UNIVERZITET U BEOGRADU
PRIRODNO–MATEMATIČKI FAKULTET
MATEMATIČKI ODSJEK

Dževad Lugić

Termi i algoritmi

Magistarski rad

Split, 1991.

Komisija

1. S. Prešić, mentor
2. Ž. Mijajlović, predsednik komisije
3. Z. Marković,
4. D. Vitas
5. M. Bošić (odbranio)

Odbranio: 28.10.97, MF

Zahvaljujem se mom voditelju, dr. Slaviši Prešiću na strpljenju koje je pokazao u radu samnom, stalnoj podršci i velikoj pomoći pri izradi ove magistarske radnje. Posebnu zahvalnost dugujem svima koji su me podržavali u mom radu.

Termi i algoritmi

Dževad Lugić

5. februara 1991.

Sadržaj

Sažetak

Termi, poput $f(x)$, $g(a, f(y), 2)$, imaju veliki značaj u teoriji algoritama, pa prema tome i u višim programskim jezicima kao Pascal, C itd. Također su gradbeni elementi Predikatskog računa I reda, pa su zato nezaobilazni u logičkom programiranju i jezicima Lisp i Prolog. Recimo, u Pascalu je svaki izraz zapisiv termovski. Tako je naprimjer

$$\text{Begin}(:=(x,3),\ \text{if}(>(x,5),:=(y,7),:=(y,8)),$$

prevod u termovskoj algebri Pascalskog izraza:

```
Begin
  x:=3;
  if x>5 then y:=7
  else x:=8
End
```

U skladu sa prethodnim, odgovarajuća termovska algebra se javlja kao nadteorija navedenih programskih jezika. Rad se odnosi na istraživanje i rješavanje nekih problema Algebre terma. Tako se razmatraju ovakva pitanja:

- algoritam supstitucije,
- algoritam funkcije,
- algoritam rekurzije i sl.

Da bi se konkretno realiziralo postavljena pitanja, napravljen je mali funkcionalni jezik LLisp i na njemu su algoritmi testirani.

Uvod

Prvi programski jezik namjenjen obradi simboličkih podataka je funkcionalni jezik LISP razvijen na MIT-u 1960 g. pod vodstvom J. McCarty-a. Njegove osnove su izložene u radu [1]. Baziran je na *lambda računu* A. Church-a. Naziv jezika je kratica je od *"List Processing Language"*, što znači "Jezik za obradu lista", jer su podaci u njemu organizirani tako da imaju oblik lista. Jezik ima veliku fleksibilnost i snagu i izuzetno je zanimljiv jezik. Npr.

- Strukture podataka se kreiraju dinamički bez potrebe za alokacijom memorije.
- Deklaracije tipa podataka nisu potrebne, pa npr. jedna te ista varijabla može jednog trenutka imati za vrijednost objekt jednog tipa (npr. broj), drugi put imati za vrijednost drugu varijablu, treći put za vrijednost neku funkciju, četvrti put npr. objekt tipa binarnog drva (lista), itd.
- Mogućnost definicije funkcije bez imena, koja postoji samo u trenutku njene realizacije.

Ovakve njegove karakteristike čine ga pogodnim za istraživanja na području umjetne inteligencije, gdje je najviše i korišten. U zadnje vrijeme se u ta istraživanja uključuje i Prolog.
U ovom radu sam pokušao da prikažem neke od algoritama koji se koriste u svim jezicima, ali posebno s primjenom na Lisp. Algoritmi nisu dati samo teoretski, već je razvijen mali Lisp interpreter i u njemu primjenjeni ti algoritmi.

1 Lambda račun

Pod utjecajem teoreme Gödel-a o nepotpunosti, objavljenoj 1931.g., pojavio se novi pojam povezan sa algoritmima. To je bila klasa cijelih funkcija koje su se nazivale *rekurzivne funkcije*. Church je pomoću λ-apstrakcije[1] i β-preslikavanja, formirao tzv. *λ-račun* i ustanovio da je λ-definicija funkcije ekvivalentna rekurzivnoj funkciji. Neovisno od toga je Turing ustanovio da su funkcije 'izračunjive na Turing-ovom stroju' također ekvivalentne rekurzivnim funkcijama. I sva ostala istraživanja su dovodila do zaključka da su izračunljive funkcije ujedno i rekurzivne funkcije. U tim uvjetima je Church 1936.g. postavio tezu da je svaka intuitivno shvaćena izračunljiva funkcija rekurzivna. Dakle, svaka rekurzivna funkcija ima algoritam evaluacije, a s druge strane svaka funkcija za koju postoji algoritam evaluacije je rekurzivna.

1.1 Sintaksa λ-računa

Izrazi u λ-računu se predstavljaju u prefiksnoj formi, tj. operator se piše odmah na početku. Na taj način se umjesto $a/b + b * c$ piše $(+\ (/\ a\ b)\ (*\ b\ c))$. Proces traženja vrijednosti izraza naziva se *evaluacija* izraza. Evaluacija izraza se izvodi izborom i pojednostavljenjem onih dijelova tog izraza koji se mogu evaluirati. Dio koji se može pojednostavniti se naziva *redeks*, pri čemu je i sam redeks jedan izraz. Operacija pojednostavljenja se naziva *redukcija*. Transformaciju nekog izraza E u izraz F primjenom višestruke redukcije se označava sa $E \rightarrow F$. Proces redukcije je završen kad u izrazu, dobivenom redukcijom, više nema redeksa. Izraz koji više ne sadrži redekse, naziva se *normalna forma*. Proces redukcije izraza $A(= A_0)$, pri kojem se redom dobivaju izrazi $A_1,\ A_2,\ \ldots,\ A_n(= B)$, zapisuje se u obliku

$$A(= A_0) \rightarrow A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow \ldots \rightarrow A_n(= B).$$

Npr., pri evaluaciji izraza $(+\ (/\ 6\ 2)\ (*\ 2\ 5))$, prvo se izabiru redeksi $(/\ 6\ 2)$ i $(*\ 2\ 5)$ i uprošćavaju do 3 i 10. Pritom nikakvog značenja nema redosljed redukcija $(/\ 6\ 2)$ i $(*\ 2\ 5)$. Čak bi se mogli istovremeno reducirati (*paralelizam*)! Izraz $(+\ 3\ 10)$ dobiven redukcijom se može također reducirati u izraz 13. Kako je uobičajena redukcija s lijeva na desno, tada se proces pojednostavljenja zapisuje u obliku:

$$(+\ (/\ 6\ 2)\ (*\ 2\ 5)) \rightarrow (+\ 3\ (*\ 2\ 5)) \rightarrow (+\ 3\ 10) \rightarrow 13.$$

Rezultat 13 se ne može pojednostavniti, pa je dakle dobivena normalna forma.

1.1.1 λ-apstrakcija

Ideja λ-apstrakcije je u sljedećem. U izrazu $(f\ x)$ koji koristimo za izražavanje funkcije nejasno je: da li to označava funkciju f ili njenu vrijednost za zadanu vrijednost

[1] ili λ-funkcije

parametra x. Zato je za jasnu definiciju funkcije f uveden izraz $\lambda x(f\ x)$. Dakle, kada izraz M hoćemo razmatrati kao funkciju od x, koristimo zapis λxM.[2] Dobijanje izraza λxM iz M se naziva λ- *apstrakcija*, parametar x uz λ se naziva formalni parametar, izraz M *tijelo* λ-apstrakcije, a sam izraz λxM se naziva λ-izraz.
Npr. izraz $(+\ x\ 1)$ se može definirati kao funkcija $\lambda x(+\ x\ 1)$ koja varijabli x dodaje broj 1. λx kaže da je $\lambda x(+\ x\ 1)$ λ-apstrakcija, čiji je formalni parametar x, a $(+\ x\ 1)$ tijelo. Slično je $\lambda x(x + y)$ funkcija od x, a ne od y.

Na osnovi svega navedenog, sintaksa λ-izraza može se zadati na sljedeći način:

$$
\begin{aligned}
&< \lambda\text{−izraz} >::= \\
&\qquad < \text{konstanta} > \mid \\
&\qquad < \text{varijabla} > \mid \\
&\qquad (< \lambda\text{−izraz} >< \lambda\text{−izraz} >) \mid \\
&\qquad (\lambda < \text{varijabla} >< \lambda\text{−izraz} >)
\end{aligned}
$$

Tijelo λ-apstrakcije može biti bilo koji izraz. Tijelo λ-apstrakcije je maksimalni izraz nakon λ<ime varijable>.
Npr., u λ-izrazu:

$$((\lambda x\lambda y(+\ (*\ x\ y)\ (/\ y\ x)))\ 3\ 7),$$

tijelo λ-funkcije je izraz $(+\ (*\ x\ y)\ (/\ y\ x))$.

Ako je $f = \lambda xM$, tada se primjena $(f\ A)$ funkcije f na izraz A zapisuje u obliku $((\lambda xM)\ A)$. To se svodi na zamjenu izraza A u varijablu x unutar tijela M, što se označava sa $M[A/x]$ Dakle, možemo pisati

$$((\lambda xM)\ A) = \lambda A\ M[A/x].$$

1.2 Semantika λ-računa

Metoda redukcije se ne može primjeniti koristeći se samo λ-apstrakcijom. Za to su nam potrebna još tri pravila transformacije λ-izraza.

1.2.1 Vezana i slobodna varijabla

Promotrimo sljedeći λ-izraz:

$$((\lambda x(+\ x\ y))\ 3).$$

Da bi se gornji izraz evaluirao, potrebno je da y ima neku vrijednost. Što se tiče x-a, iz zapisa λx je jasno da je to formalni parametar i da primjena λ-izraza na 3 znači da se x-u daje vrijednost 3. Na taj se način varijabla koja je formalni parametar funkcije naziva *vezana varijabla*, a varijabla koja se ne nalazi uz λ (u gornjem izrazu y), se naziva *slobodna varijabla*. Tako vrijednost λ-izraza ovisi o vrijednosti slobodne varijable. Npr., u izrazu

$$((\lambda x(+\ ((\lambda y(+\ y\ z))\ 7)\ x)$$

[2] ili $(\lambda x\ M)$

x i y su vezane varijable, a z je slobodna varijabla.
Također, varijabla opisana jednim te istim simbolom može biti i vezana i slobodna u istom izrazu. Sve zavisi od njenog položaja u izrazu. Npr., u izrazu

$$(+\ x\ ((\lambda x(+\ x\ 1))\ 4))$$

prva varijabla x je slobodna, a druga je vezana. Dakle, to su dvije sasvim različite varijable.

Definicija slobodne varijable 1 *1. Varijabla x je slobodna, ako se izraz sastoji samo od nje same.*

2. Varijabla x je slobodna u izrazu E ili izrazu $F \Leftrightarrow$ x je slobodna varijabla u izrazu $(E\ F)$.

3. Varijabla x je slobodna varijabla u izrazu $E \Leftrightarrow$ x je slobodna varijabla u izrazu $(\lambda y E)$.

Definicija vezane varijable 1 *1. Varijabla x je vezana varijabla u izrazu E ili izrazu $F \Leftrightarrow$ x je vezana varijabla u izrazu $(E\ F)$.*

2. Varijable x i y su jednake i x je slobodna varijabla u izrazu E ili vezana varijabla u izrazu $E \Leftrightarrow$ x je vezana varijabla u izrazu $\lambda y E$.

1.2.2 Zamjena i njena svojstva

Ako se umjesto slobodne varijable x u izrazu E stavi izraz M, to se naziva *zamjena* i označavamo sa $E[M/x]$. Zamjena $E[M/x]$ se definira na na sljedeći način:

Definicija zamjene $E[M/x]$ 1 *1. $x[M/x] = M$*

2. $c[M/x] = c$, gdje je c ili konstanta ili proizvoljna varijabla različita od x.

3. $(E\ F)[M/x] = (E[M/x]\ F[M/x])$.

4. $(\lambda x E)[M/x] = \lambda x E$.

5. $(\lambda y E)[M/x] =$ (y se razlikuje od x)

(a) $= \lambda y E[M/x]$, ako je x vezana varijabla izraza E, ili ako je y vezana varijabla izraza M.

(b) $= (\lambda z E[z/y])[M/x]$, inače

(z je nova promjenjiva, koja nije slobodna u izrazima E ili M).

1.2.3 β-transformacija

λ-apstrakcija je samo formalizam za definiciju funkcije. Pitanje je kako izgleda pravilo za primjenu te funkcije funkcije na argument. Npr., u izrazu

$$((\lambda x(+\ x\ 1))\ 4)$$

zapisani su u nizu λ-apstrakcija $\lambda x(+\ x\ 1)$ i argument 4 i taj zapis znači da se λ-apstrakcija $\lambda x(+\ x\ 1)$ primjenjuje na argument 4.
Pravilo te 'primjene' je jednostavno i sastoji se u sljedećem:

"Rezultat primjene λ-izraza na argument je tijelo izraza koje se dobije zamjenom vezane varijable (formalnog parametra) λ-apstrakcije (koja je slobodna varijabla u tijelu λ-apstrakcije) sa argumentom".

To označavamo ovako:

$$(\lambda x M\ N) \Leftrightarrow M[N/x].$$

Tako u gornjem primjeru imamo:

$$((\lambda x\ (+\ x\ 1))\ 4) \Leftrightarrow (+\ x\ 1)[4/x] \Leftrightarrow (+\ 4\ 1).$$

To pravilo transformacije nazivamo *supstitucija* ili *β-transformacija*. Redukcija pomoću β-transformacije se naziva *β-redukcija* i označava pomoću simbola $\rightarrow$ na sljedeći način:

$$((\lambda x(+\ x\ 1))\ 4) \rightarrow (+\ 4\ 1) \rightarrow 5.$$

Evo još nekoliko primjera β-transformacija:

$$((\lambda x\ 3)\ 5) \rightarrow 3,$$

$$((\lambda x(+\ x\ x))\ 5) \rightarrow (+\ 5\ 5) \rightarrow 10,$$

$$((\lambda x((\lambda y(/\ y\ x))\ 6))\ 2) \rightarrow ((\lambda y(/\ y\ 2))\ 6) \rightarrow (/\ 6\ 2) \rightarrow 3.$$

Argument može biti i bilo koji izraz:

$$((\lambda f(f\ 3))\ (\lambda x(+\ x\ 1))) \rightarrow ((\lambda x(+\ x\ 1))\ 3) \rightarrow (+\ 3\ 1) \rightarrow 4.$$

Pri zamjeni vezane varijable sa argumentom, važno je primjetiti da jedno te isto ime varijable može biti jednom ime vezane, a drugi put ime slobodne varijable. Npr., u procesu redukcije

$$\begin{aligned}&(((\lambda x(((\lambda x(+\ (-\ x\ 1)))\ x)\ 3))\ 9)) \rightarrow \\ &\rightarrow ((\lambda x(-\ (-\ x\ 1))\ 3)\ 9) \rightarrow (+\ (-\ 3\ 1)\ 9) \rightarrow (+\ 2\ 9) \rightarrow 11\end{aligned}$$

u tijelu λ-izraza $(((\lambda x(+\ (-\ \underline{x}\ 1))\ x)\ 3)$ samo je prva (podcrtana) varijabla vezana, pa se pri prvoj β-transformaciji ona ne zamjenjuje.
Da bi pojednostavnili pisanje λ-izraza sa ugniježđenim strukturama i sa mnoštvom

zagrada, možemo npr. umjesto $(\lambda x(\lambda y(\lambda z E)))$ pisati $(\lambda x \lambda y \lambda z E)$.[3] Kod ovakvih struktura, neće biti dvosmislenosti. Korištenje β-transformacije u obratnom smjeru naziva se β-apstrakcija. Npr.,

$$(+\ 4\ 1) \rightarrow ((\lambda x(+\ x\ 1))\ 4).$$

Pokažimo sada jedan primjer u kome se koristi β-transformacija.
Definirajmo funkcije cons, car i cdr pomoću λ-apstrakcija:

$\text{cons} = (\lambda a \lambda b \lambda f(f\ a\ b))$.

$\text{car} = (\lambda c(c\ (\lambda a \lambda b\ a)))$,

$\text{cdr} = (\lambda c(c\ (\lambda a \lambda b\ b)))$.

