacije steka, potrebno je da simulator prethodno bude dobro istestiran, jer bi obraćanje pokazivaču iz steka kao broju (ili obrnuto) dovelo do grešaka u radu simulatora koje se vrlo teško otkrivaju. Performanse ovakve realizacije steka su neznatno slabije od realizacije sa dva steka (manje od pola procenta u proseku), ali je zato iskorišćenje memorije znatno bolje.

Iz analize realizacije pomoćnog steka prikazane u ovom odljeku se može zaključiti da nepažljiva realizacija opšte strukture (a kojoj se često pristupa), može imati značajnog uticaja na performanse simulatora.

5.4.3.3 Usmeravajući čvorovi

Neka pravila za redukciju R (odeljak 4.3.2) narušavaju potpunu lenjost SK redukcione mašine, kod onih kombinatora koji od svojih argumenata selektuju jedan. Rezultat redukcije u tom slučaju je postojeći (a ne kreirani) čvor, koji može biti i deljen izmedju različitih delova grafa.

Na Sl. 5.32 je prikazana redukcija kombinatora **K** koji se redukuje tako da od svoja dva argumenta selektuje prvi. Na levoj slici je prikazan deo grafa pre redukcije, na kojoj čvor označen sa # predstavlja podizraz koji se trenutno redukuje i koga treba zameniti rezultatom redukcije. Na desnoj slici je prikazan rezultujući graf, na kojoj je čvor **x** kopiran na mesto čvora #. Ukoliko je čvor označen sa **x** deljen (tj. na njega pokazuju još neki pokazivači) čvor **x** će se redukovati dva puta, što potencijalno može da degradira performanse izvršavanja simulatora SK mašine, pogotovu ako je čvor **x** koren velikog podgrafa. U odeljku 4.3.2.1, na strani 104 se nalazi kompletan spisak ugradjenih kombinatora SK redukcione mašine za implementaciju LL-a, kod kojih se ovakav problem može javiti. Mašina koja ne pokušava da reši ovaj problem nije potpuno lenja.

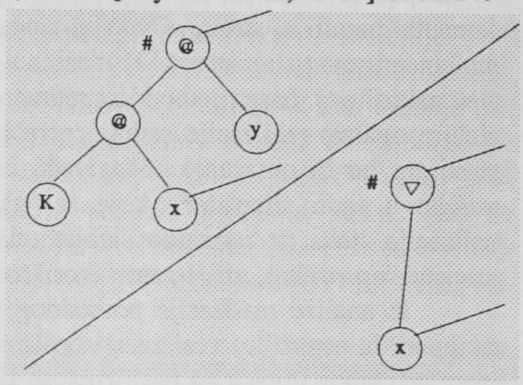


Sl. 5.32 Redukcija kombinatora K

Problem se može rešiti na više načina, od kojih je jedan mogući prikazan na Sl. 5.33. Umesto kopiranja čvora **x** u čvor označen sa #, taj čvor postaje tzv. usmeravajući čvor (engl. *indirection node*) koji pokazuje na "pravi" čvor **x**. Na taj

način je izbegnuto eventualno dupliranje redukcije čvora x, ali se javlja opasnost od stvaranja lanaca usmeravajućih čvorova, koji bi mogli da uspore rad simulatora SK mašine. Za implementaciju usmeravajućih čvorova je potrebno proširiti definiciju čvora prikazanu na Sl. 5.3, sledećim slučajem: **IndSE: SExp)**. Za nešto širu diskusiju ovog problema i moguća rešenja, videti [Peyton Jones, 1987] str. 214.

U ovom odeljku će biti prikazana analiza performansi izvršavanja simulatora u zavisnosti od različitih strategija rešenja navedenog problema. Neka od mogućih rešenja su sugerisana u [Peyton Jones, 1987] i ovde se, uz originalne, kratko opisuju. Implementirane su sledeće verzije simulatora SK redukcione mašine:



Sl. 5.33 Redukcija kombinatora K sa usmeravanjem

- "originalna" verzija, koja problem (potencijalno) dupliranih redukcija nikako ne rešava (te nije potpuno lenja);
- verzija koja uvodi usmeravajuće čvorove, uz potencijalnu opasnost da se stvore lanci takvih čvorova i tako uspore simulator. Pored toga, potreba da se na različitim mestima u implementaciji simulatora ispituje postojanje ovih čvorova, može da uspori simulator čak i ako u grafu ne postoji ni jedan usmeravajući čvor;
- verzija koja proširuje prethodnu, tako da skupljač otpadaka "prevezuje" sve lance usmeravajućih čvorova i tako ih skraćuje.
- verzija koja proširuje prethodnu, tako da se i u toku prolaženja kroz lance usmeravajućih čvorova vrši njihovo prevezivanje i skraćivanje.
- verzija u kojoj postoje usmeravajući čvorovi, ali u kojoj se sprečava kreiranje njihovih lanaca. Kreiranje lanaca je moguće sprečiti na osnovu razmatranja da će posle primene kombinatora koji samo selektuju neki od sopstvenih argumenata, obavezno doći do redukcije upravo selektovanog argumenta. Zbog toga je moguće pre selektovanja argumenta izvršiti njegovu redukciju, a tek posle toga postaviti usmeravajući čvor. Na taj način se nikad ne može formirati više od jednog takvog čvora u nizu.
- potpuno lenja verzija bez usmeravajućih čvorova, a na osnovu razmatranja iz prethodnog pasusa. Jedina razlika je u tome što se posle izvršene redukcije argumenta koji bi se selektovao ne formira usmeravajući čvor, nego se (redukovani) argument kopira. Razlika u odnosu na originalnu verziju je u tome što je potencijalna dupla redukcija selektovanog argumenta predupredjena njegovom prethodnom redukcijom.

U tabeli na Sl. 5.34 su prikazani rezultati merenja izvršavanja testnih programa na simulatoru SK mašine, sa svih šest opisanih načina izbegavanja mogućih duplih redukcija. U tabeli su prikazana vremena izvršavanja testnih programa. Razlike u vremenu izvršavanja, su u ovom slučaju, prouzrokovane ili

promenom (apsolutnog) broja redukcija potrebnih za izvršavanje programa, ili promenom broja redukcija u sekundi. Najbolja vremena su u tabeli označena oznakom ✓, dok su ostala vremena data procentualnom razlikom u odnosu na njih.

Testni programi	Orig.	Preusm.	Preusm. Skp.otp.	Preusm. Skp.otp Veze	Preusm. Skp.otp Izbeg.	Izbeg.
Fib(21)	✓	+9.30%	+8.70%	+12.08%	+11.55%	+1.34%
NFib(21)	✓	+8.99%	+8.30%	+11.62%	+10.98%	+1.70%
Tak(12,9,3)	✓	+8.94%	+8.53%	+11.97%	+10.04%	+0.62%
Queen(4)	✓	+11.13%	+10.35%	+14.46%	+12.52%	+1.76%
nthPrime(50)	+42.83%	+11.80%	+10.28%	+10.58%	+13.80%	✓
nth(600)	✓	+14.60%	+14.39%	+17.79%	+12.69%	+2.71%
nthFirst(740)	+8.03%	+11.66%	+11.39%	+14.06%	+14.85%	✓
Diag(10)	✓	+9.14%	+9.21%	+13.04%	+14.87%	+5.31%

```
Smanjenje u broju redukcija posle uvodjenja presumeravajućih cvorova:

Queen(4) = -1.97%
nthPrime(50) = -47.44%
nthFirst(740) = -9.48%
```