Može se pokazati da su ovako definirane λ-apstrakcije, funkcije cons, car i cdr iz Lisp-a. One naime zadovoljavaju odgovarajuće aksiome za funkcije cons, car i cdr. Pokažimo to na primjeru jedne aksiome $(\text{car}(\text{cons}\ p\ q)) = q$:

$$\begin{aligned}
&(\text{car}\ (\text{cons}\ p\ q)) \\
&= ((\lambda c(c\ (\lambda a \lambda b\ a))(\text{cons}\ p\ q)) \\
&\rightarrow ((\text{cons}\ p\ q)(\lambda a \lambda b\ a)) \\
&= (((\lambda a \lambda b \lambda f(f\ a\ b))\ p\ q)(\lambda a \lambda b\ a)) \\
&\rightarrow ((\lambda f(f\ p\ q))(\lambda a \lambda b\ a)) \\
&\rightarrow ((\lambda a \lambda b\ a)\ p\ q) \\
&\rightarrow ((\lambda b\ p)\ q) \\
&\rightarrow p
\end{aligned}$$

1.2.4 α-transformacija

U λ-apstrakcijama

$$(\lambda x(+\ x\ 5))$$
$$(\lambda y(+\ y\ 5))$$

koriste se različite vezane varijable x i y. Međutim, rezultati β-transformacija, dobiveni primjenom tih apstrakcija na bilo koji argument, su jednaki. Dakle, ime vezane varijable nema stvarnog značaja. Na taj način, ako dopustimo promjenu izraza, tako da nema protivrječja između formalnog parametra i vezane varijable, tada se smisao izraza neće promijeniti.
Dakle, ako y nije slobodna varijabla u izrazu E. tada je $\lambda x E \Leftrightarrow \lambda y E[y/x]$. Ova transformacija naziva se *α-transformacija.*[4]
Prema tome

$$(\lambda x(+\ x\ 5)) \rightarrow (\lambda y(+\ y\ 5)) \ \text{ i } \ (\lambda x(+\ x\ 5)) \leftarrow (\lambda y(+\ y\ 5)),$$

[3] ili $(\lambda\ (x\ y\ z)\ M)$
[4] ili *zamjena imena vezane varijable*

što se kraće zapisuje sa

$$(\lambda x(+\ x\ 5)) \leftrightarrow (\lambda y(+\ y\ 5)).$$

Pri α-transformaciji se ne mora za novo ime varijable koristiti ime slobodne varijable u tijelu te λ-apstrakcije. α-transformacija se koristi da bi se izbjegli konflikti i dvosmislenosti prilikom zamjene imena u toku upotrebe β-transformacije.
Npr., ako koristimo β-transformaciju u izrazu

$$((\lambda y((\lambda x \lambda y(-\ x\ y))\ y)\ 4))\ 5),$$

tada dobijamo

$$((\lambda y((\lambda x \lambda y(-\ x\ y))\ y)\ 4))\ 5) \rightarrow ((\lambda y((\lambda y(-\ \underline{y}\ y))\ 4))\ 5) \rightarrow 0.$$

U tom je slučaju unutrašnja varijabla y (podcrtana) vezana u unutrašnjem λ-izrazu, što dovodi do nepravilnog rezultata.
Zato treba odmah na početku primjeniti α-transformaciju:

$$((\lambda y((\lambda x \lambda y(-\ x\ y))\ y)\ 4))\ 5) \rightarrow ((\lambda y((\lambda x \lambda z(-\ x\ z))\ y)\ 4))\ 5),$$

a zatim primjenom β-transformacije, dobijamo pravilan rezultat:

$$\begin{aligned}&((\lambda y((\lambda x \lambda z(-\ x\ z))\ y)\ 4))\ 5) \rightarrow \\ &\rightarrow ((\lambda y((\lambda z(-\ y\ z))\ 4))\ 5) \rightarrow ((\lambda z(-\ 4\ z))\ 5) \rightarrow (-\ 4\ 5) \rightarrow -1\end{aligned}$$

1.2.5 η-transformacija

η-transformacija je efikasno pravilo transformacije za uprošćenje složene λ-apstrakcije. Uzmimo npr. sljedeća dva izraza:

$$\begin{aligned}&(\lambda x(+\ 1\ x)) \\ &(+\ 1)\end{aligned}$$

Ako oba izraza primjenimo na proizvoljni argument, rezultat će biti dodavanje jedinice tom argumentu. Dakle:

$$\begin{aligned}&((\lambda x(+\ 1\ x))\ 5) \rightarrow (+\ 1\ 5) \rightarrow 6 \text{ ili} \\ &((+\ 1)\ 5) \rightarrow (+\ 1\ 5) \rightarrow 6.\end{aligned}$$

Drugim riječima, moguće je govoriti o nekom preslikavanju $(\lambda x(+\ 1\ x)) \rightarrow (+\ 1)$. Dakle, ako je x vezana varijabla u F i F je funkcija, tada je $\lambda x(F\ x) \Leftrightarrow F$. Ova transformacija se naziva *η-transformacija*[5] i ona je veoma efikasna s obzirom na uprošćenje λ-apstrakcije.

[5] ili *ekstenzija*

1.3 Rekurzivne funkcije

U λ-računu nema pridruživanja imena λ-izrazu, tj. koriste se samo bezimene funkcije. Tako je u osnovnom obliku λ-računa nemoguće definirati pojam rekurzije. Međutim, u funkcionalnom programiranju je pojam rekurzije veoma važan, pa bi korištenjem λ-računa bilo u njemu nemoguće rekurzivno definirati funkciju. Međutim ako λ-izrazu $(\lambda n(\text{if } (= n\ 0)\ 1\ (* n\ (fak\ (- n\ 1)))))$ pridružimo ime *fak*, tada smo sa

$$fak = (\lambda n(\text{if } (= n\ 0)\ 1\ (* n\ (fak\ (- n\ 1)))))$$

rekurzivno definirali funkciju *faktorijel*.

1.4 Poredak redukcije

Kako se forme $((\lambda x M)\ N)$ i $\boldsymbol{\lambda x}(M\ x)$ mogu pojednostavniti (reducirati) sa

$$((\lambda x\ M)\ N) \rightarrow M[N/x] \quad \text{i} \quad \lambda x(M\ x) \rightarrow M,$$

tada se primjenjujući β- i η-transformacije s lijeva na desno, može pojednostavniti i bilo koji složeni izraz. Takvo korištenje pravila se naziva *β-redukcija* i *η-redukcija*. Dakle, pojednostavljenje izraza u λ-računu se provodi periodičnom primjenom β- i η-redukcije i taj proces se naziva *evaluacija*. Rezultat evaluacije je izraz na koji se više ne mogu primjeniti gornja pravila redukcije. Takav se izraz zove *elementarni izraz* ili *elementarna forma*.
Evaluacija izraza M se završava kad se dobije elementarni izraz N, pri čemu se N naziva *rezultat evaluacije* izraza M (što ne znači da elementarna forma postoji za svaki izraz).

Način redukcije nekog izraza očigledno ovisi o poretku izbora redeksa i o poretku izbora pravila transformacije. Ako prilikom redukcije biramo uvijek prvi lijevi redeks, tada ustvari odlažemo evaluaciju argumenata funkcije i pristupamo evaluaciji same funkcije. Tada je potrebna evaluacija onih njenih dijelova u kojima se poziva na te argumente. Takav način redukcije nazivamo *redukcija normalnog poretka* i on odgovara *pozivu po imenu* u proceduralnim programskim jezicima.
Redukcija u kojoj, prije nego što pređemo redukciji redeksa $((\lambda x M)\ N)$, moramo prvo odrediti vrijednost N, naziva se *redukcija aplikativnog poretka*. Ona dakle odgovara *pozivu po vrijednosti* u proceduralnim jezicima.
Ako pri korištenju različitih poredaka redukcije dobijemo dvije različite elementarne forme, tada se one (bar teoretski) mogu transformirati jedna u drugu korištenjem α-transformacije.
Pri normalnom poretku, evaluacija λ-funkcije se može završiti sa neodređenim vrijednostima parametra funkcije, što je nemoguće pri aplikativnom poretku. Npr., u $((\lambda x\ (* 0\ x))\ 2)$ se $(* 0\ x)$ evaluira u 0, prije nego se primjeni β-apstrakcija.
S druge strane, pri aplikativnom poretku se evaluacija parametara funkcije radi samo

samo jednom, a pri normalnom poretku svaki put kada vrijednost parametra zatreba. 'Stroj' koji 'računa' koristeći se λ-računom, nazivamo *redukcijski stroj*.

Kako izraz obično sadrži nekoliko redeksa, postoji mnogo poredaka redukcije u zavisnosti od toga koji redeks izabiremo. Naprimjer, redukcija izraza $(+\ (*\ 4\ 5)\ (/\ 8\ 2))$ može se izvršiti na sljedeće načine:

$$\begin{aligned}(+\ (*\ 4\ 5)\ (/\ 8\ 2)) &\rightarrow (+\ 20\ (/\ 8\ 2)) \\ &\rightarrow (+\ 20\ 4) \\ &\rightarrow 24\end{aligned}$$

ili

$$\begin{aligned}(+\ (*\ 4\ 5)\ (/\ 8\ 2)) &\rightarrow (+\ (*\ 4\ 5)\ 4) \\ &\rightarrow (+\ 20\ 4) \\ &\rightarrow 24\end{aligned}$$

ili

$$\begin{aligned}(+\ (*\ 4\ 5)\ (/\ 8\ 2)) &\rightarrow (+\ 20\ 4) \\ &\rightarrow 24\end{aligned}$$

(U posljednjem se slučaju dijelovi $(*\ 4\ 5)$ i $(/\ 8\ 2)$ izračunavaju istovremeno (paralelno)). Ako izraz nema normalnu formu, redukcija se ne mora završiti u konačnom broju koraka. Npr., redukcija izraza $(D\ D)$, gdje je $D = (\lambda x(x\ x))$, se produžava u beskonačnost:

$$\begin{aligned}(D\ D) &= ((\lambda x(x\ x))\ (\lambda x(x\ x))) \\ &\rightarrow ((\lambda x(x\ x))\ (\lambda x(x\ x))) \\ &\rightarrow ((\lambda x(x\ x))\ (\lambda x(x\ x))) \\ &\rightarrow \ldots\end{aligned}$$

To odgovara beskonačnoj petlji u jeziku proceduralnog tipa.

Od načina izbora redeksa (tj. poretka redukcije) zavisi mogućnost određivanja normalne forme. Npr., ako za izraz $((\lambda x\ 3)\ (D\ D))$ prvo pokušavamo odrediti redukciju $(D\ D)$ (aplikativni poredak!), tada, pošto taj izraz nema normalnu formu, proces redukcije se proteže u beskonačnost. Međutim, ako prije redukcije izraza $(D\ D)$ primjenimo λ-apstrakciju $(\lambda x\ 3)$ na argument $(D\ D)$ (normalni poredak!), tada odmah dobijemo normalnu formu 3.

Na taj način za određivanje normalne forme veliki značaj ima način izbora redeksa,tj. poredak redukcije.

I tako se, s obzirom na način izbora redeksa i dobivanja normalne forme, postavljaju dva pitanja:

1. Da li su jednake normalne forme koje se dobiju iz jednog izraza na dva različita poretka redukcije?

2. Kako dobiti normalnu formu u minimalnom broju koraka?

Odgovor na prvo pitanje daju teoreme Church-Rossera.

Teorema Church-Rossera I. 1 *Neka je* $E_1 \leftrightarrow E_2$. *Tada postoji izraz* E *takav da je* $E_1 \rightarrow E, E \rightarrow E_2$.

Iz te teoreme slijedi sljedeća tvrdnja.

Posljedica 1 *Nijedan se izraz ne može transformirati u različite normalne forme (s tačnošću do* α*-transformacije).*

Ova tvrdnja nam pokazuje da se, za bilo koji proces redukcije, koji se završava u konačnom broju operacija, dobija uvijek jedna te ista normalna forma. Na taj način, proces redukcije ima jedno važno svojstvo, koje iskazuje sljedeći teorem.

Teorema Church-Rossera II. 1 *Neka je* $E_1 \rightarrow E_2$ *i* E_2 *normalna forma. Tada postoji redukcija normalnog poretka iz* E_1 *u* E_2.

Ova teorema kaže da, ako postoji normalna forma, obavezno postoji i redukcija normalnog poretka, pomoću koje se dobija ta normalna forma. Po njoj se uvijek bira krajnji lijevi redeks izraza i prvo se izvršava redukcija tog redeksa. Tako se naprimjer u $((\lambda x\ 3)\ (D\ D))$ izaborom krajnjeg lijevog redeksa $(\lambda x\ 3)$ omogućilo dobivanje pravilne normalne forme.

Drugo pitanje se tiče optimalnosti redukcije normalnog poretka. Obično redukcija normalnog poretka nije optimalan način redukcije. U slučaju tzv. *redukcije aplikativnog poretka*, prvo se bira najdalji redeks u izrazu. Prednost ovog načina je jednostavnija realizacija na računalu, ali mu je velika mana nedostatak garancije dobivanja normalne forme. Tako se u primjeru $((\lambda x\ 3)\ (D\ D))$, pri redukciji aplikativnog poretka, prvo bira redeks $(D\ D)$, a tada se normalna forma ne može dobiti.

1.4.1 Metode evaluacije funkcije

U Von Neumann-ovski baziranim računalima postoji nekoliko metoda evaluacije funkcije.

Proceduralne metode:

Ako je $P(x_1,\ x_2,\ \ldots,\ x_n)$ neka procedura i $f(x_1,\ x_2,\ \ldots,\ x_m)$ neka funkcija, a $P(y_1,\ y_2,\ \ldots,\ y_n)$ i $f(y_1,\ y_2,\ \ldots,\ y_m)$ njihovi pozivi, tada se svaki x_i naziva formalnim argumentom, a svaki y_i stvarnim argumentom. Pri njihovom pozivu je potrebno nekako povezati formalni sa stvarnim argumentom, za što postoji nekoliko načina, kao što su *poziv po vrijednosti*, poziv po adresi, poziv po tekstu, *poziv po imenu*, itd.

1. U *pozivu po vrijednosti* formalni argument x_i je lokalna varijabla koja postoji samo pri pozivu funkcije, odnosno procedure, a po njihovom napuštanju, formalni argument dobija početne vrijednosti (ako ih je imao).

2. U *pozivu po adresi*, ako je stvarni argument y_i varijabla, tada se formalni argument x_i podudara s njom. To znači da svaka promjena formalnog parametra x_i automatski mijenja i y_i.

3. U *pozivu po tekstu* se izraz u imenu stvarnog argumenta zapisuje u poziciju formalnog argumenta. Ako u zapisanom izrazu postoje lokalne varijable koje se koriste samo u toj proceduri ili funkciji i druge varijable istog imena, tada se pri pozivu po tekstu (jer je izraz takvog izgleda zapisan u funkciji ili proceduri) te varijable koriste kao lokalne. Npr. neka je data procedura $P(x, y) \equiv y = 3; \text{print}(x)$. Ako je do poziva procedure bilo $y = 1$ i $z = 2$, tada poziv $P(y+z, y)$ daje vrijednost 5, a ne 3. To je zato što je y ime lokalne varijable u funkciji. No vrijednost $y = 1$ neće se izmjeniti nakon okončanja poziva.

4. *Poziv po imenu* je sličan pozivu po tekstu, osim što ne dopušta mogućnost nedoumica u slučaju jednakih imena varijabli, kao poziv po tekstu. Ovdje se samo pri evaluaciji unutar procedure ili funkcije koriste veze među varijablama i njihove vrijednosti do poziva. Pri ovom pozivu bi vrijednost funkcije iz prethodnog primjera bila 3.

Funkcionalne metode:

1. Ako pri redukciji normalnog poretka odložimo evaluaciju argumenta pri pozivu funkcije i pristupimo evaluaciji same funkcije, tada je potrebna evaluacija njenih dijelova. Tu se dakle koristi poziv po imenu.

2. U poretku koji nije normalan, pristupa se prvo evaluaciji argumenata. Takav način odgovara pozivu po vrijednosti, a evaluacija koja ga koristi zove se *energična evaluacija*.

3. Funkcionalni jezici imaju i jednu metodu evaluacije koja se ne koristi u proceduralnim jezicima. To je evaluacija zasnovana na *pozivu po potrebi* i naziva se *lijena evaluacija*.

1.4.2 Lijena evaluacija

U proceduralnim jezicima se pri pokušaju izvođenja neke funkcije (komande) prvo izračunavaju (evaluiraju) njeni argumenti, tj. koristi se poziv po vrijednosti. Kako se u tijelu funkcije ne koriste uvijek svi argumenti, tada se izračunavanje svih argumenata funkcije pokazuje neracionalnim. U tom slučaju se pokazuje efektivnijim izračunavanje vrijednosti funkcije bez prethodnog izračunavanja argumenata, već u trenutku kada se ukaže potreba za vrijednošću argumenta. Takva metoda izračunavanja vrijednosti funkcije se zove *poziv po potrebi* [2].[6]

Međutim, ovaj se metod u proceduralnim jezicima ne koristi, zbog sljedećih razloga:

[6] *ili poziv po zahtjevu*

1. U proceduralnim jezicima se pri izračunavanju argumenta pojavljuje pobočni efekat, zbog čega se mogi dobiti različiti rezultati programa. Dakle, ako se trenutak izračunavanja argumenta ne može odrediti, nemoguće je tačno odrediti posljedice programa.

2. Pri korištenju steka teško je efektivno realizirati poziv po potrebi.

Evaluacija pri pozivu po potrebi obavlja se sa zadrškom, pa zato taj proces nazivamo *lijena evaluacija* [2, 7, 8].[7] Lijena evaluacija je veoma važan pojam u funkcionalnim jezicima (npr. u Lisp-u.). Razlog tome je mogućnost da se pomoću lijene evaluacije izbjegnu nepotrebna izračunavanja, obrade beskonačnih struktura i beskonačnog broja podataka.

Bit lijene evaluacije je u sljedećem:

1. Do evaluacije argumenta funkcije dolazi ne u trenutku primjene funkcije, već u trenutku izvršenja funkcije, kad dolazi do potrebe za njenom vrijednošću.

2. Evaluacija argumenta se izvodi samo jedanput, iako se koristi na više mjesta u procesu izvršenja funkcije.

Primjer:
Neka je $(f\ x\ y) \equiv (\text{if}\ (>\ x\ 0)\ 1\ (*\ (y\ (+\ y\ 1))))$. Ako treba odrediti $(f\ (g\ a)\ (h\ a))$ i ako je $(>\ (g\ a)\ 0)$, tada je jasno da ne treba računati $(h\ a)$. U evaluaciji aplikativnog poretka je u sličnim slučajevima evaluacija $(h\ a)$ neophodna.
Lijena evaluacija je pogodna s tačke evaluacije izraza, ali je problematična s tačke brzine izvođenja programa. Zato se u praksi koriste većinom oni funkcionalni jezici koji ne koriste lijenu evaluaciju.
S gledišta poretka redukcije, lijena evaluacija odgovara redukciji normalnog poretka, a energična evaluacija redukciji aplikativnog poretka.

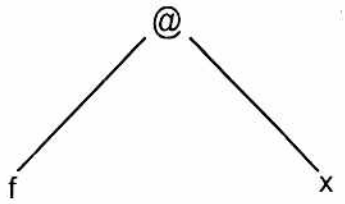
1.4.3 Redukcija grafova

Izraz koji podliježe redukciji, se u memoriji računala realizira u obliku strukture grafa i uprošćenjem te strukture efektivno izvrši redukcija izraza [6]. Jednostavnija struktura grafa je struktura drva i izraz koji ne sadrži rekurziju možemo predočiti pomoću strukture *drvo*. Imamo nekoliko načina predstavljanja izraza pomoću drva, a u svima je problem kako predstaviti primjenu funkcije na svoje argumente.
Općenito, primjena funkcije f na argument x se označava sa $(f\ x)$, a ako ima više argumenata $x_1,\ x_2,\ \ldots,\ x_n$ sa $(f\ x_1\ x_2\ \ldots\ x_n)$. Funkciju nekoliko argumenata možemo interpretirati kao rezultat 'sinteze' funkcija jednog argumenta. Npr., ako izraz (+ 3 5) napišemo kao ((+ 3) 5), tada (+ 3) možemo promatrati kao funkciju koja nekom

[7] *ili evaluacija sa zadrškom*

argumentu (ovdje 5) pribraja vrijednost 3. Samu tu funkciju možemo smatrati rezultatom primjene funkcije + na argument 3. Kako se (+ 3) ne može reducirati, to je normalna forma. Tako primjenom (+ 3) na 5 imamo redukciju ((+ 3) 5) $\rightarrow$ 8. Predstavljanje funkcija pomoću funkcija jedne varijable se naziva *karizacija*(prema H.B. Curry [5]).[8] Kako velik broj zagrada otežava razumjevanje izraza, umjesto ((+ 3) 5) pisaćemo (+ 3 5), ali misliti na ((+ 3) 5).
Uzmimo primjer $(f\ x)$ primjene funkcije f na argument x.

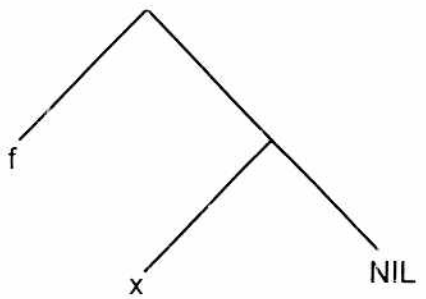
1. U prvom slučaju se u listovima drva nalaze konstante, imena funkcija ili varijable, a u vrhu drva znak @. Tako se $(f\ x)$ može predočiti sljedećim grafom:



Slika 1: Graf unarne funkcije (I)

Znak @ kaže da se funkcija na lijevoj strani primjenjuje na argument na desnoj strani.

2. Drugi način se razlikuje od prvog u tome što se uvodi konstanta *NIL*, koja označava nepostojanje argumenata. Znak u vrhu drva nije potreban.



Slika 2: Graf unarne funkcije (II)

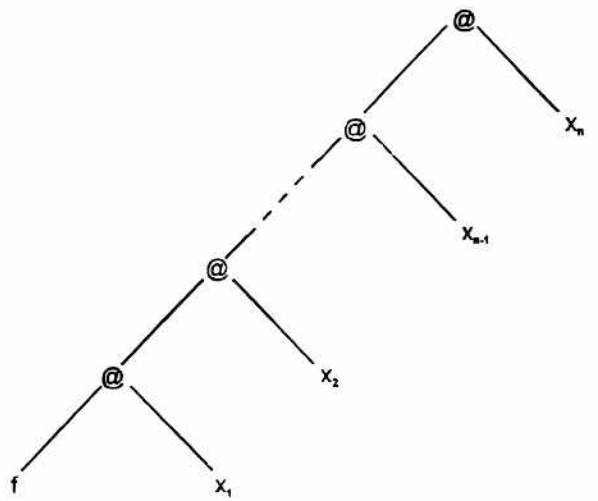
3. Treći način u vrhovima drva ima oznake operacija, a u listovima argumente. Ovaj način je pogodan za slučaj binarnih operacija (kao npr. $(f\ a\ b)$), ali nije u slučaju $(f\ a\ b\ c)$ ili sl., jer se mora odstupiti od ideje binarnog drva koja je korištena u prethodna dva slučaja.

[8] Drugi naziv-Optimizacija Curry

Slika 3: Graf binarne i ternarne funkcije

Zato promotrimo samo prva dva slučaja.

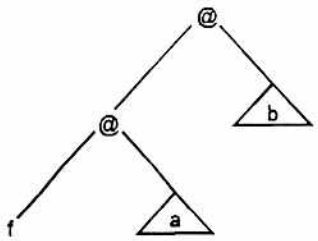
1. Ako želimo primjeniti funkciju na više argumenata, npr. $(f\ x_1\ x_2\ \ldots\ x_n)$, možemo se poslužiti metodom karizacije. Prvo f primjenimo na x_1 i dobijemo funkciju $(f\ x_1)$, zatim tu funkciju primjenimo na x_2 i dobijemo $((f\ x_1)\ x_2)$, zatim rezultat primjenimo na x_3 itd. Nastavljajući dalje, na kraju dobivenu funkciju primjenimo na argument x_n. Graf tako definirane funkcije je:



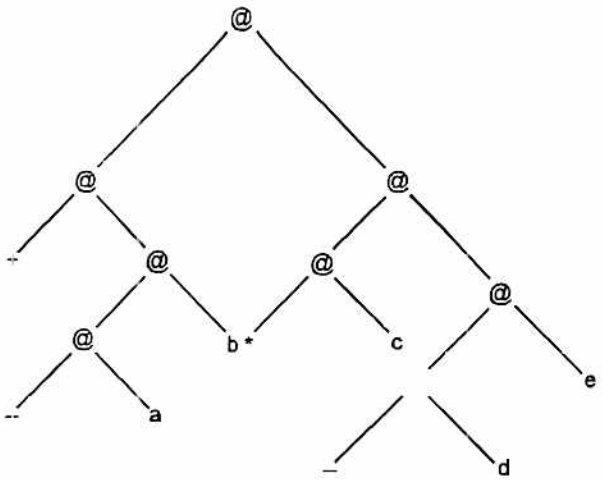
Slika 4: Graf funkcije od n argumenata (I)

(a) Ako sa a i b označimo bilo koja dva izraza, a sa f funkciju, tada se izraz $(f\ a\ b)$ može predstaviti pomoću grafa na slici 5.

(b) Tako se npr. izraz $(+\ (-\ a\ b)\ (*\ c\ (-\ d\ e)))$ može prikazati u obliku grafa na slici 6.

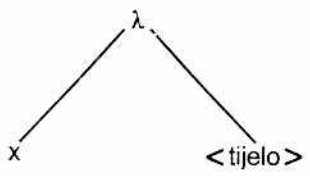


Slika 5: Graf binarnog izraza



Slika 6: Graf izraza (+ (- a b) (∗ c (- d e)))

(c) λ-apstrakcija ($\lambda x <$ *tijelo* $>$) ima sljedeći graf:

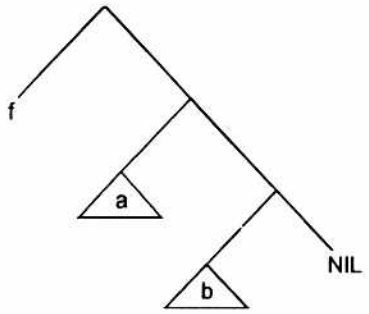


Slika 7: Graf λ-apstrakcije

Vrh λ nam kaže da se niže toga nalazi λ-apstrakcija, da je formalni parametar na lijevoj strani, a tijelo na desnoj.

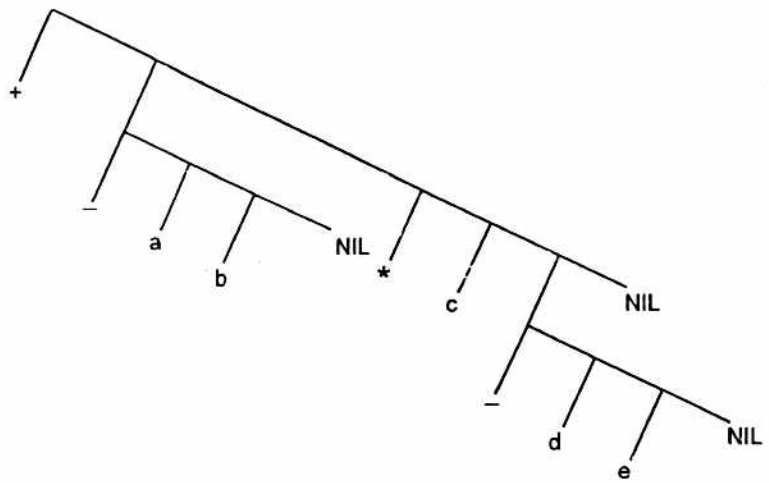
2. Sada pogledajmo drugi način predstavljanja.

   (a) Ako su a i b bilo koja dva izraza, a f neka funkcija, tada je drvo izraza $(f\ a\ b)$ dato slikom:



Slika 8: Graf binarnog izraza

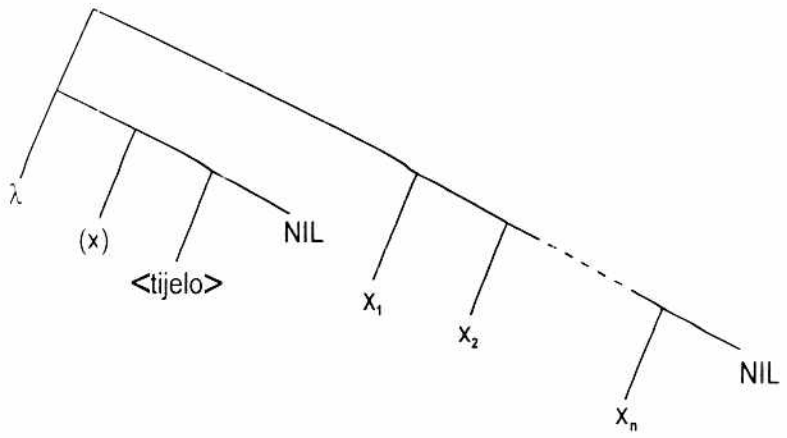
   (b) Tako je drvo izraza $(+\ (-\ a\ b)\ (*\ c\ (-\ d\ e)))$ dato sljedećom slikom:



Slika 9: Graf izraza (+ (- a b) (* c (- d e)))

3. λ-apstrakciju $(\lambda x < tijelo >)$ ćemo u ovom slučaju predočavati grafom sa slike 10. List λ nam kaže da se u desnoj grani nalazi λ-apstrakcija, da su svi formalni parametri u prvoj idućoj lijevoj podgrani, a tijelo u drugoj idućoj lijevoj podgrani.

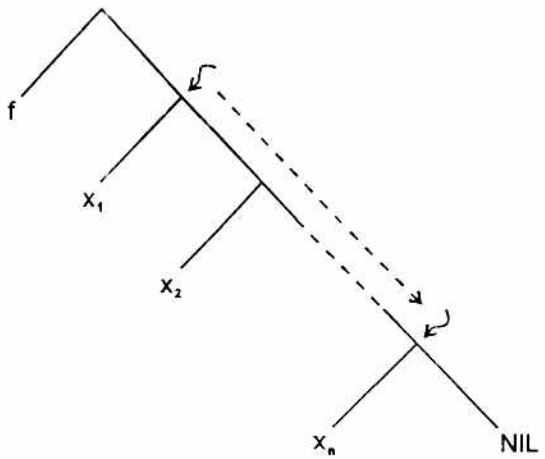
   Smatram da drugi način predstavljanja ima prednosti, pa ga ovdje i predstavljam (i kasnije realiziram).



Slika 10: Graf λ-apstrakcije

Kako proces redukcije normalnog poretka sigurno završava, razmotrićemo detaljnije proces redukcije normalnog poretka.

Ako je zadan graf izraza $(f\ x_1\ x_2\ \ldots\ x_n)$ tada se promatra list f lijeve grane.

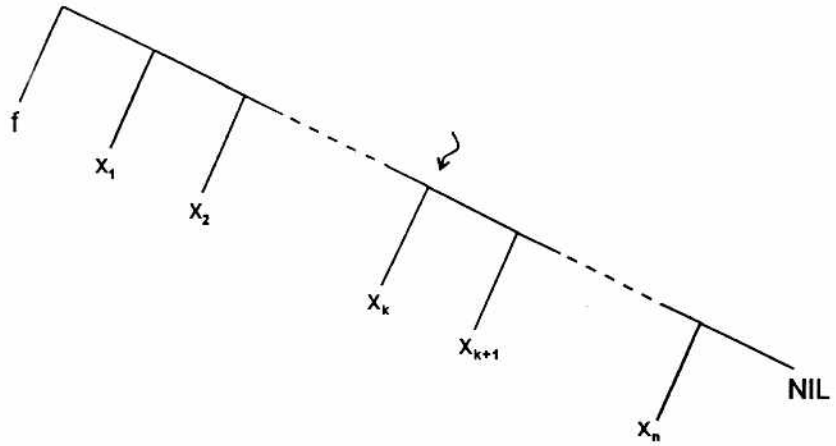


Slika 11: Graf funkcije od n argumenata (II)

f može biti podatak, sistemska ili korisnički zadana funkcija i λ-apstrakcija.

Podatak Podatak mora biti broj, neka druga nenumerička konstanta ili varijabla. Ako pritom nije $n = 0$ javi se greška, a inače je redukcija odmah završena, a rezultat je taj podatak.

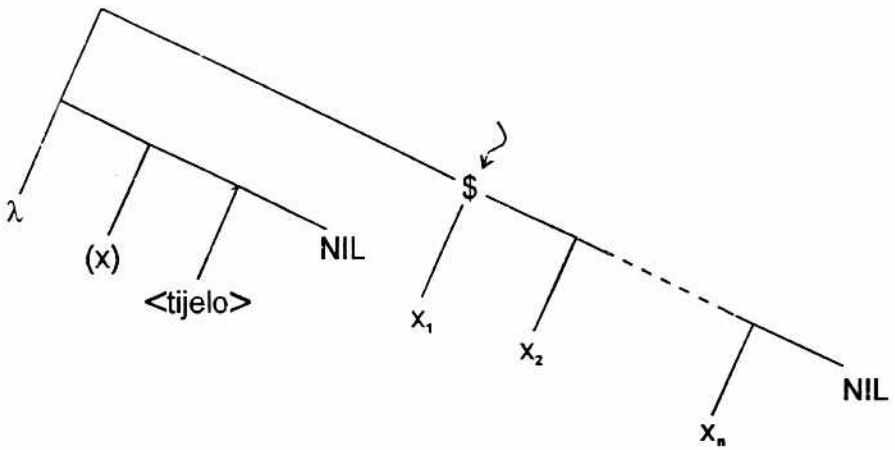
Sistemski ili korisnički zadana funkcija od k argumenata Ako je $n \geq k$, tada je $(f\ x_1\ x_2\ \ldots\ x_k)$ krajnji lijevi redeks. Ako je $n < k$, redukcija je nemoguća,



Slika 12: Primjena ugrađene funkcije

pa izraz ostaje neizmjenjen.