Sl. 5.34 Poredjenje različitih implementacija SK mašine

Implementacija SK mašine na osnovu pravila iz odeljka 4.3 ("originalna" implementacija) u većini testnih primera pokazuje najbolje rezultate. Uvodjenje usmeravajućih čvorova degradira performanse za 8 do 15%, zbog potrebe da se tokom redukovanja svaki čvor ispituje i po potrebi preskače, ukoliko je usmeravajući. Ako se skupljanje otpadaka proširi tako da se svi postojeći lanci u trenutku poziva skupljača otpadaka prevežu i maksimalno skrate (3. implementacija), performanse se nešto poboljšavaju u odnosu na 2. implementaciju. Poboljšanja nisu veća jer je broj dostupnih čvorova u trenutku poziva skupljača otpadaka relativno mali, te je relativno mali i broj aktivnih lanaca usmeravajućih čvorova. Ako se tokom "preskakanja" usmeravajućih čvorova lanci istovremeno i prevezuju (4. implementacija), performanse su najslabije. Ovaj metod uspeva da premosti više lanaca od prethodne implementacije, ali uz "previsoku cenu". Izbegavanje stvaranja lanaca prethodnim izračunavanjem vrednosti odgovarajućeg argumenta, a uz postojanje usmeravajućih čvorova (5. implementacija) pokazuje takodje vrlo slabe performanse. I najzad 6. implementacija (prethodno izračunavanje bez postojanja usmeravajućih čvorova) je po svojim performansama neznatno slabija od prve implementacije. U takvoj implementaciji je "opterećenje" steka (tj. broj rekurzivnih poziva funkcije **Eval**) veće, jer redukcija kombinatora **K**, **I**, **IF** itd. nije završena dok se ne završi redukcija jednog od njihovih argumenata. Iz istog razloga ova implementacija ne može da podrži izračunavanje nekih programa koje može da

podrži originalna (prva) implementacija - naročito one koje intenzivno opterećuju stek (**Diag**, na primer).

Samo dva testna programa (**nthPrime** i **nthFirst**) pokazuju bolje performanse pri izbegavanju potencijalno duplih redukcija, ali su ta dva programa važni predstavnici većih klasa. Ova dva programa imaju značajno manje redukcija u potpuno lenjoj implementaciji simulatora, što je i glavni razlog njihove veće efikasnosti - procenat umanjenja redukcija je srazmeran vremenskom ubrzanju. Primetimo takodje da se program **Queen** ne ponaša kao ova dva programa, jer je usporenje zbog specijalnog tretmana odredjenih kombinatora (ili uvodjenja usmeravajućih čvorova) ipak veće od uštede ostvarene smanjenjem broja redukcija.

Prikazani rezultati merenja nameću dva zaključka:

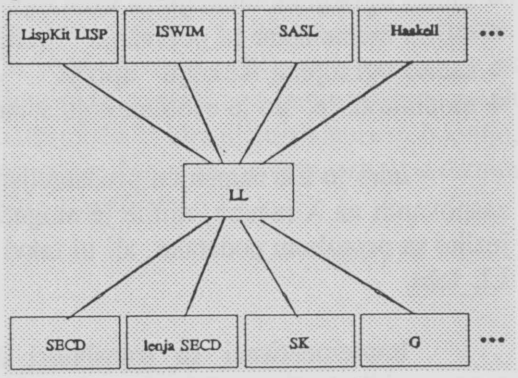
- uvodjenje usmeravajućih čvorova nije opravdano, jer značajno komplikuje realizaciju (simulatora) SK mašine što se odražava i u njenim performansama.
- implementacija potpuno lenje mašine je opravdana, jer neznatno degradira performanse već potpuno lenjih programa, a postiže mnogo bolje rezultate kod ostalih programa. Pored toga, relativno se jednostavno implementira. Jedini njen nedostatak je veće opterećenje pomoćnog steka i internog steka implementacionog jezika kojima se realizuju rekurzivni pozivi (procedure **Eval**).

Glava 6

Zaključak

U prethodnih pet poglavlja je opisan novi medjujezik LL za implementaciju čisto-funkcionalnih programskih jezika zasnovanih na λ računu. Kako su takvi jezici trenutno najbrojniji i najuticajniji, primenljivost LL-a je velika.

Uloga medjujezika LL je u implementaciji čisto-funkcionalnih programskih jezika slikovito predstavljena na Sl. 6.1.



Sl. 6.1 Mesto LL-a u implementacijama

LL je definisan tako da omogućava podjednako jednostavno predstavljanje jezika iz klasa F^S_+ i F^S_-; i F^T_+ i F^T_- i nezavisan je od načina implementacije. Po te dve karakteristike se LL razlikuje od svih analiziranih medjujezika. LL sadrži i konstrukcije koji omogućavaju predstavljanje jezika iz F^U_+ (_tuple) i F^X_+ (_case), a postojanjem anonimne funkcije u LL-u (_lambda) i ekvivalencije uvedene u odeljku 1.1 je omogućeno predstavljanje jezika iz F^C_+ i F^C_-.

Osnovna karakteristika LL-a je odvajanje semantike medjujezika od tehnika njegove implementacije. Osnovni stav ovakvog pristupa je da je LL delom notacija, te u skladu s tim:

- pozivi pojedinih ugradjenih funkcija LL-a nemaju definisanu vrednost za sve moguće vrednosti sopstvenih argumenata i
- LL jeste netipiziran medjujezik, čija implementacija medjutim nije obavezna da kontroliše ukladjenost tipova podataka tokom izvršavanja programa (te se LL može neformalno nazvati jezikom koji je niti tipiziran niti netipiziran).

Semantika "nedovoljno definisanih" ugradjenih funkcija se u potpunosti definiše pravilima prevodjenja poziva ugradjenih funkcija LL-a u mašinske komande neke apstraktne mašine i tako može biti tretirana kao tehnika implementacije. Semantika primene ugradjenih funkija LL-a se razlikuje jedino po tome kakvu vrednost ima ako je vrednost nekog od argumenata nedefinisana. Ideja o razdvajanju stroge semantike funkcionalnog programskog (medju)jezika od tehnika njegove implementacije je poznata i od ranije [Stoy, 1977] str. 173, [Traub, 1991] str. 28, ali nije bila često primenjivana.

Kontrola uskladjenosti tipova podataka takodje spada u domen tehnike implementacije i izvršava se ili implementacijom simulatora apstraktne mašine, ili tokom transliranja izvornog programa.

LL se od analiziranih medjujezika razlikuje i po sledećim karakteristikama:

- jednostavnije je semantike od većine analiziranih medjujezika.
- sadrži prošireni skup operatora za rukovanje listama (**_member**, **_len**, ...), koji su uvedeni zbog veće efikasnosti.
- sadrži ugradjene funkcije za kreiranje nizova i pristupanje elementima nizova.
- sadrži mehanizam za pozivanje "stranih" funkcija.
- sadrži ugradjenu funkciju **_apply**.
- modularan je, što je osobina koju samo neki medjujezici poseduju.

Iako je LL namenjen predstavljanju čisto-funkcionalnih programskih jezika zasnovanih na λ računu, njime je moguće predstaviti i jezike zasnovane na FP-u i jezike sa protokom podataka, ali bi takvi LL programi bili nečitkiji, a translacija u LL teža.

Implementirani su translatori iz nekoliko izvornih čisto-funkcionalnih programskih jezika u LL, kao i prevodioci LL-a u mašinske jezike nekoliko apstraktnih mašina. Programi su testirani simulatorima apstraktnih mašina. Sve implementacije su izvršene na mikro-računarima klase IBM PC, pod operativnim sistemom MS-DOS sa ograničenim memorijskim kapacitetima. Slične implementacije na mikro-računarima su, zbog ograničenih memorijskih resursa, retke.

Iako je LL nastao kao posledica detaljne analize postojećih čisto-funkcionalnih programskih jezika, medjujezika i njihovih tehnika implementacije, moguće je da će u bliskoj budućnosti pravac razvoja u ovoj oblasti usloviti i neke izmene u medjujeziku LL. U daljem tekstu navodimo nekoliko mogućih.

LL je definisan sa prevashodnom idejom o implementaciji čisto-funkcionalnih programskih jezika transliranjem u jezike nižeg nivoa, te je kao LL program odabran **_let** odnosno **_letrec** izraz. Ovi blokovi su medjutim relativno veliki i nepodesni za interaktivne implementacije funkcionalnih jezika. Ukoliko dalji

razvoj i korišćenje LL-a pokaže da postoji interes i za razvoj interaktivnih implementacija, program LL-a bi mogao da bude i **_lambda** izraz.