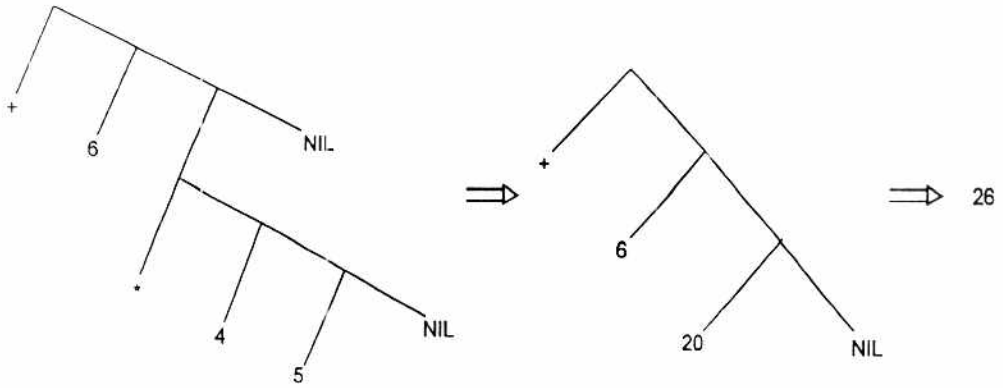
λ-apstrakcija Ako je $f \equiv (\lambda x < tijelo >)$, tada je redeks $(f\ x_1)$, tj. $((\lambda x < tijelo >)\ x_1)$. Korjenski vrh tog redeksa je označen sa \$. Ako je $n = 0$, argumenata nema, redukcija je nemoguća, pa izraz ostaje neizmjenjen.



Slika 13: Primjena λ-apstrakcije

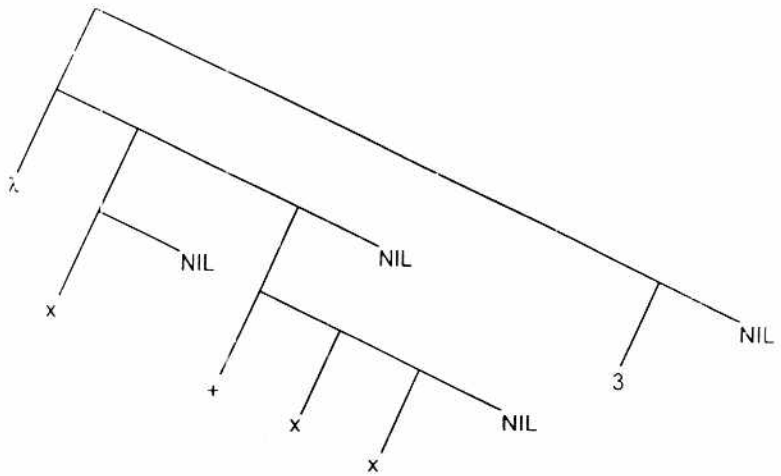
Nakon što je redeks nađen, pogledajmo na jednostavnim primjerima, na koji se način provodi redukcija normalnog tipa na grafu:

a) U slučaju (+ 6 (* 4 5)) za izračunavanje sume trebaju svi argumenti, pa se zato izračunava izraz (* 4 5), a zatim (+ 6 20).



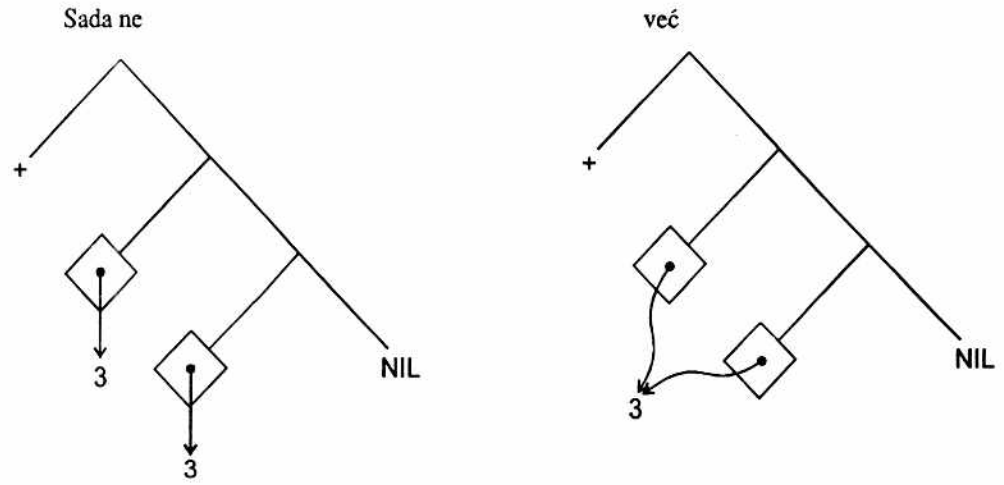
Slika 14: Redukcija grafa izraza $(+\ 6\ (*\ 4\ 5))$

b) $((\lambda x(+\ x\ x))\ 3) \rightarrow (+\ 3\ 3) \rightarrow 6$
Pri zamjeni stvarnog argumenta 3 umjesto formalnog argumenta x u slučaju β-redukcije, potrebno je za svaki argument napraviti kopiju i promatrati načine zamjene tom kopijom. Kako za formiranje kopije treba i vrijeme i memorija, a ta kopija može sadržavati i redeks, tada pri redukciji normalnog poretka s pozivom po potrebi treba uraditi redukciju za svaku kopiju, što znači da taj postupak nije dovoljno efikasan. Zato umjesto njega možemo koristiti zamjenu pointerom argumenta (grafa) bez formiranja kopije, tako da se redukcija obavlja samo na jednom mjestu.



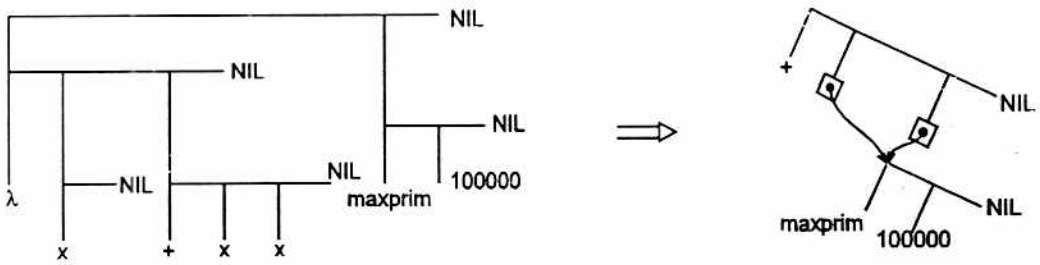
Slika 15: Graf izraza $((\lambda\ x\ (+\ x\ x))\ 3)$

c) To je još više izraženo na primjeru izraza $((\lambda x(+\ x\ x))$ (maxprim 100000)), gdje funkcija *maxprim* traži najveći prost broj manji od 100000. Kako ar-



Slika 16: Redukcija grafa $((\lambda\ x\ (+\ x\ x))\ 3)$

gument sadrži redeks (maxprim 100000), funkcija *maxprim* treba mnogo vremena i memorije za izračunavanje najvećeg prostog broja manjeg od 100000. Korištenjem kopije argumenta ovaj se izraz izračunava dvaput, a korištenjem pointera jedanput. Metoda zasnovana na ovakvom korištenju pointera se zove *metoda općeg grafa.*



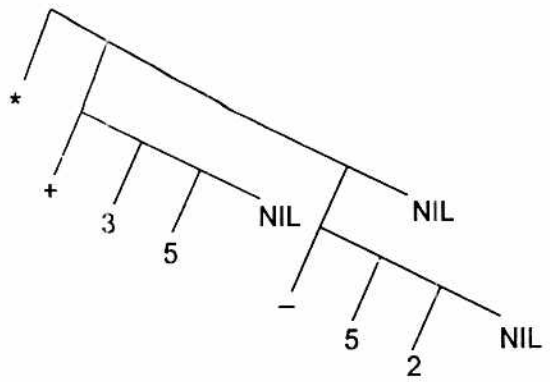
Slika 17: Redukcija grafa $((\lambda x\,(+\ x\ x))$ (maxprim 100000))

Vidljivo je da redukcija grafa ima dva stadija. Prvi se naziva *silazni stadij* i do njega dolazi kad se proces redukcije odvija od korjena grafa k listovima. Kad proces redukcije dođe do listova, tada započinje *uzlazni stadij.* U ovom stadiju se dešava redukcija od listova ka korjenu grafa. Ako izbacimo prvi stadij, i redukciju započinjemo sa drugim stadijem, imaćemo redukciju aplikativnog poretka.[9]

[9]To je jasno, jer redukcija aplikativnog poretka u traganju za argumentima funkcija ide sve do listova i odatle započinje redukciju

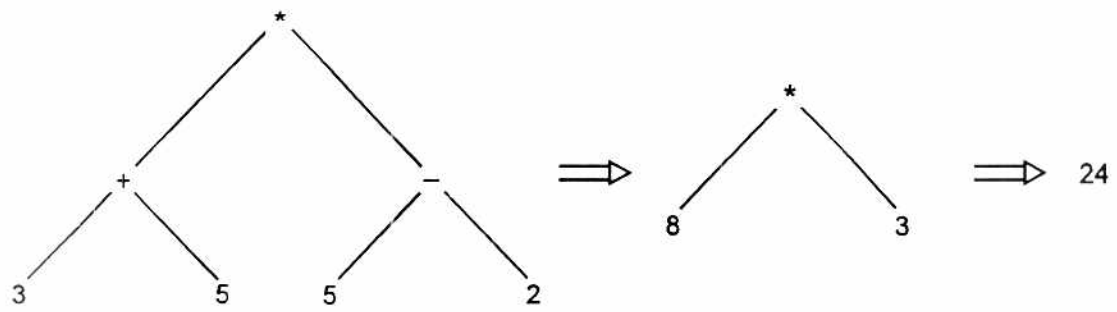
1.4.4 Paralelna redukcija

Dosad smo pretpostavljali da se redukcija izvršava redom na po jednom redeksu. Ako izraz sadrži nekoliko redeksa, moguće ih je istovremeno reducirati. Kako se funkcionalni program razlikuje od od tradicionalnog proceduralnog programa i pošto rezultat programa ne ovisi o tome kako se i kojim redom ostvaruje redukcija redeksa, tada se pomoću paralelne redukcije može lako ubrzati evaluaciju izraza. I u slučaju paralelne redukcije se koristi predstavljanje u obliku grafa. Na donjoj slici je prikazan



Slika 18: Drvo izraza (∗ (+ 3 5) (- 5 2))

graf izraza (∗ (+ 3 5) (- 5 2)). Za objašnjenje pojma procesa paralelne redukcije, možemo ga predstaviti i sljedećim grafom iz kojeg je vidljivo da se izrazi (+ 3 5) i (- 5 2) mogu izračunati nezavisno (paralelno).



Slika 19: Paralelna redukcija drva izraza (∗ (+ 3 5) (- 5 2))

2 Strukture podataka i strukture upravljanja

Svaki programski jezik kojim se na računalu zapisuje neki program, mora zadovoljavati određene uvjete.

- On mora imati mogućnost predstavljanja *struktura (tipova) podataka* koji su objekti obrade programa.
- Također, taj jezik mora imati sredstva za predstavljanje metoda obrade tih podataka. tj. *strukture upravljanja* ili *algoritme.*

Strukture podataka i strukture upravljanja su veoma povezane i čine osnov jezika programiranja, tj. određuju funkcionalne mogućnosti jezika.

2.1 Strukture podataka

Osnovni zadaci u strukturama podataka su:

a) Kako predstaviti podatke?

b) Kako organizirati podatke iz formiranih struktura podataka?

Već prema tome na koji način i u kom obliku predstavljamo podatke, strukture podataka možemo razmatrati kao apstraktne i konkretne.

2.1.1 Apstraktne strukture podataka

Apstraktne strukture se predstavljaju na algebarski način. Ideja je u tome da se nad skupom podataka $\mathcal{S}$ zada skup operacija $\mathcal{O}$ i aksiome $\mathcal{A}$ tih operacija. Podaci se tada definiraju kao struktura generirana tim operacijama. Korištenje algebarskih sredstava je povezano sa mogućnošću primjene matematičkih metoda na strukture podataka, kao npr. teorije asocijativnih i komutativnih grupa.

Pri razmatranju tipova podataka, u početku se aksiomatski zadaju neki početni tipovi podataka, koje nazivamo osnovnim tipovima podataka. Takvi su npr.:

1. **atomi**
   (a) logički (boolean) tipovi $\top$ i $\bot$
   (b) simbolički tipovi (karakteri)
   (c) cijeli i realni brojevi

2. **uređeni par**

Ovi tipovi imaju apstraktne defininicije. Npr. definicija logičkog tipa može biti sljedećeg oblika:

Tip *logički*
Operacije :
T :↦ logički
NIL :↦ logički
not : logički ↦ logički
and : logički × logički ↦ logički
or : logički × logički ↦ logički
Aksiome :
not(T) = NIL
not(NIL) = T
and(T, T) = T
and(T, NIL) = NIL
and(NIL, T) = NIL
and(NIL, NIL) = NIL
or(T, T) = T
or(T, NIL) = T
or(NIL, T) = T
or(NIL, NIL) = NIL
Kraj *logički*

Ovim su ustvari zadane istinosne tablice za operacije and i or. Atomi T i NIL ovdje reprezentiraju istinosne vrijednosti $\top$ i $\bot$.

Složene strukture podataka su *lista*, *drvo* i *graf*.

Uzmimo primjer podatka tipa *lista*. Kao osnovne operacije nad listama uvedimo operaciju konstrukcije liste cons, operacije dekompozicije liste car i cdr, (unarnu) operaciju nil kojom se definira 'prazna lista' i operaciju null koja provjerava da li je lista prazna ili ne.
Definicije tih operacija zapisujemo na sljedeći način:

Operacije : $y : a$
nil :↦ lista
cons : (a, lista) ↦ lista
car : lista ↦ a
cdr : lista ↦ lista
null : lista ↦ logički

Simbol a koji se gore pojavljuje, označava tip podatka svakog elementa neprazne liste. Kako taj podatak očigledno može biti bilo kojeg tipa, *listu* nazivamo polimorfnim tipom podataka.
Kako su apstraktno definirane operacije car i cdr matematičke funkcije, tada one

moraju biti definirane za svaku listu. Zato ćemo ih dodefinirati i za 'praznu listu' nil. Zato, ako stavimo $car(\text{nil}) = \text{nil}$ i $cdr(\text{nil}) = \text{nil}$, zbog definicija $car : lista \mapsto a$ i $cdr : lista \mapsto lista$, neće biti problema.

Iz prethodne definicije objekta *lista*, također nije jasno koja je veza između operacija *car*, *cdr* i *null*. Zato se uz definiciju podatka mora navesti i aksiome koje te operacije nad tim podatkom moraju zadovoljiti. Za listu tako imamo sljedeće tri aksimome:

Aksiome : $x : lista;\ y : a$
$\text{cons}(\text{car}(x), \text{cdr}(x)) = x$
$\text{null}(\text{nil}) = \top$
$\text{null}(\text{cons}(a, x)) = \bot$

Pomoću definicija operacija i aksioma, sada je formalno potpuno definiran apstraktni tip podataka *lista*.
Ako istinosne vrijednosti $\top$ i $\bot$ respektivno reprezentiramo sa T i NIL (pri čemu nema nedoumica i ako umjesto NIL pišemo znak 'prazne liste' nil ili (), i obratno), tada konačna apstraktna definicija objekta *lista* glasi:

Tip *lista*
Operacije : $y : a$
nil :$\mapsto$ lista
cons : $(a, \text{lista}) \mapsto$ lista
car : lista $\mapsto a$
cdr : lista $\mapsto$ lista
null : lista $\mapsto$ logički
Aksiome : x : lista; $y : a$
$\text{cons}(\text{car}(x), \text{cdr}(x)) = x$
$\text{null}(\text{nil}) = \text{T}$
$\text{null}(\text{cons}(a, x)) = \text{NIL}$
Kraj *lista*

Postoji još jedan način definicije liste. Nazivamo ga *metoda terma.*[10] U tom slučaju operacija cons od dva terma obrazuje novi term, tj.:

$$\text{cons} : (x, y) \mapsto (x|y)$$

Znak '|' se naziva *konstruktor terma*. Zato se može pisati i $|(x, y)$. Funkcije car i cdr tada predstavljaju operacije koje vraćaju lijevi i desni dio terma, formiranog operacijom cons. Atom (lista) nil se predstavlja termom nil. Funkcija null ispituje da li je term jednak nil ili nije.
Ova definicija također zadovoljava aksiomu liste cons(car(L), cdr(L))=L.

Naime, ako je term $L = M|N$ konstruiran od dva terma M i N, tada dobijamo

$$\text{cons}(\text{car}(L), \text{cdr}(L)) = \text{cons}(M, N) = M|N = L.$$

[10]Koristi se u Prolog-u

Također, očigledno je

$$\text{null(nil)} = \text{T} \text{ i } \text{null(cons}(a, L)) = \text{NIL}.$$

Apstraktna definicija liste je tada realizirana u univerzumu svih takvih terma.