U LL se ne pravi razlika izmedju celih i realnih brojeva, što je u duhu deklarativnog programiranja i kompatibilno sa tretiranjem brojeva u mnogim funkcionalnim programskim jezicima (Scheme i Miranda, na primer). Ukoliko dalji razvoj funkcionalnog programiranja dobije drugačiji kurs i u LL-u bi se tretiranje celih i realnih brojeva moralo razdvojiti.

LL poseduje samo konstruktore podataka za gradjenje listi (**_cons** i **_nil**) i za gradjenje svih ostalih tipova podataka (**_tuple**). Posebni konstruktori za gradjenje liste postoje u LL-u, zbog toga što je jedini složeni tip podataka u netipiziranim jezicima upravo lista. Ukoliko se neki od postojećih netipiziranih jezika želi proširi novim složenim ugradjenim tipom podataka, tada se i LL mora proširiti odgovarajućim konstruktorima podataka. Kratka diskusija o tome zašto se tipovi podataka netipiziranih jezika ne mogu realizovati prevodjenjem u opšti konstruktor LL-a **_tuple**, se nalazi na strani 29.

Rezervisane reči LL-a se ne smeju biti identifikatori u izvornom programu koji se translira u LL. Kako se u implementacijama ne bi posebno vodilo računa o svim rezervisanim rečima LL-a, preporuka je da se u izvornom programu ne koriste identifikatori koji počinju sa _ (jer i rezervisane reči LL-a počinju istim znakom). Ako se poštuje ovo ograničenje, tada je jednostavnije realizovati translatore iz izvornih funkcionalnih jezika u LL, u kojima se često traži uvodjenje potpuno novih identifikatora: implementacija translatora tada može da bira identifikatore slobodnije, tako da počinju znakom _. Ovo je jedino ograničenje za programere u izvornim funkcionalnim programskim jezicima koji se transliraju u LL.

Literatura

1. Abelson, H., Sussman, G.J., Sussman, J. (1985). Structure and Interpretation of Computer Programs, MIT Press, Cambridge.

2. Ackermann, W. B. i Dennis, J. B. (1979). VAL - A Value Oriented Algorithmic Language Preliminary Reference Manual, Laboratory for Computer Science MIT/LCS/TR-218, MIT, Boston, Massachusets.

3. Aho, A.V., Sethi, R., Ullman, J.D. (1985). Compilers - Principles, Techiques, and Tools Addison Wesley, Reading, 1985.

4. Arvind i Gostelow, K.P. (1982). The U-interpreter, Computer 15, 2, 42-50.

5. Augustsson, L. (1984). A Compiler for lazy ML, U Proc. of the ACM Symposium on Lisp and Functional Programming, Austin, 218-227.

6. Augustsson, L. (1985). Compiling Pattern Matching. U Proc. of Conference on Functional Programming Languages and Computer Architecture, ed. Jouannaud, LNCS 210, Springer Verlag, 368-381.

7. Augustsson, L. (1987). Compiling Lazy Functional Languages, Part 2, doktorska teza, Chalmers University of Technology, Göteborg, Sweden.

8. Backus, J. (1978). Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. Communications of ACM, vol. 21, No. 8, 613-641.

9. Barendregt, H. P. (1984). The Lambda Calculus, its Syntax and Semantics, North Holland, Amsterdam.

10. Bird, R. i Wadler, P. (1988). Introduction to Functional Programming, Prentice Hall, New York.

11. Bloss, A. G. (1989). Path Analysis and the Optimization of Non-strict Functional Languages, doktorska teza, Research Report YALEU/DCS/RR-704, Yale University.

12. Boutel, B.E. (1988). Tui Language Manual, Tech. Rep. CSD-8-021. Victoria University of Wellington, New Zealand.

13. Brus, T., van Eekelen, M.C.J.D., van Leer, M. Plasmeijer, M. J., (1987). CLEAN - A Language for Functional Graph Rewriting. U *Proc. of the III International Conference on Functional Programming Languagea and Computer Architecture* (Portland, Oregon), *Springer Lecture Notes on Computer Science* 274, 364-384

14. Budimac, Z. (1990). LispKit Lisp - Description and Implementation, Ekspozitorni rad, Institut za matematiku, Univerzitet u Novom Sadu, Novi Sad, 41 str.

15. Budimac, Z. (1993). Do We Need Another Intermediate Functional Intermediate Language, Bulletin for Applied Mathematics, u štampi.

16. Budimac, Z. i Ivanović, M. (1989). New Data Type in Pascal. U *Proc. of the DECUS Europe Symposium* (The Hague, Holland), 193 - 199.

17. Budimac Z. i Ivanović M. (1990a). An Implementation of Functional Language using S-expressions. U *Proc. of XIV Information Technologies Conference "Sarajevo-Jahorina 90"* (Sarajevo), 111-1 - 111-8.

18. Budimac, Z. i Ivanović, M. (1990b). A Useful Pure Functional Language Interpreter. U *Proc. of XII International Symposium "Computer at the University"* (Cavtat), 3.17.1 - 3.17.6.

19. Budimac, Z. i Ivanović, M. (1991a). On some Modula-2 Influences to an Implementation of Functional Language. U *Proc. of II International Conference "Modula-2 and beyond"* (Loughborough Leicestershire, Great Britain), 322 - 331.

20. Budimac, Z. i Ivanović, M. (1991b). An Implementation of Function APPLY in a Compiler of Functional Language LispKit LISP. U *Proc. of XIII Intern. Conference on Information Technology Interfaces*, (V. Čerić, V. Dobrić, V. Lužar, R. Paul eds.), University Computing Centre, Zagreb, 125-130.

21. Budimac, Z. i Ivanović, M. (1991c). An Improved Implementation of the LispKit Lisp Language. U *Proc. of XVI International Summer School "Programming '91"* (Sofia, Bulgaria), 101-104.

22. Budimac, Z. i Ivanović, M. (1992b). Simulator lenje SECD mašine, verzija Nuam/lSECD. *Autorska agencija Jugoslavije - odeljenje za Srbiju, broj S-64/92*, 21. jul 1992.

23. Budimac, Z. i Ivanović, M. (1992c). Two Approaches to the "Mark and Sweep" Garbage Collection Algorithm. Prihvaćeno za prezentaciju na XVI simpozijumu o Informacionim tehnologijama "Sarajevo-Jahorina 1992", štampano u "Radovi Instituta za matematiku", no. 1, 1992, 33-40.

24. Budimac, Z. i Ivanović, M. (1993a). The Relationship between Lambda Calculus and Functional Programming, u pripremi.

25. Budimac, Z. i Ivanović, M. (1993b). Sintaksa i sematika čisto funkcionalnog programskog jezika LispKit LISP, verzija Null/SECD, u štampi.

26. Budimac, Z., Ivanović, M., Putnik, Z. i Tošić, D. (1991). LISP kroz primere, Institut za matematiku PMF, Univerzitet u Novom Sadu, ix + 254 strane.

27. Budimac, Z., Ivanović, M. i Živkov, S. (1992). Striktni kompajler za ISWIM, verzija Nui/SECD. *Autorska agencija Jugoslavije - odeljenje za Srbiju, broj S-63/92*, 21. jul 1992.

28. Budimac, Z., Ivanović, M., Živkov, S., Maćoš (1993). Kompajler za ISWIM, verzija Nui/SK, u pripremi.

29. Budimac, Z. i Maćoš, D. (1993). Realizacija stranih funkcija u lenjom funkcionalnom programskom jeziku. U Zb. radova 37. konferencije za ETAN (Beograd), 339-344.

30. Budimac, Z., Maćoš, D. i Ivanović, M. (1993). Another Bracket Abstraction Algorithm, VIII Seminar za primenjenu matematiku (Tivat), u štampi.

31. Budimac, Z. i Nikolajević, B. (1993). Implementation of SASL via Scheme. U Proc. of Fourth Theme ETAI Symposium with International Participation, (Ohrid, FYR Macedonia), u štampi.

32. Burstall, R.M., MacQueen, D.B. i Sanella, D.T. (1980). Hope: an Experimental Applicative Language, CSR-62-80, Department of Computer Science, University of Edinburgh.