2.1.2 Konkretne strukture podataka

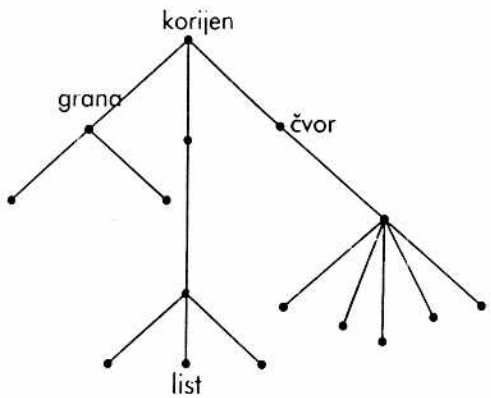
Konkretne strukture podataka se predočavaju na sljedeći način.

1. **Atom** se reprezentira tačkom a u prostoru osnovnih podataka S.

2. **Uređeni par** $(a\ b)$ se reprezentira slikom:

Slika 20:

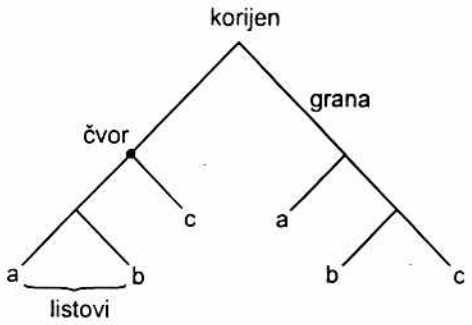
3. **Drvo** se reprezentira slikom:



Slika 21:

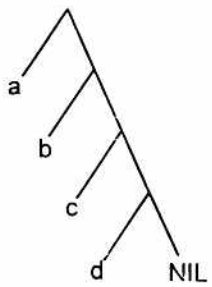
Vrh drva nazivamo *korjenski vrh* ili *korjen*, dole lijevo i desno su *čvorovi* povezani *granama* sa korjenom, pod čvorovima su drugi čvorovi povezani grana ma sa vrhom itd. Na dnu su *listovi* u kojima se smješteni osnovni podaci.

4. **Binarno drvo** ima istu reprezentaciju kao i uređeni par, osim što su listovi sada atomi ili druga binarna drva.



Slika 22:

5. **Lista** $(x_1\ x_2\ x_3\ \ldots\ x_n)$ je poseban slučaj binarnog drva. Npr. lista $(a\ b\ c\ d)$ se prikazuje sa



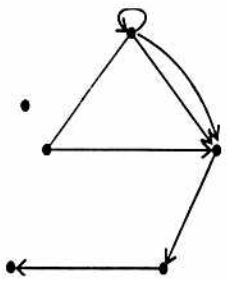
Slika 23:

6. **Graf** može biti veoma složena struktura. Npr. graf je struktura na slici 24.

2.2 Strukture upravljanja

U strukturama upravljanja se radi ili o sekvencijalnom ili paralelnom upravljanju. U našem slučaju se razmatra (klasično) sekvencijalno upravljanje. Sekvencijalno upravljanje mora riješiti probleme:

- Strukture uvjetnog grananja **(if ... then ... else)**
- Strukture cikličkog izvršavanja funkcija
  (repeat ... until ... ili **while ... do ...)**



Slika 24:

- Strukture poziva funkcije (procedure) i povratka iz nje
- Strukture rekurzivnog poziva funkcije
- itd

3 Strukture podataka u LLispu

Objekti u LLisp-u se predstavljaju pomoću terma i zovu se $*$-*termi* (*'zvijezda' termi*). Ti se objekti se ne interpretiraju, sami objekti su termi. Oni mogu biti iz jedne od sljedećih kategorija:

- atom,
- tačasti par.

3.1 Atom

Atom je:

- numerički atom,
- alfanumerički atom,
- specijalni simbol.

a) Numerički atomi:
Oni mogu biti cijeli ili racionalni brojevi.
Npr.:

$$6$$
$$-124$$
$$3.162$$

b) Alfanumerički atomi (ili identifikator):

Npr.:

$$B$$
$$r12$$
$$Atom$$

c) Specijalni simboli:

To su znakovi operacija:

$$+, -, *, /, <, =, >, '$$

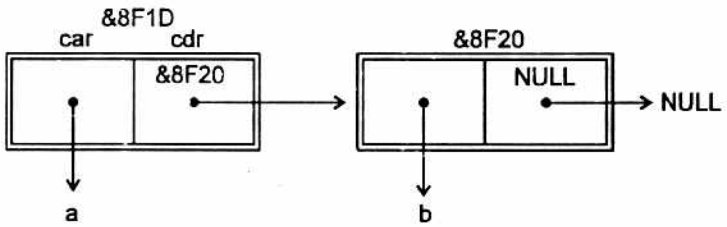
i znak _.

3.2 Tačkasti par

Tačkasti par je uređeni par $*$-terma koji se označava sa $(a.b)$ (odatle 'tačkasti par'). U ovom radu smo uveli oznaku $(a|b)$ jer ćemo . koristiti u drugom smislu. Objekt a nazivamo *prvi član*, b *drugi član* tačkastog para, a | se može shvatiti kao binarni operator formiranja tačkastog para.

3.2.1 Realizacija tačkastog para

Strukturu tačkastog para realiziraćemo pomoću pointera. Pointeri su, jednostavno rečeno, strukture pomoću kojih se realizira pristup adresama. Prostor u računalu u kojem su smješteni ti pointeri nazivamo *ćelija*.
Svaka ćelija se sastoji od dijelova *car* i *cdr* koji sadrže pointere na drugi tačkasti par ili atom. Ćelija *car* sadrži pointer na prvi, a ćelija *cdr* na drugi član tačkastog para.



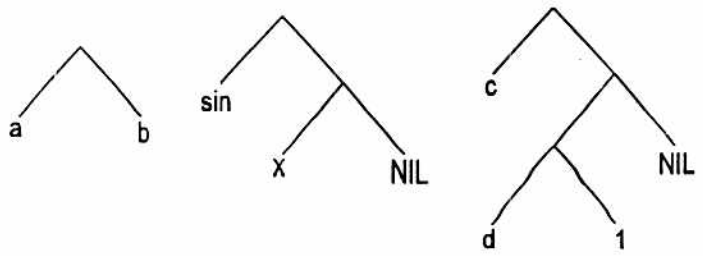
Slika 25: LLisp ćelija

Tačkasti parovi su npr.:

$$(a|b),$$
$$(\sin|(x|\mathrm{NIL})).$$
$$(c|((d|1)|\mathrm{NIL})),$$

a njihovi grafovi

U prvom slučaju a i b su pointeri na alfanumeričke atome a i b.



Slika 26: Grafovi tačkastih parova

Alfanumerički atom NIL u drugom i trećem slučaju ima specifično značenje i označava nepostojanje pointera, tj. označava kraj lanca pointera u nizu. On je dakle reprezent liste bez elemenata, tzv. "prazne" liste (). Zato se drugi primjer može uzeti kao način prikazivanja liste (sin x).[11]

3.3 Liste

Često se informacija (test knjige, matematička formula itd.) može predstaviti kao niz simbola. Takav niz se lako može predočiti u LLisp-u. Npr. tekst *ovo je lista* možemo predočiti u obliku $(ovo|(je|(lista)))$. Kako je niz simbola veoma popularan način predstavljanja podataka, a $*$-term ne isuviše pogodan za prikazivanje većih nizova simbola, u LLisp je uveden novi, izvedeni, tip podatka *lista*.

Definicija liste 1 *1. () je lista.*

2. Ako je a $*$-term. a b lista, tada je $(a|b)$ lista.

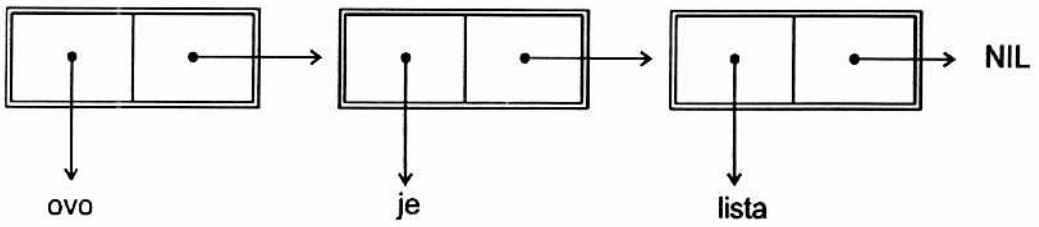
3. Lista se dobije samo primjenom konačnog broja gornjih pravila.

Na osnovi toga svaka je lista ujedno i tačkasti par, ali obratno ne važi. Tako je $(ovo|(je|(lista))$ lista, ali $(ovo|(je|lista))$ nije. Prvi član a liste se naziva *glava liste*, a drugi član *rep liste*.

Lista sastavljena od $*$-terma $s_1, s_2, \ldots, s_n$ se predstavlja u obliku linearne strukture $(s_1\ s_2\ \ldots\ s_n)$, a prazna lista kao ().
Dakle možemo pisati:

$$NIL \leftrightarrow ()$$
$$(s_1|(s_2|(s_3|(\ldots(s_n|NIL)\ldots))) \leftrightarrow (s_1\ s_2\ \ldots\ s_n)$$

gdje $\leftrightarrow$ predstavlja ekvivalentnost.
Na taj se način $(ovo|(je|(lista|\text{NIL})))$ može predočiti listom $(ovo\ je\ lista)$. U memoriji

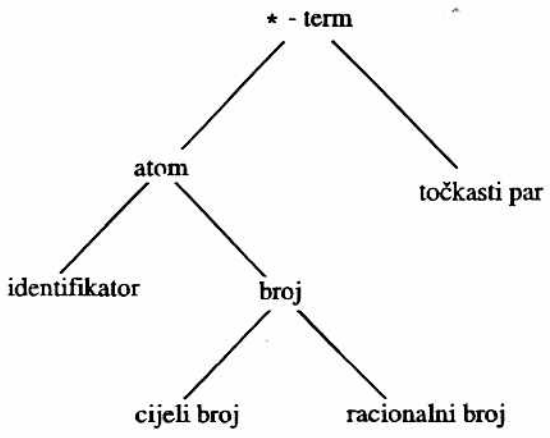


Slika 27: Realizacija liste '(ovo je lista)'

će to biti predočeno slikom 27. Dakle, unutrašnja reprezentacija liste i tačkastog para je identična.
Lista je slična polju, ali nije statička struktura, već dinamička, tj. može se povećavati i smanjivati. Također može sadržavati proizvoljan broj elemenata različitih tipova. Liste se također razlikuju od dinamičkih podataka u drugim jezicima jer je lista u Lispu jednostavan podatak. Naime:

- mogu se pridružiti varijabli,
- mogu biti ulazni parametar funkcije,
- mogu biti izlazni parametar funkcije.

Veze između svih ovih podataka mogu se prikazati slikom:

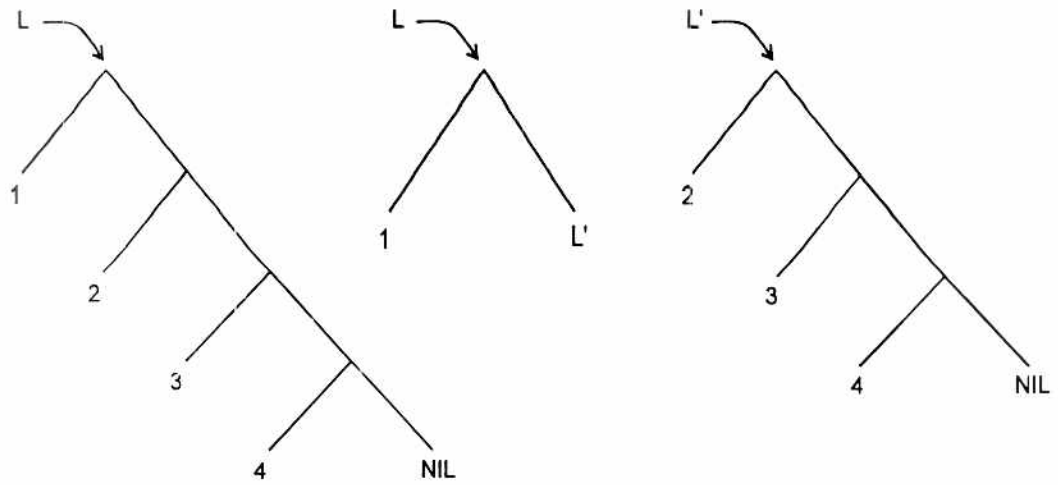


Slika 28: Veze između LLisp tipova podataka

[11] Ako tačkasti par promatramo kroz njegovu realizaciju, kao binarnog drva, prvi član para se može zvati **glava**, a drugi **rep** tačkastog para.

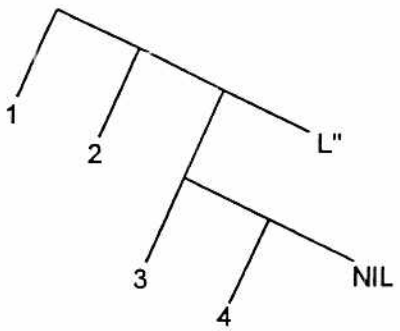
3.3.1 Veza između liste i tačkastog para

Osim za praznu listu NIL, lista je uvijek ekvivalentna nekom tačkastom paru. Npr. ako sa L označimo listu (1 2 3 4), tada se L može predstaviti u obliku (1|L'), gdje je 1 glava, a L' rep (2 3 4) liste L.



Slika 29: Drvo liste L, tj. $(1|L')$

Slično se (1 2 (3 4)|L") može predstaviti binarnim drvom



Slika 30: Drvo liste $(1\,2\,(3\,4)|L")$

Kako tačkasti parovi $(a|(b|(c|\text{NIL})))$, $(a|(b|(c)))$, $(a|(b\ c))$ i $(a\ b\ c)$ imaju jednake reprezentacije, možemo reći da je time zadana relacija ekvivalencije '=' lista. S obzirom na to možemo navedene parove smatrati ekvivalentnim. odnosno 'jednakim'. Tako su npr., koristeći činjenicu da je NIL ekvivalentno sa () i činjenicu da je $(a|\text{NIL})$ isto što i (a), tada su sljedeća četiri $*$-terma ekvivalentna:

(NIL|NIL), (()|()), (()), (NIL).

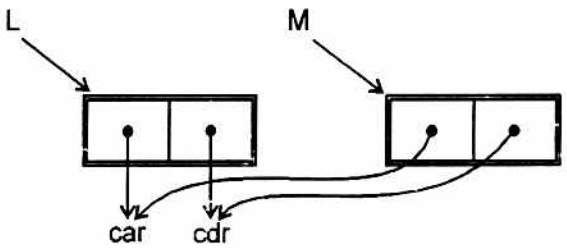
Pokažimo da na ovaj način realizirana lista udovoljava apstraktno danim aksiomama liste. Operacija cons od dva pointera formira *car-cdr* ćeliju. car i cdr operacije vraćaju car- i cdr- dio ćelije. Operacija null ispituje da li je bilo koji pointer ćelije, pointer NIL.
Očigledno je ispunjena aksioma

$$\text{cons}(\text{car}(L), \text{cdr}(L)) = L$$

Dalje, očigledno je null(NIL)=T i null(cons(a,L))=NIL, jer se operacijom cons ne može dobiti pointer NIL.
Prema tome ovakva realizacija udovoljava aksiomama liste. Možemo reći da je izrazom

$$\text{cons}(\text{car}(L), \text{cdr}(L)) = L$$

faktički zadana relacija ekvivalencije '=' lista. U konkretnoj realizaciji se pointer



Slika 31: Realizacija operacije *cons*

na novu ćeliju M stvorenu operacijom cons ne mora podudarati sa pointerom na početnu ćeliju L. Međutim, koristeći se navedenom aksiomom, liste L i M se smatraju ekvivalentnim, jer imaju jednake car- i cdr- dijelove.

4 Sintaksa LLisp-a

Jedan tako fleksibilan jezik kao što je LLisp, je izuzetno teško opisati sintaktičkim navodima. Svaki jezik, pa i LLisp, bi bio potpuno zadan sa:

- Skupom dozvoljenih simbola
- Skupom pomoćnih znakova
- Skupom riječi
- Skupom pravilnih programa
- Značenjem programa

Pokušajmo opisati svaki od navedenih zahtjeva.