33. Burton, F.W. (1982). A Linear Space Translation of Functional Programs to Turner Combinators, Inform. Proc. Lett. Vol. 14, No. 5, 201-204.

34. Church, A. (1941). The Calculi of Lambda Conversion, Princeton University Press, Princeton.

35. Clinger, W. i Rees, J. (1991). The Revised[4] Report on Algorithmic Language Scheme, ACM Lisp Pointers 3, 1-55.

36. Cousineau, G., Curien, P.-L. i Mauny, M. (1987). The Categorical Abstract Machine, Science of Computer Programming 8.

37. Curien, P.-L. (1986). Categorical Combinators, Sequential Algorithms and Functional Programming, London, Pitman.

38. Curry, H.B. i Feys, R. (1958). Combinatory Logic, Vol. 1, North Holland, Amsterdam.

39. Dobler, H. (1991). Top-Down Parsing in Coco-2, SIGPLAN Notices vol. 26, no. 3, 79-87.

40. Dobler, H. i Pirklbauer, K. (1990). Coco-2, A New Compiler Compiler, SIGPLAN Notices vol. 25, no. 5, 82-90.

41. Diller, A. (1988). Compiling Functional Languages, John Wiley and sons, Chichester.

42. Eekelen, van, M.C.J.D., Nöker E.G.J.M.H., Plasmeijer, M.J., Smetsers, J.E.W. (1990). Concurrent CLEAN, Version 0.6, Technical Report 90-20, University of Nijmegen.

43. Ekanadham, K. (1991). A Perspective on Id, u Parallel Functional Languages and Compilers (Szymanski, B. urednik), ACM Press, 197-253.

44. Fairbairn, J. (1985). Design and Implementation of a Simple Typed Language Based on the Lambda Calculus, doktorska teza, Univ. of Cambridge, TR, no. 75.

45. Fairbairn, J. i Wray, S. (1986). Code Generation Techniques for Functional Languages. U Proc. of the ACM Conference on Lisp and Functional Programming, Boston, 94-104.

46. Fairbairn, J. i Wray, S. (1987). TIM: A Simple Lazy Abstract Machine to Execute Supercombinators, U Proc. of Conference on Functional Programming and Computer Architecture.

47. Field, A. J. i Harrison, P. G. (1988). Functional Programming, Addison Wesley, New York.

48. Glaser, H., Hankin, C. i Till, D. (1984). Principles of Functional Programming, Prentice Hall, London.

49. Glauert, J. R., Kennaway, J. R., Sleep, M. R. i Somner, G. W. (1988). Final Specification of DACTL, Internal Report SYS-C88-11, University of East Anglia, Norwich.

50. Glauert, J. R., Leth, L. i Thomsen, B. (1991). A New Process Model for Functions. U *Proc. of teh SemaGraph '91 Symposium on the Semantics and pragmatics of Generalized Graph Rewriting*, (Nijmegen, Holland), Technical Report no. no. 91-25, University of Nijmegen, 167-182.

51. Gordon, M., Milner, R., Morris, L., Newey, M. i Wadsworth, C. (1978). A Metalanguage for interactive proof in LCF. U Proc. of the V Annual ACM Symposium on Principles of Programming Languages, ACM, 119-130.

52. Gurd, J. R., Kirkham, C. C. i Watson, I. (1985). The Manchester Prototype Dataflow Computer, *Communications of ACM* Vol. 28. No. 1, 34-52.

53. Hancock, P. (1987a). Polymorphic Type Checking, Poglavlje u Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall, London, 139-162.

54. Hancock, P. (1987b). A Type Checker, Poglavlje u Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall, London, 163-182.

55. Henderson, P. (1980). Functional Programming - Application and Implementation, Prentice Hall, New York.

56. Henderson, P., Jones, G.A. i Jones, S.B. (1983). The LispKit Manual, Technical Monograph PRG-32 (Vol. 1 i 2), Oxford University Computing Laboratory.

57. Henderson, P., Morris, J.H. (1976). A Lazy Evaluator, U Proceedings of the 3. ACM Symposium on Principles of Programming Languages, Atlanta, 95-103.

58. Hudak, P. (1984). ALFL Reference Manual and Programmer's Guide, Research Report YALEU/DCS/RR-322, Yale University.

59. Hudak, P. (1986). Arrays, Non-Determinism, Side-Effects, and Parallelism: A Functional Perspective. Proc. of LANL/MCC Graph Reduction Workshop, preprint.

60. Hudak, P. (1989). Conception, Evolution, and Application of Functional Programming Languages, *ACM Computing Surveys* Vol. 21, No. 3, 359-411.

61. Hudak, P. (1991). Para-Functional Programming in Haskell. U Parallel Functional Languages and Compilers (ed. Szymanski, B.K.), ACM Press, New York.

62. Hudak, P., Fasel, J.H. (1992). A Gentle Introduction to Haskell, SIGPLAN Notices Vol. 27, No. 5, T-1 - T-53.

63. Hudak, P., Peyton Jones, S. L. i Wadler, P. (urednici) (1991). Report on the Programming Language Haskell - a Non-strict, Purely Functional Language, Version 1.0, Internal Report of the Yale University.

64. Hudak, P., Peyton Jones, S. L. i Wadler, P. (urednici) (1992). Report on the Programming Language Haskell - a Non-strict, Purely Functional Language, Version 1.2, SIGPLAN Notices Vol. 27, No. 5, R-1 - R-162.

65. Hudak, P. i Sundaresh, R.S. (1988). On the Expressiveness of Purely Functional I/O Systems, Res. Rep. YALEU/DCS/RR-665.

66. Hughes, J. (1982). Graph Reduction with Super-Combinators, Technical Monograph PRG-28, Oxford University Computing Laboratory, Oxford.

67. Hughes, J. (1984). The Design and Implementation of Programming Languages, doktorska teza, PRG-40, Programming Research Group, Oxford.

68. Ivanović, M. i Budimac, Z. (1989). Usage of S-Expression in Pascal. U *Proc. of XI International Symposium "Computer at the University"* (Cavtat), 3.18.1 - 3.18.6.

69. Ivanović, M. i Budimac, Z. (1990). Involving Coroutines in Interaction between Functional and Conventional Language, *ACM SIGPLAN Notices* Vol. 25, No. 11, 65 - 74.

70. Ivanović, M. i Budimac, Z. (1993). An Implementation of ISWIM-like Language via Scheme, *ACM SIGPLAN Notices* Vol. 27, No. 4, 29-38.

71. Ivanović, M., Budimac, Z. i Nikolajević, B. (1993). Another Algorithm for Pattern Matching Compilation, Bulletin for Applied Mathematics, u štampi.

72. Ivanović, M., Budimac, Z. i Putnik, Z. (1991). LispKit LISP sistem, verzija Null/SECD. *Autorska agencija Jugoslavije - odeljenje za Srbiju, broj S-63/91*, 23. jul 1991.

73. Ivanović, M., Stojković, V., Budimac, Z. i Putnik, Z. (1985). Implementacija mini-BASIC interpretatora na LispKit LISP jeziku. U Zb. rad. VII medjunarodnog simpozijuma "Kompjuter na sveučilištu" (Cavtat), 102-1 - 102-12.

74. Johnson, S.C. (1975). YACC - Yet Another Compiler Compiler, Tech. Rep. No. 32, Bell Laboratories.

75. Johnsson, T. (1984). Efficient Compilation of Lazy Evaluation, U Proc. of the ACM Conference on Compiler Construction, Montreal, 122-132.

76. Johnsson, T. (1987). Compiling Lazy Functional Languages, doktorska teza, Chalmers University of Technology, Göteborg, Sweden.

77. Joy, M. S. (1989). The Translation of High-Level Functional Languages to FLIC, Research Report 142, Department of Computer Science, University of Warwick.

78. Kelsey, R. A. (1989). Compilation by Program Transformation, doktorska teza, Research Report of Yale University YALEU/CSD/RR-702.

79. Keller, R.M. (1982). FEL Programmers Guide, AMPS TR 7, University of Utah.

80. Kennaway, J.R. (1984). The Complexity of a Translation of λ-Calculus to Combinators, Internal Report CSA/13/1984, School of Information Systems, University of East Anglia.

81. Kennaway, J. R., Klop, J. W., Sleep, M. R. i de Vries, F. J. (1991). Event Structures and Orthogonal Term Graph Rewriting. U *Proc. of the SemaGraph '91 Symposium on the Semantics and pragmatics of Generalized Graph Rewriting*, (Nijmegen, Holland), Techical Report no. no. 91-25, University of Nijmegen, 313-337.

82. King, I. (1991). A Technique for Determining the Efficiency of Abstract Machines. U Proc. of SemaGraph '91 Semantics and Pragmatics of Generalized Graph Rewriting (Nijemegen, Holland), objavljeno kao Tech. Report no. 91-25, University of Nijmegen, 395-419.

83. Koopman, P. J., Lee, P. i Siewiorek, D. P. (1992). Cache Behaviour of Combinator Graph Reduction, *ACM Transactions on Programming Languages and Systems* Vol. 14 No. 2, 265-297.

84. Koopman, P. J., Smetsers, J.E.W., van Eekelen, Plasmeier, M.J. (1991). Efficient Graph Rewriting using the Annotated Functional Strategy. U *Proc. of the SemaGraph '91 Symposium on the Semantics and pragmatics of Generalized Graph Rewriting*, (Nijmegen, Holland), Technical Report no. 91-25, University of Nijmegen, 225-248.

85. Kranz, D. A. (1988). ORBIT: An Optimizing Compiler for Scheme, doktorska teza, Research Report YALEU/DCS/RR-632, Yale University.

86. Lambek, J. (1980). From lambda Calculus to Cartesian Closed Categories. U "To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism" (Seldin, J.P., Hindley, J.R., eds.), London, Academic Press, 375-402.

87. Landin, P.J. (1964). The Mechanical Evaluation of Expressions, Computer Journal, Vol. 6, No. 4, 308-320.

88. Landin, P.J. (1966). The Next 700 Programming Languages, Communications of ACM Vol. 9, No. 3, 157-166.

89. Lindsey, C.H. (1993). A History of ALGOL 68. U Proc of II History of Programming Languages Conference (Cambridge, Massachusetts), SIGPLAN Notices, vol. 28, no. 3, 97-132.

90. Lins, R.D. i Thompson, S.J. (1990). Implementing SASL using Categorical Multi-Combinators, Soft. Pract. Exper., Vol. 20, No. 11, 1137-1165.

91. Lord, A. M. (1987). A Miranda to FLIC Translator (A Study of Functional Programming), Research Report No. 9, University of Warwick.

92. MacLennan, B. (1990). Functional Programming, Theory and Practice, Addison Wesley.

93. Maćoš, D. i Budimac, Z. (1992). Some Experiences with SK Reduction Machine - User's Perspective, U *Proc. of abstracts of VI Conference on Logic and Computer Science "LIRA '92"* (Novi Sad), 17.

94. Maćoš, D. i Budimac, Z. (1993). Simulator SK mašine, verzija Nuam/SK, u pripremi.

95. Maranget, L. (1992). Compiling Lazy Pattern Matching. U Proc. of ACM Conference on LISP and Functional Programming, 21-31.

96. McCarthy, J. (1960). Recursive Fnctions of Symbolic Expressions and their Computation by Machine Part I, Commun. ACM 3, 4, 184-195.

97. McGraw, J. R. (1982). The VAL Language: Description and Analysis, TOPLAS, 4, 1, 44-82.

98. McGraw, J., Allan, S., Glauert, J. i Dobes, I. (1983). SISAL: Streams and Iteration in a Single Assignment Language, Language Reference Manual, Tech. Rep. M-146, Lawrence Livermore National Laboratory.

99. Milner, R. (1978). A Theory of Type Polymorphism in Programming, Journal of Computer and System Science, vol 17, 348-375.

100. Milner, R. (1984). A Proposal for Standard ML. U Proc. of ACM Conference on LISP and Functional Programming, ACM, 184-197.

101. Mitić, N. (1989). Implementacija FP jezika u LispKit LISP jeziku, Magistarski rad, Matematički fakultet, Beograd.

102. Nikolajević, B. i Budimac, Z. (1992). On Compilation of Pattern Matching in SASL Language. U *Proc. of VI Conference on Logic and Computer Science "LIRA '92"*, (Novi Sad), 85-92.

103. Nikhil, R.S. (1988). Id (version 88.1) Reference Manual, Computation Structures Group Memo 284, MIT Laboratory for Computer Science, Cambridge, Massachusets.

104. Noshita, K. (1985). Translation of Turner Combinators in O(*n*log*n*) Space, Inform. Proc. Lett. 20(1985), 71-74.

105. Nöcker, E.G.J.M.H., Smetsers, J.E.W., van Eekelen, M.C.J.D., Plasmeijer, M.J. (1991). Concurrent CLEAN. U *Proc. of the Conference on Parallel Architectures and Languages* (Eindhoven Holland), *Springer Verlag Lecture Notes on Computer Science* 505.

106. Peterson, J. i Jones, M. (1993). Implementing Type Classes. U Proc. of ACM Conference on Programming Languages and Design (Albuquerque, U.S.A.), 227-236.

107. Peyton Jones, S. L. (1987). The Implementation of Functional Programming Languages, Prentice Hall, New York.

108. Peyton Jones, S. L. (1988). FLIC - A Functional Language Intermediate Code, *ACM SIGPLAN Notices* Vol. 23, No. 8, 30-48; objavljeno i kao: Internal Note 2048, Department of Computer Science, University College London, 1987.

109. Peyton Jones, S. L. i Joy, M. S. (1990). FLIC - A Functional Language Intermediate Code, Last Revision, Research Report 148, Department of Computer Science, University of Warwick.

110. Putnik, Z., Budimac, Z. i Ivanović, M. (1992a). Some Improvements of a LispKit LISP Syntax Analyzer. U *Proc. of I International Symposium DECSYM '92* (Side-Antalia, Turkey), 243-252.

111. Putnik, Z., Budimac, Z. i Ivanović, M. (1992b). Dependency Analysis in an Untyped Functional Language: an Exercise in Functional Programming. U *Proc. of VI Conference on Logic and Computer Science "LIRA '92"*, (Novi Sad), 99-106.

112. Putnik, Z., Ivanović, M. i Budimac, Z. (1993). Primena analize zavisnosti u implementaciji funkcionalnih programskih jezika. U Zb. radova 37. konferencije za ETAN (Beograd), 351-356.

113. Rechenberg, P i Mössenböck, H. (1989). A Compiler genretaor for Microcomputers, Prentice Hall, London.

114. Rees, J. A. i Adams, N. I. (1982). T: a Dialect of LISP or, Lambda: the Ultimate Software Tool. U *Proc. of ACM Conference on LISP and Functional Programming*, 114-122.

115. Robinson, J.A. (1965). A Machine-Oriented Logic based on the Resolution Principle, Jornal of the ACM, vol. 12, no. 1, 23-41.

116. Scheevel, M. (1986). Norma: a Graph Reduction Processor. U Proc. of the ACM Conference on LISP and Functional Programming (Cambridge, Massachusets), 212-290.

117. Schönfinkel, M. (1924). Über die bausteine der mathematischen logik, Mathematishe Annalen 92, 305.

118. Scott, D. i Strachey, C. (1971). Towards a Mathematical Semantics for Computer Languages, Tech. Monograph PRG-6, Oxford University Computing Laboratory.

119. Skedzielewski, S. K. (1991). Sisal. U Parallel Functional Languages and Compilers (Szymanski, B., urednik, ACM Press.

120. Smith, J.D. (1988). An Introduction to Scheme, Prentice Hall, Englewood Cliffs.

121. Sokolowski, S. (1991). Applicative High Order Programming, Chapman & Hall Computing.

122. Stojković, V., Stojmenović, I., Jerinić, Lj., Mirčevski, J. i Kulaš, M. (1983). Programska implementacija simulatora SECD mašine. U Zb. radova 27. Jugoslovenske konferencije ETAN-a (Struga), IV, 337-344.