- Skup (dozvoljenih) simbola je:

$$a, ..., z, A, ..., Z, +, -, *, /, 0, ..., 9, ., |, ', \_, <, >, =, ;$$

- Pomoćni znakovi su (,)

- Niz simbola, jedan za drugim, čine riječi. Riječi mogu reprezentirati operacije, konstante i varijable.[12] Dakle jezik LLisp-a je skup $J = Op \cup Cons \cup Var$, pri čemu je:

  - $Op = \{+, -, *, /, >, <, =$, sin, cos, exp, ln, abs, sgn, set, setq, defun, eval, apply, lambda, print, read, cond, if, progn, quote, ', car, cdr, cons, list, append, member, and, or, atomp, pairp, listp, idp, constp, eq, equal, numberp, zerop, minusp, null, length, atomlist, cls, mem, load, exit$\}$
  - $Cons = \{$T, NIL, LAMBDA, UNDEF, racionalni brojevi, operacije$\}$
  - Var je skup varijabli. Za njih se dozvoljava da budu samo alfanumerici.

- Nad tako definiranim jezikom J mogu se graditi termi. Ima nekoliko znakova koje pri gradnji terma ne treba odvajati prazninama, iako su posebni dijelovi naredbi programa. Ti znakovi se zovu *separatori*. Separatori su:

$$+, -, *, /, =, <, >, |, ', ;, (,)$$

Dakle, riječi u LLispu su $*$-termi, tj. atomi i tačkasti parovi. Formalizirajmo njihove definicije iz prethodnog poglavlja. Pritom ::= znači "jednako po definiciji", | znači "ili", a u $< \ldots >$ su objekti.

Definicija atoma 1 *<slovo>* ::=*a*|*b*|*c*|*d*|*e*|*f*|*g*|*h*|*i*|*j*|*k*|*l*|*m*|*n*|*o*|*p*|*q*|*r*|*s*|*t*|*u*|*v*|*w*|*x*|*y*|*z*|
A|*B*|*C*|*D*|*E*|*F*|*G*|*H*|*I*|*J*|*K*|*L*|*M*|*N*|*O*|*P*|*Q*|*R*|*S*|*T*|*U*|*V*|*W*|*X*|*Y*|*Z*

<cifra>::=1|2|3|4|5|6|7|8|9|0

<cijeli broj bez znaka>::=*<cifra>* | *<cijeli broj bez znaka><cifra>*

<cijeli broj>::=*<cijeli broj bez znaka>* |+ *<cijeli broj bez znaka>* |
− *<cijeli broj bez znaka>*

<racionalni broj>::=.*<cijeli broj>* | *<cijeli broj>* .|
<cijeli broj>.*<cijeli broj bez znaka>*

<broj>::= *<cijeli broj>* | *<racionalni broj>*

[12] Operacije su konstante u LLisp-u!

<identifikator>::=<slovo> | <slovo><cifra> |
_ <identifikator> | <identifikator> _|
<identifikator><identifikator>

<specijalni simbol>::=_|'| + | − | ∗ |/| > | = | < |;

<atom>::=<broj> | <identifikator> | <specijalni simbol>

Atom može reprezentirati konstante ili varijable, tj. možemo atom definirati i ovako:

Definicija atoma 2 <atom>::=<konstanta> | <varijabla>

Sljedeće dvije definicije sintaktički definiraju konstantu i varijablu u LLisp-u:

Definicija konstante 1 <alfanumerička konstanta>::=NIL, T, LAMBDA, UNDEF

<operacija>::=Op

<konstanta>::=<broj> | <operacija> | <alfanumerička konstanta>

Definicija varijable 1 *Varijabla je svaki identifikator koji nije konstanta.*

Konačno:

Definicija tačkastog para 1 <tačkasti par>::= (<atom> | <atom>)|
(<atom> | <tačkasti par>)|
(<tačkasti par> | <atom>)|
(<tačkasti par> | <tačkasti par>)

Definicija liste 1 <lista>::= $NIL|(<*\text{-term}> \mid <\text{lista}>)$

Definicija ∗-terma 1 *1.* <∗-term>::=<atom> | <tačkasti par>

2. ∗-term se dobije samo konačnom primjenom gornjeg pravila.

Programi su međutim 'pravilni' termi nad jezikom *J*. Na koji način graditi te terme trebalo bi opisati pomoću gramatičkih pravila. Međutim, za ovakav jezik to je prilično teško. Stvar pojednostavljuje činjenica da je program u LLisp-u također jedna funkcija. Sintaksu i semantiku programa opisaćemo u dijelu o semantici LLisp-a.

5 Semantika LLisp-a

Da bi se u LLisp-u izvršila neka operacija, potrebno je napisati neku naredbu. To se radi pomoću specijalne liste koju zovemo *funkcija*. Njen prvi član je ime funkcije, a ostali članovi su argumenti funkcije (dozvoljava se da funkcija nema ulaznih argumenata). Argumenti mogu biti bilo koji $*$-termi.
Da bi objasnili osnovne funkcije LLisp-a, moramo opisati tipove argumenata koji se koriste. Uvedimo sljedeće skraćenice:

- n je broj ili numerički atom,
- a je alfanumerički atom,
- l je bilo koja lista,
- s je bilo koji $*$-term,
- f je funkcija,
- p je predikat.

Sada navodimo sintaksu i semantiku funkcija LLisp-a.

5.1 Funkcija *quote*

Ovo je specijalna funkcija, jer ona sprečava evaluaciju svog argumenta. Njena definicija je (quote s) = s.
Tako je npr. (quote (+ 1 2)) = (+ 1 2).
Postoji i kraći zapis te funkcije. Umjesto (quote s) pišemo $'s$, pa je npr.

$$'a = a \text{ ili } '(+\ 1\ 2) = (+\ 1\ 2).$$

5.2 Selektori

Selektori su funkcije koje iz $*$-terma odabiru neki dio. Zato su oni definirani samo na *tačkastim parovima*. Tačkasti par je jednostavno uređeni par $*$-terma. Funkcije car i cdr izabiru redom prvi i drugi član $*$-terma. Dakle one se mogu definirati sa:

$$\text{(car } (s_1|s_2)) = s_1,$$
$$\text{(cdr } (s_1|s_2)) = s_2,$$

gdje su s_1 i s_2 neki $*$-termi.
Kako je i lista neki tačkasti par, tj. $(s_1\ s_2\ \ldots\ s_n) = (s_1|(s_2\ \ldots\ s_n))$, imamo:

$$\text{(car } (s_1\ s_2\ \ldots\ s_n)) = s_1,$$
$$\text{(cdr } (s_1\ s_2\ \ldots\ s_n)) = (s_2\ \ldots\ s_n).$$

Od toga postoji jedan izuzetak. Lista NIL ne sadrži nijedan element. Zato smo dodefinirali funkcije car i cdr sa (car NIL) = NIL i (cdr NIL) = NIL, tako da su funkcije sada definirane za svaku listu.

5.3 Konstruktori

Konstruktori su funkcije koje od dijelova prave cijelinu.
Takva je funkcija cons. Ona od dva $*$-terma pravi tačkasti par. Njena definicija je:

$$(\text{cons } s_1\ s_2) = (s_1|s_2).$$

Međutim konstrukcija liste pomoću *cons* je nezgrapna. Npr.

$$(\text{cons } x\,(\text{cons } y\ (\text{cons } z\ \text{nil}))) = (x|(y|(z|\text{nil}))) = (x\ y\ z).$$

Zato se za konstrukciju liste koristi funkcija *list*. Njena definicija je:

$$(\text{list } s_1\ s_2\ \ldots\ s_n) = (s_1\ s_2\ \ldots\ s_n).$$

Za konstrukciju liste od drugih lista koristi se funkcija append. Njena sintaksa je:

$$(\text{append } l_1\ l_2\ \ldots\ l_n).$$

Npr.

$$(\text{append } (x_1\ x_2)\ (x_3\ x_4\ x_5)\ \text{nil}\ (x_6)) = (x_1\ x_2\ x_3\ x_4\ x_5\ x_6).$$

5.4 Aritmetičke funkcije

Funkcije + i $*$ su uobičajene funkcije sume i produkta, i mogu imati proizvoljan broj argumenata. Dakle:

$$(+\ n_1\ n_2\ \ldots\ n_k) = n_1 + n_2 + \ldots + n_k,$$

$$(*\ n_1\ n_2\ \ldots\ n_k) = n_1 * n_2 * \ldots * n_k.$$

Funkcije $-$ i / određuju razliku i količnik dva broja:

$$(-\ n_1\ n_2) = n_1 - n_2 \ \text{ i } \ (/\ n_1\ n_2) = n_1/n_2.$$

Funkcije $>, =, <$ su definirane kao:

$$\begin{array}{lll} (>\ n_1\ n_2) & = \text{T}, & \text{ako je } n_1 > n_2, \\ & = \text{NIL} & \text{inače.} \\ (=\ n_1\ n_2) & = \text{T}, & \text{ako je } n_1 = n_2, \\ & = \text{NIL} & \text{inače.} \\ (<\ n_1\ n_2) & = \text{T}, & \text{ako je } n_1 < n_2, \\ & = \text{NIL} & \text{inače.} \end{array}$$

Funkcije sin, cos, exp, ln, abs, sgn su uobičajene matematičke funkcije.

5.5 Predikati

Funkcije koje omogućuju razlikovanje $*$-terma, nazivaju se *predikati*. Njihov zadatak je da provjere zadovoljavaju li njihovi argumenti određeni uvjet. U skladu s tim predikat vraća vrijednost koja se interpretira kao laž ili istina. Ovdje smo uveli dva posebna atoma T i NIL da označe istinu i laž. Ali osim T omogućili smo da svaki $*$-izraz različit od NIL označava istinu. Dakle moglo bi se reći da u LLispu nije na snazi dvoznačna, već dvoipoznačna logika.
Rekli smo da su atom i tačkasti par dva osnovna tipa podataka u LLisp-u. S tim u vezi imamo predikate *atomp* i *pairp*:

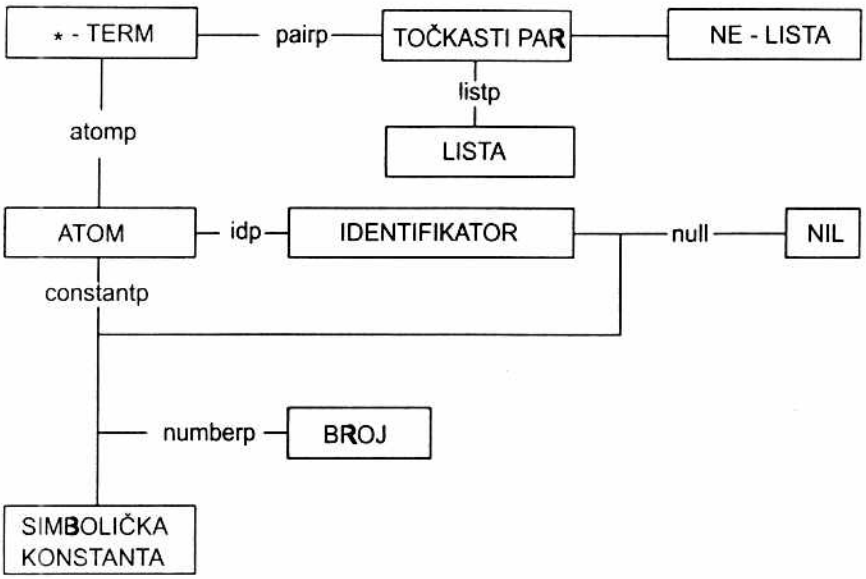
(atomp s)	= T	ako je s atom,
	= NIL	inače,
(pairp s)	= T	ako je s točkasti par,
	= NIL	inače.

Ostali tipovi razlikuju se pomoću sljedećih predikata:

(numberp s)	= T	ako je s broj,
	= NIL	inače.
(zerop n)	= T	ako je $n = 0$,
	= NIL	inače.
(minusp n)	= T	ako je $n < 0$,
	= NIL	inače.
(evenp n)	= T	ako je n paran broj,
	= NIL	inače.
(oddp n)	= T	ako je n neparan broj,
	= NIL	inače.
(null s)	= T	ako je s prazna lista NIL,
	= NIL	inače.
(listp s)	= T	ako je s lista,
	= NIL	inače.
(boundp a)	= T	ako je a vezana varijabla,
	= NIL	inače.
(idp s)	= T	ako je s identifikator,
	= NIL	inače.
(constp s)	= T	ako je s konstanta,
	= NIL	inače.

Funkcije eq i equal razlikuju $*$-terme međusobno:

(eq s_1 s_2)	= T	ako su izrazi s_1 i s_2 identični, tj. unutrašnja reprezentacija je identična,
	= NIL	inače.
(equal s_1 s_2)	= T	ako su s_1 i s_2 jednaki,
	= NIL	inače.



Slika 32: Tipovi podataka LLisp-a i predikati koji ih razlikuju

Definicija dvomjesnog predikata member:

$$\begin{aligned}(\text{member } s\ l) &= \text{T} \quad \text{ako je izraz } s \text{ element liste } l,\\ &= \text{NIL} \quad \text{inače.}\end{aligned}$$

5.6 Uvjetni izrazi

U LLisp-u imamo dvije vrste uvjetnih izraza. Jedni su u vezi sa funkcijom if, a drugi u vezi sa funkcijom cond.
Funkcija if je funkcija tri argumenta i njena definicija je:

$$\begin{aligned}(\text{if } p\ f_1\ f_2) &= f_1 \quad \text{ako je } p \neq \text{NIL.}\\ &= f_2 \quad \text{inače.}\end{aligned}$$

Funkcija cond je univerzalnija funkcija od funkcije if, a ima i neke druge posebnosti. Ovo je jedina funkcija koja ne može evaluirati svoje argumente. Njena sintaksa je dosta složena:

$$\begin{aligned}(\text{cond}\ \ &(p_1\ f_{11}\ f_{12}\ \ldots\ f_{1n_1})\\ &(p_2\ f_{21}\ f_{22}\ \ldots\ f_{2n_2})\\ &\ldots\\ &(p_m\ f_{m1}\ f_{m2}\ \ldots\ f_{mn_m})).\end{aligned}$$

p_i označavaju uvjete koji se ispituju, a f_{kl} neke funkcije. Ispituju se redom uvjeti p_i u svakom od argumenata funkcije cond, sve dok se ne naiđe na onaj koji nije NIL. Ako se takav ne nađe, rezultat je NIL, a ako se nađe, evaluiraju se redom svi izrazi

f_{kl} u tom argumentu, a koji se nalaze iza tog uvjeta. Rezultat posljednjeg izraza je rezultat funkcije cond.

5.7 Logičke funkcije

Logičke funkcije su not, or i and. Definicija funkcije not je:

$$\begin{aligned}(\text{not } p) &= \text{T} \quad \text{ako je } p \neq \text{NIL},\\ &= \text{NIL} \quad \text{inače.}\end{aligned}$$

Iz definicije vidimo da je funkcija not identična sa funkcijom null. Veznici and i or su funkcije neodređenog broja argumenata. Oni su definirani sa:

$$\begin{aligned}(\text{or } p_1\ p_2\ \ldots\ p_n) &= p_L \quad \text{gdje je } p_L \text{ prvi argument po redu različit od NIL,}\\ &= \text{NIL} \quad \text{inače.}\\ (\text{and } p_1\ p_2\ \ldots\ p_n) &= \text{NIL} \quad \text{ako je bar jedan argument jednak NIL,}\\ &= p_n \quad \text{inače.}\end{aligned}$$

5.8 Funkcije pridruživanja

Ove funkcije dodjeljuju ime nekom $*$-termu. $*$-term može biti funkcija, ali i definicija funkcije.
To su funkcije *set*, *setq* i *defun*.
Funkcija *set* sa (set f s) = y, vrijednosti prvog argumenta f dodjeljuje vrijednost drugog argumenta s i tu vrijednost vraća kao rezultat.
Funkcija setq sa (setq a s) = y, neevaluiranom prvom argumentu a dodjeljuje vrijednost drugog argumenta s i tu vrijednost vraća kao rezultat.
Funkcija *defun* sa (defun a l f_1 f_2 ... f_n) neevaluiranom prvom argumentu a dodjeljuje λ-definiciju funkcije sa listom argumenata l i tijelima f_1, f_2, ..., f_n i vraća a kao ime te funkcije. λ-definicija funkcije je realizacija pojma λ-apstrakcije.

5.9 Funkcije ulaza-izlaza

To su funkcije *read*, *print*, *load* i *exit*.
Sa (read) se u trenutku evaluacije u program ubacuje proizvoljni izraz.
Sa (print s) se izraz s ispisuje na ekranu računala.
Sa (load f) se u memoriju računala učitava program čije je ime rezultat funkcije f.
Sa (exit) se izlazi iz programa.

5.10 Funkcije evaluacije

To su funkcije *eval*, *apply* i *lambda*.

Osnovna funkcija u LLisp-u je *eval*. Ona potiče evaluaciju (izračunavanje) $*$-terma.

(eval s) vraća kao rezultat evaluiranu vrijednost $*$-terma.
Slična je funkcija *apply*. Sa (apply s l) se primjenjuje term s na listu argumenata l i rezultat toga vraća kao rezultat funkcije *apply*. Ova funkcija je primjer realizacije pojma primjene funkcije λ-apstrakcije u sistemu λ-računa.
Sintaksa funkcije *lambda* je (lambda l s_1 s_2 ... s_n), a njen rezultat je definicija λ-apstrakcije, sa listom argumenata l i tijelima s_1, s_2, ..., s_n.

5.11 Funkcija progn

Program u LLisp-u se može sastojati od niza funkcija, koje međusobno predaju podatke jedna drugoj kroz globalne varijable. Svi ti pozivi funkcija mogu se objediniti u jednu funkciju, tako da možemo zaista reći da je cijeli LLisp-program, jedna funkcija. To se postiže funkcijom progn. Njena sintaksa je:

$$(\text{progn } f_1\ f_2\ \ldots\ f_n),$$

a njeno značenje se dobije evaluacijom svih argumenata $f_1\ f_2\ \ldots\ f_n$ redom, a rezultat je rezultat evaluacije posljednjeg argumenta.

5.12 Ostale funkcije

(length l) daje dužinu liste l.
(atomlist) daje spisak svih atoma u 'listatom'.
(mem) ispisuje stanje memorije.
(cls) briše ekran.

5.13 Posebni atomi

U LLisp-u su uvedena četiri posebna atoma T, NIL, UNDEF, LAMBDA.

- T i NIL predstavljaju logičke vrijednosti 'istina' i 'laž' u LLisp-u.
- UNDEF je atom koji predstavlja vrijednost slobodne varijable, tj. označava da ona nije vezana.
- LAMBDA je atom kojim se započinje definicija λ-funkcije. On je formalizacija simbola λ iz lambda-računa.

Neke od ovih funkcija se obično urade u strojnom kodu, zbog brzine. Jezgro takvih funkcija obično sadrži read, print, eval, cons, apply, car, cdr, set, defun, null, atomp, cond i progn. Ostale funkcije se mogu definirati pomoću baznih funkcija i nalaze se u posebnoj biblioteci.

6 Struktura eksperimentalnog funkcionalnog jezika LLisp

Prema principima rada, programski jezici se dijele u sljedeće grupe:

- proceduralni (Fortran, Pascal, C, ...)[13]
- funkcionalni (Lisp)[14]
- objektno-orjentirani (Smalltalk)
- logički (Prolog)[15]

Osobine *proceduralnih jezika* su:

1. Opis operacija u nekom programu dat je u nekom poretku i poredak izvršavanja operacija isti je s opisanim redosljedom.
2. Izvršavanje programa se definira kao promjena stanja memorije.

Ova druga osobina osim što omogućava da se iz ulaznih podataka dobiju izlazni podaci, mijenjanjem stanje memorije unosi i neke neželjene promjene. Ta pojava nosi naziv *pobočni efekt* ili *side-effect*.

Osobine *logičkog jezika* Prolog su:

1. Unutrašnja struktura se naslanja na pojam relacije i predikatski račun prvog reda.
2. Program se sastoji u opisu problema kroz definicije i osnovne podatke o problemu, a rješavanje se prepušta ugrađenoj 'logici'.

U proceduralnim i funkcionalnim jezicima, operacije se primjenjuju na subjekt, a računanje je predstavljeno u obliku korištenja operacija s nekim objektima. Obratno, u *objektno-orijentiranim jezicima* se objekt operacija uzima za subjekt, primjena operacija na taj objekt se shvaća kao predaja zahtjeva objektu, a interpretaciju zahtjeva i djelovanje izvršava sam objekt, koji se dakle pojavljuje kao subjekt.

U osnovi *funkcionalnog jezika* leži funkcija (ideja λ-funkcije!). Program u ovom jeziku je skup definicija funkcija, a može se predstaviti jednom jedinom funkcijom f. Rezultat programa se shvaća kao rezultat $f(x)$ djelovanja te funkcije na ulazni

[13] ili imperativni
[14] ili aplikativni
[15] ili deklarativni, deskriptivni

podatak x. Pri određivanju vrijednosti funkcije $f(x)$ obično se koriste i druge funkcije ili ta ista funkcija. Npr. ako je

$$f = \lambda x g(h(x), k(x)), \text{ tada je } f(x) = g(h(x), k(x)).$$

Strogo govoreći ovaj jezik nema pobočnog efekta. Također, u strogom funkcionalnom jeziku se garantira jednakost rezultata $f(x)$ bez obzira na redosljed dobijanja unutrašnjih rezultata $h(x)$ i $k(x)$. Sve navedeno ukazuje na mogućnost korištenja funkcionalnog jezika kao jezika za paralelnu obradu podataka.

U funkcionalnom jeziku se koristi nekoliko načina redukcije. Pri normalnom poretku evaluacija se može završiti sa neodređenim vrijednostima parametra, što je nemoguće pri aplikativnom poretku. S druge strane, pri aplikativnom poretku se evaluacija parametara funkcije radi samo samo jednom, a pri normalnom poretku svaki put kada vrijednost parametra zatreba. Zato se, zbog veće efikasnosti, gotovo u svim funkcionalnim jezicima koristi kombinacija:

- Program se piše u funkcionalnom stilu,
- a koristi se aplikativni način evaluacije.

6.1 Uvod

Naš interes nije obrada podataka fiksirane strukture, kao brojeva, matrica i sl., već mogućnost jednostavne obrade najrazličitijih struktura i jednostavno programiranje. Podaci se mogu jednostavno podijeliti u dvije grupe: numeričke i nenumeričke podatke. U naučno-tehničkim disciplinama i komercijalnim djelatnostima, objekti obrade su uglavnom numerički - cijeli i realni brojevi, matrice itd. U oblasti umjetne inteligencije neophodno je obrađivati proizvoljne objekte, kao i relacije među njima. Ti se objekti, kao i relacije među njima obično označavaju slovima, pa se u računalu ne predstavljaju kao numerički, već kao simbolički podaci. Obrada simboličkih podataka se veoma razlikuje od obrade jednostavnih podataka tipa brojeva ili matričnih struktura nad njima. Razlika je u tome što simbolički podaci imaju složenu strukturu, dinamički promjenjivu u toku obrade. Takvu strukturu imaju liste podataka proizvoljnog tipa. Obrada takvih simboličkih objekata se naziva *simbolička obrada*. Jezik LLisp[16] je primjer jezika za simboličku obradu lista i oslanja se na jezik Lisp. U njemu je formiran strogi matematički model na osnovi ranije opisanih struktura podataka i ranije opisanog pojma funkcije kao λ-apstrakcije. Model λ-računa omogućuje da se uz pomoć dovoljno jednostavne sintakse i semantike, predstave sve funkcije funkcionalnog programa [3], [4]. U LLisp-u je primjenjena redukcija normalnog poretka, tj. biran je uvjek prvi lijevi redeks. U većini funkcija korištena je evaluacija aplikativnog poretka, osim u nekoliko funkcija o kojima će kasnije biti govora, a koje koriste 'lijenu' evaluaciju[9]. Funkcionalni stil pisanja programa je zadržan. Također

[16]Zbog Lugić Lisp

je ostvaren ideal funkcionalnog programiranja: "Cijeli program = jedna funkcija". Dakle program u LLisp-u je neka funkcija f. a rezultat programa se shvaća kao rezultat djelovanja te funkcije $(f\ x_1\ x_2\ \ldots\ x_n)$ na ulazne argumente $x_1, x_2, \ldots, x_n$.

6.2 O varijablama

Uvriježeno je mišljenje da se tipovi podataka se ne smiju miješati i da varijable moraju biti deklarirane prije upotrebe. LLisp međutim omogućuje miješanje tipova podataka. Svaka varijabla može biti bilo kog tipa, tj. čuvati podatke bilo kog tipa.
Ono što daju LLispu posebnu specifičnost i draž, a što drugi jezici nemaju, je da sadržaj varijable može biti ime druge varijable ili čak druge funkcije. Time se mogu formirati izvanredno složene strukture podataka (npr. pointerske strukture), a bez razmatranja kako se to stvarno realizira.
Također, svaka je funkcija varijabla, čija je vrijednost definicija funkcije. Prema tome u definiranju i primjeni varijabli i funkcija nema razlika. Tek evaluacija definicije neke funkcije vrši odgovarajuću funkciju.

U LLisp-u je uvedeno jedno ograničenje glede varijabli. Naime, ne dozvoljava se da imena sistemskih funkcija i specijalnih atoma, budu varijable. Ovo dosta ustrožuje sintaksu LLisp-a, mada i suprotno nije teško realizirati.

Nakon što kreiramo alfanumerički atom, zgodno mu je pridružiti neku vrijednost koja bi naglasila da atom još nema vrijednost. U tu svrhu je uveden atom UNDEF. Ovaj atom i atomi NIL, T i LAMBDA su veoma specifični atomi po tome što ne mogu biti varijable i evaluiraju se u same sebe.

6.3 O funkcijama

U LLisp-u nema principijelne razlike između funkcija i ostalih podataka. To je zato što je tijelo neevaluirane funkcije običan $*$-izraz, koji se može obrađivati kao i ostali podaci, a i zato što funkcija može biti podatak.

Međutim u LLisp-u se mogu i definirati funkcije. Ustvari, programiranje u LLispu se prvenstveno sastoji od definiranja novih funkcija. Definiranje funkcije se sastoji u tome da se imenu funkcije pomoću tzv. λ-izraza pridruži definicija funkcije kao njena vrijednost. Dakle ako sami definiramo neku funkciju, njena osnova je λ-izraz. To je primjer realizacije pojma λ-apstrakcije i primjene funkcije u ranije opisanom sistemu λ-računa.

U skladu sa λ-računom, u kome nema funkcija sa imenom, lambda-izraz se prije svega može iskoristiti kao funkcija bez imena. To je zgodno iskoristiti ako je funkcija potrebna samo na jednom mjestu, pa nije potrebno zauzimati memoriju sa nepotrebnim imenima. Kažemo da su to *lokalno definirane funkcije*. Sintaksa lambda-izraza je:

(LAMBDA <lista_argumenata> <tijelo_funkcije1> <tijelo_funkcije2> ...)

Prilikom evaluacije lambda-izraza, evaluiraju se sva tijela redom i vraća rezultat evaluacije posljednjeg tijela. Npr.:

$$((\text{LAMBDA } (x)\ (\text{setq } y\ (+\ x\ x))\ (+\ y\ y))\ 2)$$

će dati rezultat 8.

Ponešto odstupajući od osnovne ideje λ-računa, u LLisp-u su uvedene i funkcije s imenom.

Tako sa

$$(\text{defun } <\text{alfanumerik}> <\lambda\text{-izraz}>),$$

λ-izraz dobija ime alfanumerika.

Npr. sa

$$(\text{setq } f\ '(\text{LAMBDA } (x)\ (*\ x\ x))),$$

varijabla f ima za vrijednost λ-apstrakciju (λ-izraz) (LAMBDA (x) $(*\ x\ x)$). Zbog naše navike da razlikujemo varijable i funkcije, uvedena je jednostavnija sintaksa:

(defun <alfanumerik> <lista_arg> <tijelo_funkcije1> <tijelo_funkcije2> ...),

a rezultat je opet isti: funkcija imena <alfanumerik> sa λ-apstrakcijom kao definicijom (vrijednošću).

Ranije pomenuta funkcije f se sada može definirati i sa:

$$(\text{defun } f\ (x)\ (*\ x\ x)).$$

Funkcija može biti i bez argumenata. Npr. (defun nista () (print)).

Otkucamo li *nista*, dobit ćemo vrijednost atoma *nista*, tj. (LAMBDA () (print)).

Otkucamo li (*nista*) dobit ćemo evaluiranu vrijednost od *nista* , tj. prazan red.

6.4 Pravila evaluacije

LLisp interpreter vrši računanje $*$-terma.

1. Ako je ulaz LLisp interpretera atom, on se odmah evaluira.
2. Ako je ulaz tačkasti par tada, da bi se mogao evaluirati, mora biti lista posebnog oblika.
3. Ako ništa od prethodnog nije moguće, LLisp prijavljuje grešku.

Ovaj proces redukcije i izračunavanja vrijednosti $*$-terma se naziva *evaluacija*. Rezultat evaluacije se zove *vrijednost $*$-terma*.

6.4.1 Evaluacija atoma

Numerički atomi atomi se evaluiraju u svoju vrijednost.
Alfanumerički atomi se mogu iskoristiti za definiranje varijabli. Predviđeno je da samo alfanumerički atomi koji počinju slovom mogu biti varijable. Alfanumerički atom postaje varijabla, ako mu se pridruži neka vrijednost. Tako varijabla x dobija vrijednost y naredbom (setq x y) pri čemu vrijednost y može biti bilo koji $*$-term.
Ako je alfanumerički atom varijabla, vrijednost pridružena tom atomu je rezultat evaluacije.

6.4.2 Evaluacija tačkastog para

Da bi se tačkasti par mogao evaluirati, potrebno je da bude lista određene strukture:

(<ime_funkcije> <argument1> <argument2> ...)

Prvi član liste može biti ime_funkcije ili definicija_funkcije koja se treba primjeniti nad argumentima funkcije.
Argumenti funkcije (kojih može biti i 0) su $*$-termi nad kojima se funkcija treba izvršiti. Kada LLisp evaluira neku takvu listu, on prvo evaluira sve njezine argumente, pa zatim primjenjuje funkciju na te argumente. Npr. evaluacija izraza (car (a b c)) teče ovako:

1. Prvo se evaluira prvi član liste, *car*.
2. Zatim se pokuša evaluirati argument funkcije *car*. Kako je argument (a b c) lista, a se tumači kao funkcija sa dva argumenta b i c.
3. Kako a nije ni sistemska ni korisnička funkcija, prijavljuje se greška.

Dakle, rad LLisp interpretera je jedna stalna evaluacija. Većina LLisp funkcija odmah evaluira sve svoje argumente. Međutim neke funkcije odstupaju od tog pravila i imaju specijalan tretman prilikom evaluacije.
Npr.:

- *quote* spriječava evaluaciju svog argumenta
- *setq* smije evaluirati samo drugi argument
- *defun* ne evaluira nijedan svoj argument
- *cond* ima argumente koji se ne mogu evaluirati
- *eval* izvršava ekstra evaluaciju tamo gdje nije predviđena
- *if*, *and*, *or* evaluiraju argumente sa zadrškom ('lijena' evaluacija!)

Npr. (quote (+ 1 2)) ne dozvoljava evaluaciju izraza (+ 1 2).
Sa (setq x 'y) i (setq y 'z) se nakon x dobije rezultat y, a nakon (eval x) rezultat z.

Korištenje 'lijene' evaluacije je specifičnost LLisp-a. Zahvaljujući 'lijenoj' evaluaciji, većina funkcija u LLisp-u nema pobočnog efekta. Tako npr. (or 1 x) će dati 1, iako varijabla x nije vezana. Slično se u (if (atomp 2) 2 (car 2)) neće pojaviti poruka o greški, jer se (car 2) neće evaluirati.

7 Unutrašnja realizacija LLisp struktura

Osnovni element za izgradnju LLisp struktura su alfanumerički atomi. Ovdje je urađena jednostavnija realizacija atoma koji ima:

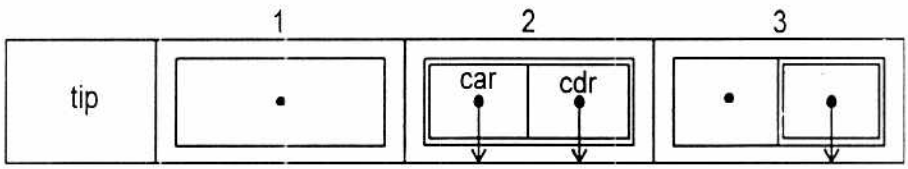
- ime,
- vrijednost.

Druga osnovna LLisp struktura je tačkasti par. Ideja za njenu realizaciju opisana je ranije.
Nakon opisanih struktura podataka, problem je kako ih realizirati na datom računalu. Zajedničko svim podacima je da je adresa na kojoj je on smješten u memoriji računala pointer. Zato je C-struktura kojom su realizirane LLisp strukture zasnovana na pointerima umjesto na podacima. C-struktura koja zadovoljava sve potrebne zahtjeve formirana je na sljedeći način:

```
typedef CVOR *drvo;
typedef struct cvor
  {
  int tip;
  union
    {
    char *Atom;
    struct
      {
      struct cvor *Lrep;
      struct cvor *Drep;
      } rep;
    struct
      {
      char *Ime;
      struct cvor **Pointer;
      } id;
     } ukupno;
    } CVOR;
```

Ovim je deklarirana pointerska struktura *drvo*, kao pointer na strukturu CVOR koja podržava sve zahtjeve koji su potrebni za realizaciju bilo koje LLisp strukture.

Navedenom deklaracijom određen je samo oblik strukture *drvo*, odnosno CVOR, dok se potrebni memorijski prostor rezervira tek sa definicijom strukturne varijable. Dakle, strukturom *drvo* formiran je samo pointer na strukturu CVOR, pri čemu ćelija koja u sebi nosi strukturu CVOR ima sljedeći oblik:

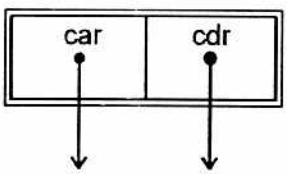


Slika 33: LLisp struktura tačkastog para

Dakle, predviđeno je da struktura *drvo* može imati tri tipa.
tip=1
Ovom strukturom se predstavlja konstantni atom, a zauzima 16 byta (koliko se rezervira za jednu riječ).