123. Stojković, V. i dr. (1984). LispKit LISP jezik, verzija ARL, Bilten SAP Vojvodine za nauku i informatiku.

124. Stoy, J.E. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming language Theory, MIT Press, Cambridge, 1977.

125. Stoye, W.R. (1983). The SKIM microprogrammer's guide, Tech. Rep. 31, University of Cambridge.

126. Stoye, W.R. (1985). The Implementation of Functional Languages using Custom Hardware, doktorska teza, University of Cambridge.

127. Sussman, G.J. i Steele, G.L. (1975). Scheme: An Interpreter for Extended Lambda Calculus, MIT Artificial Intelligence Memo, 349, 12.

128. Tarditti, D., Lee, P. i Acharya, A. (1992). No Assembly Required: Compiling Standard ML to C, Lett. on. Prog. Lang. Syst. vol. 1, no. 2, 161-177.

129. Traub, K. R. (1991). Implementation of Non-Strict Functional Programming Languages, MIT Press.

130. Tremblay, J.P. i Sorenson, P.G. (1985) The Theory and Practice of Compiler Writing, Prentice Hall, London.

131. Turner, D. A. (1976). SASL Language Manual, Tech. Rep. University of St. Andrews.

132. Turner, D. A. (1979). A New Implementation Technique for Applicative Languages, *Software - Practice and Experiences* Vol. 9, 31-49.

133. Turner, D. A. (1981a). The Semantic Elegance of Applicative Languages. U Proc. of the 1981 Conference on Functional Programming Languages and Computer Architecture, ACM, 85-92.

134. Turner, D. A. (1981b). Aspects of the Implementation of Programming Languages, doktorska teza, University of Oxford.

135. Turner, D. A. (1982). Recursion Equations as a Programming Language. U *Functional Programming and its Applications* (urednici Darlington, P., Henderson, P. Turner, D.A.), Cambridge University Press.

136. Turner, D. A. (1985). Miranda: A Non-strict Functional Language with Polymorphic Types, Functional Programming Languages and Computer Architecture, Springer-Verlag LCNS 201, 1-16.

137. Turner, D. A. (1990). SASL Language Manual, revised by M.S. Joy, Functional Language Implementation Project, Document 14, University of Warwick, University of Birmingham.

138. Wadge, W i Ashcroft (1985). Lucid, the dataflow Programming Language, Academic Press, London.

139. Wadler, P. (1987a). Efficient Compilation of Pattern-Matching, Poglavlje u Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall, London, 78-103.

140. Wadler, P. (1987b). List Comprehension, Poglavlje u Peyton Jones, S.L. *The Implementation of Functional Programming Languages*, Prentice Hall, London, 127-138.

141. Wadler, P. i Miller, Q. (1988). An Introduction to Orwell, Tech. Rep. of Programming Research Group, Oxford University.

142. Wadsworth, C. P. (1971). Semantics and Pragmatics of the Lambda-Calculus, doktorska teza, University of Oxford.

143. Williams, J. (1982). Notes on the FP Style of Functional Programming (urednici Darlington, P., Henderson, P. Turner, D.A.), Cambridge University Press.

144. Winston, P.H. i Horn, B.K.P. (1981). LISP, Addison Wesley, Reading.

145. Young, J. H. (1989). The Theory and Practice of Semantic Program Analysis for Higher-Order Functional Programming Languages, doktorska teza, Research Report YALEU/DCS/RR-669, Yale University.

146. Živkov, S., Budimac, Z. i Ivanović, M. (1993). Korisničko uputstvo za čisto funkcionalni programski jezik ISWIM, u pripremi.

Dodatak

Dodatak sadrži kompletnu definiciju translatora izvornog čisto-funkcionalnog programskog jezika ISWIM u medjujezik LL. Definicija translatora je data u COCOL-u, jeziku kompajlera kompajlera COCO-2 [Dobler, 1991; Dobler, Pirklbauer, 1991]. U definiciji translatora se koriste sledeće procedure iz implementacije LL-a, koje ukratko opisujemo.

QuoteSE - pretvara string sa zapisom s-izraza u s-izraz;
CreateSE - gradi "prazan" s-izraz;
QsSE - gradi simbolički s-izraz (atom);
QrSE - gradi numerički s-izraz (atom);
ConsSE - gradi tačkasti s-izraz;
TailSE - ima vrednost "repa" tačkastog s-izraza;
HeadSE - ima vrednost "glave" tačkastog s-izraza;
DispSE - prikazuje s-izraz na ekranu;
NullSE - oznaka za "prazan" s-izraz;
EqSE - ispituje jednakost dva s-izraza;
AppSE - spaja dva s-izraza u jedan, konkatenacijom;

```
COMPILER ISWIM

LEX << FROM SExps          IMPORT QuoteSE, SExp;
       FROM Lib            IMPORT AddAddr;
       FROM InOut          IMPORT WriteString;
    >>

SEM << FROM SExps          IMPORT SExp, CreateSE, QsSE, QrSE, ConsSE, ReTaSE,
                                  TailSE, HeadSE, DispSE, NullSE, EqSE;
       FROM LLoperEgr      IMPORT AppSE;
       FROM ISWIMLex       IMPORT SpixToString;
       FROM InOut          IMPORT WriteString, WriteLn;
       FROM ISWIMGlobal    IMPORT LispSource;

       PROCEDURE One(e: SExp): SExp;
       BEGIN
          RETURN ConsSE(e, CreateSE())
       END One;
       PROCEDURE Two(e1, e2: SExp): SExp;
       BEGIN
          RETURN ConsSE(e1, ConsSE(e2, CreateSE()))
```

```
      END Two;
      PROCEDURE Three(e1, e2, e3: SExp): SExp;
      BEGIN
         RETURN ConsSE(e1, ConsSE(e2, ConsSE(e3, CreateSE())))
      END Three;
   >>

IGNORE CASE

CHARACTER SETS
  letter        = 'A'..'Z'+'a'..'z'.
  digit         = '0'..'9'.

COMMENTS
  FROM '/*' TO '*/' NESTED.
  FROM '//' TO EOL.

KEYWORDS
  'WHERE'.      'WHEREREC'.     'AND'.
  'TRUE'.       'FALSE'.        'FROM'.         'MOD'.
  'DIV'.        'LET'.          'IN'.
  'REC'.        'HD'.           'TL'.           'ATOM'.
  'NIL'.

TOKENS
  '{'.          '}'.            '='.            '"'.
  '('.          ')'.            ','.            '+'.
  '-'.          '*'.            '/'.            '<'.
  '<='.         '>'.            '>='.           '#'.
  '&'.          '|'.            '~='.           '@'.
  '~'.          '.'.            ']'.            '['.
  '++'.         ':'.            ';'.            '->'.
  '!'.          '_'.            '\'.

TOKEN CLASSES

  Id << VAR spix: SExp >> =
        letter {letter | digit}
        LEXLOCAL << VAR token: ARRAY [1..30] OF CHAR;
                        p: CharPtr;
                        i: CARDINAL;
                 >>
        LEX << p := tokenStart;
               i := 0;
               REPEAT
                  INC(i);
                  token[i] := CAP(p^);
                  p := AddAddr(p,1);
               UNTIL p = AddAddr(tokenEnd,1);
               INC(i);
               token[i] := CHR(0);
               spix := QuoteSE(token);
            >>.

  QuotedId << VAR spix: SExp >> =
        '"' letter {letter | digit} '"'
        LEXLOCAL << VAR token: ARRAY [1..30] OF CHAR;
                        p: CharPtr;
                        i: CARDINAL;
                 >>
        LEX << p := tokenStart;
               p := AddAddr(p,1);
               i := 0;
               REPEAT
                  INC(i);
                  token[i] := CAP(p^);
                  p := AddAddr(p,1);
               UNTIL p = AddAddr(tokenEnd,1);
```

```
            token[i] := CHR(0);
            spix := QuoteSE(token);
          >>.

  Number << VAR val: SExp >> =
        digit {digit} ['.' digit {digit}] ['E' ['-'] digit {digit}]
        LEXLOCAL << VAR token: ARRAY [1..30] OF CHAR;
                        p: CharPtr;
                        i: CARDINAL;
                 >>
        LEX << p := tokenStart;
               i := 0;
               REPEAT
                  INC(i);
                  token[i] := p^;
                  p := AddAddr(p,1);
               UNTIL p = AddAddr(tokenEnd,1);
               INC(i);
               token[i] := CHR(0);
               val := QuoteSE(token);
            >>.

NONTERMINALS

  ISWIM.                          Expression <<VAR e:SExp>>.
  WhereBlock <<VAR e:SExp>>.      LetBlock <<VAR e:SExp>>.
  Definition <<VAR e:SExp>>.      SimpleExp <<VAR e:SExp>>.
  Term <<VAR e:SExp>>.            Factor <<VAR e:SExp>>.
  Block <<VAR e:SExp>>.           RelOp <<VAR e:SExp; VAR neg: BOOLEAN>>.
  AddOp <<VAR e:SExp>>.           MulOp <<VAR e:SExp>>.
  WhereRec <<VAR e:SExp>>.        Conditional <<VAR e:SExp>>.
  Anonimous <<VAR e:SExp>>.       ExpList <<VAR e:SExp>>.
  IdList <<VAR e:SExp>>.          Sequence <<VAR e:SExp>>.
  Constant <<VAR e:SExp>>.        SeqExp <<VAR e:SExp>>.
  LetRec <<VAR e:SExp>>.          UnaryOp <<VAR e:SExp>>.
  AndExp <<VAR e:SExp>>.          OrExp <<VAR e:SExp>>.
  AndOp <<VAR e:SExp>>.           OrOp <<VAR e:SExp>>.
  SeqOp <<VAR e:SExp>>.           ForeignFunctionId<<VAR e:SExp>>.

RULES
  ISWIM
              = Block <<LispSource>>.

  Block <<VAR e: SExp>>
              = '{' ( WhereBlock<<e>> | LetBlock<<e>> ) '}'.

  WhereBlock <<VAR e: SExp>>
              = LOCAL << VAR exp, ltr, df, Defs: SExp; >>
                Expression<<exp>>
                WhereRec<<ltr>>
                Definition<<df>>
                SEM << Defs := One(df); >>
                { 'AND' Definition<<df>>
                   SEM << Defs := AppSE(Defs, One(df)); >>
                }
                SEM << e := ConsSE(ltr, ConsSE(exp, Defs));>>.

  LetBlock <<VAR e: SExp>>
              = LOCAL <<VAR Defs, ltr, df, exp: SExp;>>
                LetRec<<ltr>>
                Definition<<df>>
                SEM << Defs := One(df); >>
                { 'AND' Definition<<df>>
                   SEM << Defs := AppSE(Defs, One(df)); >>
                }
                ';' Expression<<exp>>
                SEM << e := ConsSE(ltr, ConsSE(exp, Defs));>>.
```

```
Definition <<VAR e: SExp>>
              = LOCAL << VAR Stack, list, idnt, exp, idnt1: SExp;>>
                Id<<idnt>>
                SEM << Stack := One(One(idnt)); >>
                { IdList<<list>>
                  SEM << Stack := ConsSE(Two(QsSE("_lambda"), list), Stack); >>
                }
                (  '=' Expression<<exp>>
                    SEM << e := exp;
                           WHILE NOT EqSE(Stack, NullSE) DO
                             IF NOT EqSE(TailSE(Stack), NullSE) THEN
                                e := One(e)
                             END;
                             e := AppSE(HeadSE(Stack), e);
                             Stack := TailSE(Stack);
                           END;
                        >>
                |
                   'FROM' Id<<idnt1>>
                    SEM << e := Three(idnt, QsSE("_from"), idnt1); >>
                ).

WhereRec <<VAR e: SExp>>
              = 'WHERE'     SEM << e := QsSE("_let"); >>
                 [ 'REC'    SEM << e := QsSE("_letrec"); >> ].

LetRec <<VAR e: SExp>>
              = 'LET'       SEM << e := QsSE("_let"); >>
                 [  'REC'   SEM << e := QsSE("_letrec"); >> ].

IdList <<VAR e: SExp>>
              =  LOCAL <<VAR idnt: SExp; >>
                 '(' SEM << e := One(CreateSE()); >>
                     [ Id<<idnt>>
                       SEM << e := One(idnt); >>
                       { ',' Id<<idnt>>
                          SEM << e:= AppSE(e, One(idnt)); >>
                       }
                     ]
                 ')'.

Expression <<VAR e: SExp>>
              = LOCAL << VAR Stack, cnd: SExp; >>
                SEM << Stack := CreateSE(); >>
                SeqExp<<e>>
                SEM << Stack := ConsSE(e, Stack); >>
                { SeqOp<<e>>
                  SEM << Stack := ConsSE(e, Stack); >>
                  SeqExp<<e>>
                  SEM << Stack := ConsSE(e, Stack); >>
                }
                SEM << e := HeadSE(Stack);
                       Stack := TailSE(Stack);
                       WHILE NOT EqSE(Stack, NullSE) DO
                         e := Three(HeadSE(Stack), HeadSE(TailSE(Stack)), e);
                         Stack := TailSE(TailSE(Stack));
                       END;
                    >>
                [ Conditional<<cnd>>
                  SEM <<e := ConsSE(QsSE("_if"), ConsSE(e, cnd)); >>
                ].

Conditional <<VAR e: SExp>>
              = LOCAL <<VAR exp: SExp;>>
                '->' Expression<<e>> ';' Expression<<exp>>
                SEM << e := Two(e, exp); >>.

 SeqExp <<VAR e: SExp>>
```

```
            = LOCAL <<VAR op, exp: SExp;>>
              OrExp<<e>>
              { OrOp<<op>> OrExp<<exp>>
                SEM << e := Three(op, e, exp); >>
              }.

OrExp <<VAR e: SExp>>
            =  LOCAL <<VAR op, exp: SExp;>>
               AndExp<<e>>
               { AndOp<<op>> AndExp<<exp>>
                 SEM << e := Three(op, e, exp); >>
               }.

AndExp <<VAR e: SExp>>
            = LOCAL <<VAR exp, op: SExp;
                          neg: BOOLEAN;
                    >>
              SimpleExp<<e>>
              [ RelOp<<op, neg>> SimpleExp<<exp>>
                SEM << e := Three(op, e, exp);
                       IF neg THEN
                          e := Two(QsSE("_not"), e);
                       END;
                    >>
              ].

SimpleExp <<VAR e: SExp>>
            = LOCAL <<VAR op, exp: SExp;>>
              Term<<e>>
              { AddOp<<op>> Term<<exp>>
                SEM << e := Three(op, e, exp); >>
              }.

Term <<VAR e: SExp>>
            = LOCAL << VAR mns, op, exp: SExp;
                       WasMinus: BOOLEAN;
                    >>
              SEM << WasMinus := FALSE; >>
              [ '-'
                SEM << WasMinus := TRUE;
                       mns := ConsSE(QsSE("_sub"),
                                  One(Two(QsSE("_quote"), QrSE(0.0))));
                    >>
              ]
              Factor<<e>>
              SEM << IF WasMinus THEN
                        e := AppSE(mns, One(e));
                     END;
                    >>
              { MulOp<<op>> Factor<<exp>>
                SEM << e := Three(op, e, exp); >>
              }.

  Factor <<VAR e: SExp>>
            = LOCAL <<VAR list, op, exp: SExp;>>
              ( '(' (Expression<<e>> ¦ Anonimous<<e>>) ')'
                ¦ Block<<e>>
                ¦ Id<<e>>
              )
              { ExpList<<list>>
                SEM << e := ConsSE(e, list); >>
              }
              ¦
              ForeignFunctionId<<e>>
              ExpList<<list>>
                SEM << e := AppSE(e, list); >>
              ¦
              '[' Sequence<<list>> ']'
```

```
                    SEM << e := ConsSE(QsSE("LIST"), list); >>
                  |
                  UnaryOp<<op>> Factor<<exp>>
                  SEM <<e := Two(op, exp);>>
                  |
                  Constant<<e>>.

Constant <<VAR e: SExp>>
              = LOCAL <<VAR idnt, Val : SExp; >>
                Number<<Val>>
                SEM << e := Two(QsSE("_quote"), Val); >>
                |
                'NIL'
                SEM << e := QsSE("_nil"); >>
                |
                QuotedId<<idnt>>
                SEM << e :=  Two(QsSE("_quote"), idnt); >>
                |
                'TRUE'
                SEM << e := Two(QsSE("_quote"),QsSE("_true")); >>
                |
                'FALSE'
                SEM << e := Two(QsSE("_quote"),QsSE("_false")); >>.

Anonimous <<VAR e: SExp>>
              = LOCAL << VAR list, idnt: SExp;>>
                '\' Id<<idnt>>
                    SEM << list := One(idnt); >>
                    { ',' Id<<idnt>>
                      SEM << list := AppSE(list, One(idnt)); >>
                    }
                    '->' Expression<<e>>
                    SEM << e := Three(QsSE("_lambda"), list, e); >> .

ForeignFunctionId<<VAR e: SExp>>
              = '_' Id<<e>>
                SEM << e := Two(QsSE("_"), e); >>.

ExpList <<VAR e: SExp>>
              = LOCAL << VAR exp: SExp;>>
                '(' SEM <<e := One(CreateSE()); >>
                   [  Expression<<exp>>
                      SEM << e := One(exp); >>
                      { ',' Expression<<exp>>
                        SEM << e := AppSE(e, One(exp)); >>
                      }
                   ]
                ')'.

Sequence <<VAR e: SExp>>
              = LOCAL << VAR list: SExp;
                              ende: BOOLEAN;
                      >>
                Expression<<e>>
                SEM << ende := TRUE; >>
                [ ',' Sequence<<list>>
                  SEM << ende := FALSE;
                         e := ConsSE(e,list);
                      >>
                ]
                SEM << IF ende THEN
                          e := One(e)
                       END;
                    >>.

UnaryOp <<VAR e: SExp>>
              = '~'    SEM << e := QsSE("_not"); >> |
                'HD'   SEM << e := QsSE("_car"); >> |
```

```
                'TL'   SEM << e := QsSE("_cdr"); >> |
                '#'    SEM << e := QsSE("_len"); >> |
                'ATOM' SEM << e := QsSE("_atom"); >> .

MulOp <<VAR e: SExp>>
              = '*'   SEM << e := QsSE("_mul"); >> |
                '/'   SEM << e := QsSE("_quo"); >> |
                'DIV' SEM << e := QsSE("_div"); >> |
                'MOD' SEM << e := QsSE("_mod"); >> |
                '!'   SEM << e := QsSE("_nth"); >> |
                '@'   SEM << e := QsSE("_rest"); >>.

AddOp <<VAR e: SExp>>
              = '+'  SEM << e := QsSE("_add"); >> |
                '-'  SEM << e := QsSE("_sub"); >>.

RelOp <<VAR e: SExp; VAR neg: BOOLEAN>>
              = '='  SEM << e := QsSE("_eq");
                            neg := FALSE;
                         >> |
                '<'  SEM << e := QsSE("_le");
                            neg := FALSE;
                         >> |
                '<=' SEM << e := QsSE("_leq");
                            neg := FALSE;
                         >> |
                '>'  SEM << e := QsSE("_leq");
                            neg := TRUE;
                         >> |
                '>=' SEM << e := QsSE("_le");
                            neg := TRUE;
                         >> |
                '~=' SEM << e := QsSE("_eq");
                            neg := TRUE;
                         >> |
                'IN' SEM << e := QsSE("_member");
                            neg := FALSE;
                         >>.

 AndOp <<VAR e: SExp>>
              = '&'  SEM << e := QsSE("_and"); >>.

 OrOp <<VAR e: SExp>>
              = '|'  SEM << e := QsSE("_or"); >>.

 SeqOp <<VAR e: SExp>>
              = ':'  SEM << e := QsSE("_cons"); >>|
                '++' SEM << e := QsSE("_append"); >>.
END ISWIM.
```

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	Doktorski rad
Autor: AU	mr Zoran Budimac
Mentor: MN	dr Djura Paunić
Naslov rada: NR	Prilog teoriji funkcionalnih programskih jezika i implementaciji njihovih procesora
Jezik publikacije: JP	srpski (latinica)
Jezik izvoda: JI	s/en
Zemlja publikovanja: ZP	SR Jugoslavija
Uže geografsko područje: UGP	Vojvodina
Godina: GO	1994
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	6/186/146/0/115/0/1

Naučna oblast: Računarske nauke
NO
Naučna disciplina: Programski jezici
ND
Predmetna odrednica/Ključne reči:
PO Funkcionalno programiranje, implementacija funkcionalnih programskih jezika, medjujezik
UDK:
Čuva se:
ČU

Važna napomena: nema
VN
Izvod:
IZ Analizirani su važniji predstavnici čisto-funkcionalnih programskih jezika i važniji načini njihove implementacije. Na osnovu uočenih osobina, jezici su podeljeni na klase. Definisan je novi medjujezik za implementaciju čisto-funkcionalnih programskih jezika kojim je moguće predstaviti više klasa funkcionalnih programskih jezika nego postojećim medjujezicima. Konstruisani su algoritmi translacije 4 viša funkcionalna programska jezika u medjujezik i algoritmi prevodjenja medjujezika u mašinske jezike 5 apstraktnih mašina. Diskutovani su neki praktični aspekti implementacije medjujezika i izvršene analize performansi nekoliko realizovanih prevodilaca.
Datum prihvatanja teme od strane NN veća: jun 1993
DP
Datum odbrane:
DO
Članovi komisije:
KO
Predsednik: dr Dragan Acketa, vanredni profesor Prirodno-matematičkog fakulteta u Novom Sadu
Član: dr Živko Tošić, redovni profesor Elektronskog fakulteta u Nišu
Član: dr Dušan Tošić, docent Matematičkog fakulteta u Beogradu
Član: dr Djura Paunić, vanredni profesor Prirodno-matematičkog fakulteta u Novom Sadu

UNIVERSITY OF NOVI SAD
FACULTY OF NATURAL SCIENCES & MATHEMATICS
KEY WORDS DOCUMENTATION

Accession number: ANO	
Identification number: INO	
Document type: DT	Monograph documentation
Type of record: TR	Textual printed material
Contents code: CC	Doctoral thesis
Author: AU	Zoran Budimac
Mentor: MN	dr Djura Paunić
Title: TI	A Contribution to the Theory of Functional Programming Languages and to an Implementation of Their Processors
Language of text: LT	Serbian (Latin)
Language of abstract: LA	en
Country of publication: CP	FR Yugoslavia
Locality of publication: LP	Vojvodina
Publication year: PY	1994
Publisher: PU	Author's reprint
Publ. place: PP	Novi Sad, Trg D. Obradovića 4
Physical description: PD	6/186/146/0/115/0/1

Scientific field: Computer Science
SF
Scientific discipline: Programming languages
SD
Subject/Key words:
SKW Functional programming, Implementation of functional programming languages, Intermediate code

UC
Holding data:
HD

Note: none
N
Abstract:
AB Important purely functional languages and important ways of their implementation are analyzed. Based on observed characteristics, functional languages are divided into appropriate classes. A new specialized intermediate code for implementation of functional programming languages is defined, which enable a representation of more classes of high-level functional languages than existing intermediate codes. Algorithms for translation of four high-level functional languages into intermediate code are constructed, as well as algorithms for compilation of intermediate code into five abstract machine languages. Performance of several implemented compilers are analyzed.

Accepted on Scientific board on: June 1993
AS
Defended:
DE
Thesis Defend board:
DB
President: dr Dragan Acketa, associate professor, Faculty of Natural Sciences and Mathematics, Novi Sad
Member: dr Živko Tošić, professor, Faculty of Electronic Engineering, Niš
Member: dr Dušan Tošić, assistant professor, Mathematical Faculty, Belgrade
Member: dr Djura Paunić, associate professor, Faculty of Natural Sciences and Mathematics, Novi Sad