Slika 34: Ćelija atoma

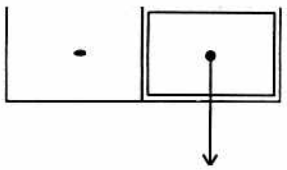
tip=2
U ovom slučaju imamo pointerski par (*lrep*| *drep*) koji reprezentira tačasti par. To radi tako što prvi pointer 'lrep' pokazuje na prvi član para, a drugi pointer 'drep' na drugi član para. U slučaju nepostojanja pointera, u odgovarajući dio ćelije smješta se poseban atom NIL koji to naznačuje. Ćeliju tipa 2, zvaćemo *car-cdr ćelija.*



Slika 35: Ćelija tačkastog para

Ova struktura zauzima 32 byta, jer se za svaki pointer rezervira po 16 byta.

tip=3
Ovakvom strukturom se realizira atom koji može biti varijabla. Prvi pointer pokazuje na ime atoma, a drugi pokazuje na vrijednost atoma. Kako je vrijednost deklarirana kao pointer na pointer, jasno je da je ovim predviđeno da vrijednost atoma može biti proizvoljna LLisp struktura.



Slika 36: Ćelija varijable

Ova struktura zauzima 32 byta (16 byta za ime varijable i 16 byta za pointer na sadržaj varijable).

8 Algoritmi jezika LLisp

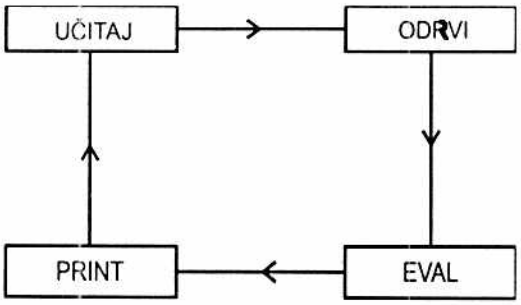
Sad kad smo programski realizirali strukture podataka, problem je kako što efektivnije (s obzirom na raspoloživo vrijeme i memoriju) formirati algoritme za njihovu obradu. Obrada tih podataka se može realizirati *paralelno* ili *sekvencijalno*. Mi ćemo obratiti pažnju samo na tradicionalni, sekvencijalni, način obrade podataka, koji se koristi u tzv. *Von Neumann-ovski baziranim računalima* i odgovarajuće algoritme. Proces obrade $*$-terma u LLisp-u, svodi se na nekoliko osnovnih algoritama:

- Upis ulaznog izraza (**učitaj**)
- Prevođenje tog izraza (**odrvi**)
- Evaluacija izraza (**eval**)
- Ispis dobijenog rezultata (**print**)

Shema tog procesa je prikazana na slici 37.

8.1 Algoritam sintaksnog analizatora

Ulaz za LLisp interpreter je $*$-term. Svaki takav izraz je neki niz karaktera. LLisp mora prepoznati taj izraz i izgraditi njegovu odgovarajuću unutrašnju reprezentaciju pomoću binarnog drva i proslijediti ga na evaluaciju LLisp interpreteru da bi on nad njim mogao izvršiti dalje operacije. Teret ovog prevođenja je na funkciji *upis*. Kada *upis* obavi ovaj posao, ostale LLisp funkcije operiraju samo sa unutrašnjim



Slika 37: Algoritam rada LLisp-a

reprezentacijama koje je *upis* uradio. Dakle LLisp funkcije nemaju nikakvog kontakta sa $*$-termom kao nizom znakova.

Funkcija *upis* ima dvije podfunkcije *ucitaj* i *odrvi* i to obavlja sljedećim algoritmom:

{

ucitaj term;

odrvi term;

}

1. Funkcija *ucitaj* čita ulazni niz karaktera i prepoznaje da li je sintaksno ispravan.
2. Nakon što se odredi vrsta $*$-terma, funkcija *odrvi* pravi njegovu unutrašnju realizaciju.

Dakle *upis* čini mnogo posla kojeg čini kompajler u većini običnih kompjuterskih jezika. Naime i tamo postoji program koji tretira niz znakova i pretvara u određenu unutrašnju strukturu. U slučaju kompajlera ta je struktura drvo kao osnova za generiranje koda, a u LLispu je binarno drvo kao osnova za evaluaciju. Dakle *upis* u LLispu nije isto što i 'input' u drugim jezicima. 'Input' samo konvertira ulazni string u broj, a *upis* učitava string i radi njegovu kompletnu sintaksnu analizu. Kako je sintaksa LLispa vrlo prosta, ovaj se zadatak može izvršiti vrlo efikasno.
Kada se LLisp programom traži ulaz nekog niza karaktera, tada je LLisp funkcija *read* vanjska reprezentacija funkcije *upis*.
Algoritam funkcije **ucitaj**:

{

do

{

čitaj ulaznu nisku znakova dok ne naiđeš na znak prelaska u novi red;

prebroj otvorene i zatvorene zagrade;

pretvori sva velika slova u mala;

if učitani znak nije dozvoljen, prijavi grešku; %funkcija 'dozvoljen'

}

while zagrade nisu 'uparene';

Izvrši sintaksnu analizu upisanog izraza;

if izraz je sintaksno neispravan, prijavi grešku; %funkcija 'analizator'

}

8.1.1 Leksički i sintaksni analizator

Tokom upisa izraza, analizira se svaki učitani znak. To radi funkcija *dozvoljen*. U slučaju učitavanja znaka (simbola) koji nije iz definiranog alfabeta, prijavi se odgovarajuća greška.
Nakon leksičke analize, provjerava se sintaksa izraza. To radi funkcija *analizator*.

8.2 Algoritam pravljenja drva izraza

Nakon provjere sintaksne ispravnosti izraza, potrebno je, u svrhu dalje obrade, formirati unutrašnju strukturu ulaznog $*$-terma. To obavlja funkcija *odrvi*. Algoritam je sljedeći:

1. Tako obrađen niz karaktera se proslijeđuje funkciji *otkid* koja određuje koje je vrste ulazni $*$-term i raščlanjuje ga na sintaksne dijelove:

   - Ako je atom, on je i ujedno i njegov elementarni rastav.
   - Ako je tačkasti par, prepoznaje ugnježdene $*$-terme i rastavlja tačkasti par na njegove sastavne dijelove. Rezultat tog rastava su prvi i drugi član para (ono što zovemo *car* i *cdr* para).

2. Prema tome koji tip $*$-terma je 'otkid' prepoznao, funkcija *odrvi* ima dvije podfunkcije:

   - *drvo_atoma*
     - *drvce*
     - *drvo_var*

- *drvo_tpar*

3. Ako je *otkid* prepoznao atom, tada se funkcijom *drvo_atoma* formira odgovarajuća struktura. I to funkcijom *drvce* struktura za konstantu, a funkcijom *drvo_var* za varijablu.

4. Ako je $*$-term tačkasti par, tada se na rekurzivan način funkcijom *drvo_tpara* prevodi tako raščlanjena struktura u odgovarajuće binarno drvo.

Algoritam funkcije **odrvi**:

{

if učitani term x je atom **then** načini drvo_atoma(x);

else if učitani term je tačkasti par **then** načini drvo_tpara(x);

else prijavi grešku;

}

Algoritam funkcije **drvo_atoma**:

{

if učitani atom x je broj, binarna relacija, operacija, atomlist, T, LAMBDA, NIL, UNDEF **then** načini drvce(x);

if učitani atom x je u 'kesi' **then** vrati njegovu reprezentaciju iz 'kese' kao rezultat;

else if učitani atom x je u 'listatom' **then** vrati njegovu reprezentaciju iz 'listatom' kao rezultat;

else

{

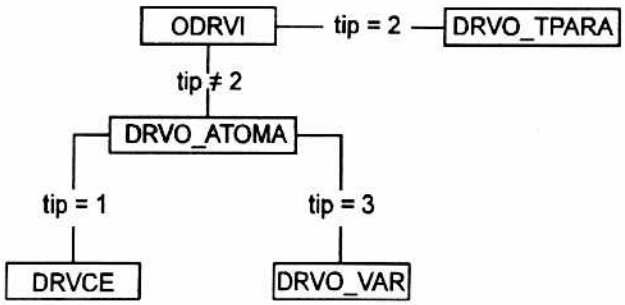
if kljucna_rijec(x) **then** načini drvce(x);

else načini drvo_var(x);

}

Dobivenu reprezentaciju atoma dodaj u listu svih atoma 'listatom';

}



Slika 38: Funkcija *odrvi* i njene podfunkcije

8.2.1 Algoritam formiranja i ažuriranja liste objekata

Istovremeno dok funkcija *odrvi* formira unutrašnju reprezentaciju ulaznog $*$-terma, formira se posebna lista *listatom*, na kojoj se nalaze svi upotrebljeni atomi. Algoritam je sljedeći:

1. Svaki put kad se *drvo_atoma* susretne sa alfanumeričkim atomom, ona traži atom tog imena u *listatom*.
2. Ako je on tamo, uzeti će njegovu egzistirajuću reprezentaciju, kao potrebnu i novu neće formirati.
3. U suprotnom će formirati novu reprezentaciju i staviti je na 'listatom'.
4. Ako atom nije alfanumerički, formira se reprezentacija tog atoma, ali se ona ne stavlja u listu.

Preciznije, algoritam **formiranja i ažuriranja liste objekata** je sljedeći:

{

if učitani term x je atom **then**

{

if x je broj, binarna relacija ili operacija, atomlist, T, NIL, UNDEF, LAMBDA, **then** formiraj njihovo drvo i ne stavi ga na listu atoma 'listatom';

if atom x je u listi lokalnih varijabli 'kesa' **then** uzmi drvo iz 'kese' kao tekuću reprezentaciju atoma;

else if atom x je u listi globalnih atoma 'listatom' **then** uzmi drvo iz 'listatom' kao tekuću reprezentaciju;

else

{

if x je ključna riječ **then** napravi drvo konstante i stavi ga u 'listatom';

else napravi drvo varijable i stavi ga u 'listatom';

}

}

}

Dakle, u toj listi će se pojaviti svi alfanumerički atomi:

- interni: imena sistemskih funkcija *car*, *list* i sl.
- eksterni: imena novih, od korisnika formiranih, konstanti, varijabli, funkcija.

Prema tome, u 'listatom' su sačuvane informacije o svim upotrebljenim alfanumeričkim atomima, varijablama i funkcijama. Zato pri formiranju drva nekog $*$-terma isti atomi imaju potpuno iste reprezentacije, tj. oni su jedinstveni. Dakle svi pozivi istog atoma pokazuju na istu memorijsku lokaciju.
Ova lista je obično dosta duga i može se štampati naredbom (atomlist) koja nema argumenata ili sa (length (atomlist)) dobiti dužinu te liste. Međutim sama lista je nedostupna za korisničku obradu.

8.3 Algoritam evaluacije

Postoje dva pristupa evaluaciji $*$-terma. Jedan je pristup rekurzivan i naziva se *nanižni*, a drugi je zasnovan na steku i naziva se *navišni*. U ovom slučaju je obrađen samo navišni pristup[17] i na njemu je i zasnovana evaluaciju u jeziku LLisp. U LLisp-u je primjenjena evaluacija aplikativnog poretka, osim za nekoliko funkcija, za koje se koristi 'lijena' evaluacija.

Osnovna funkcija u LLispu je *_eval*. Ona vrši evaluaciju LLisp izraza, ali ne prevodi program u strojni kod, pa je rad funkcije '*_eval*' rad interpretera. Algoritam evaluacije $*$-terma s je sljedeći:

{

if s je atom

{

if s je konstanta **then** s vrati kao rezultat,

[17] Jer je mnogo efikasniji

else potraži s u listi svih atoma *listatom*;

if s je u *listatom*, vrati njenu vrijednost kao rezultat

or if vrijednost je *undef* vrati grešku;

if s nije u *listatom* prijavi grešku

}

else if s je lista $s = (s_1\ s_2\ \ldots\ s_n)$

{

evaluiraj s_1 i neka je f rezultat te evaluacije

if (f je atom **and** f je funkcija) **then** evaluiraj argumente funkcije

else if (f nije atom **and** f je λ-funkcija) **then** evaluiraj argumente funkcije

else prijavi grešku

if broj formalnih i stvarnih argumenata se ne podudara **then** prijavi grešku

else primjeni funkciju f na njene argumentima

}

else prijavi grešku

}

8.3.1 Algoritam supstitucije

U LLisp-u se korištenjem funkcije etqsvakom objektu može dodijeliti neko ime. Isto ime se može dodijeliti i nekoj drugoj vrijednosti, dakle ime je varijabla. Nakon što kreiramo reprezentaciju alfanumeričkog atoma koji može biti varijabla, on ima posebnu vrijednost UNDEF. UNDEF je vrlo specifičan atom, jer ne može biti varijabla i služi samo da označi činjenicu da atom kome je UNDEF vrijednost još nije vezan.
Nakon (setq x y), algoritam je sljedeći:

{

if x nije alfanumerik **then** prijavi grešku;

else evaluiraj drugi argument y u vrijednost z;

if alfanumerik x je u listi lokalnih varijabli 'kesa' **then** zamjeni vrijednost x-a iz 'kese' sa novom vrijednošću z;

else if alfanumerik x je u listi globalnih varijabli 'listatom' **then** zamjeni vrijednost x-a iz 'listatom' sa novom vrijednošću z;

}

8.3.2 Algoritam steka

U evaluaciji izraza navišnim načinom (koji je ovdje primjenjen) nužna je upotreba steka. Ideja steka je dvostruka:

- Da se na njemu čuvaju lokalne varijable funkcija do trenutka kada nam ne budu potrebne.
- Da nam omogući pamćenje puta kojeg je potrebno prijeći po drvu izraza, dok vršimo njegovu redukciju i evaluaciju.

Da bi se odgovorilo ovim zahtjevima, koriste se dva steka:

1. Stek upravljanja
2. Stek podataka

U evaluaciji sistemskih funkcija, nije nam potreban stek podataka, jer su sve varijable globalne i čuvaju se u 'listatom'.
U u evaluaciji korisnički definiranih funkcija, potreban nam je stek podataka (ako se stek upravljanja ne koristi u tu svrhu).

Stek upravljanja je realiziran kao dinamička struktura jednostruko-povezane liste tipa *drvored*. Na njemu se čuvaju redeksi i povratne adrese. Povratna adresa je pointer koji pokazuje mjesto s kojeg se nastavlja redukcija drva izraza nakon poziva određene funkcije (evaluacije određenog redeksa).

```
typedef struct Clan
  {
  drvo vrh;
  struct Clan *rep;
  } CLAN;

typedef CLAN *drvored;
drvored stek;
```

Članovi liste su unutrašnje realizacije $*$-terma. Pointer na takvu strukturu se naziva *drvored*, jer su članovi te liste tipa *drvo*.

```
drvo stek_kesa[64];
```

Stek podataka je realiziran kao statička struktura niza pointera tipa *drvo*. Razlog odstupanja od dinamičke strukture je procjena da se na steku neće lako naći međusobni poziv više od 64 funkcije istovremeno.

Varijable definirane unutar funkcije nazivaju se *lokalne varijable* jer su nedostupne ostalim funkcijama.[18] U trenutku evaluacije smještaju se na stek podataka i sve njihove izmjene rade se na steku.
Istovremeno, globalne varijable su smještene u globalnoj listi *listatom*, tako da su dostupne iz bilo kog dijela programa.
Nakon poziva određene funkcije, koristeći se povratnom adresom, vraćamo se u nadređenu funkciju, a ranije zauzeti dio stoga oslobađa. To je razlog zašto lokalne varijable nisu dostupne ostalim funkcijama i zašto se gube izlaskom iz funkcije.

Pri redukciji i evaluaciji izraza, primjenjena je redukcija normalnog poretka, tj. drvo izraza se obilazi na 'preorder' način i traži prvi lijevi redeks.
Algoritmom evaluacije na stek je stavljeno drvo izraza koji se evaluira. To drvo je trenutno jedini element steka, tj. ono je 'vrh steka'. Nad njim se primjenjuje redukcija, koristeći se algoritmom steka, koji je sljedeći:

- Obilazi se drvo izraza od korjena prema listovima.
- Čim se nađe prvi redeks, stavi se na stek povratna adresa i zatim sam redeks.
- Opisani postupak se nastavlja sve dok se ne nađe redeks normalne forme.
- Kad se takav redeks nađe, on je vrh steka i pristupa se njegovoj evaluaciji, primjenjujući algoritam funkcije.
  - U slučaju da je na steku samo redeks normalne forme, on se evaluira i nakon evaluacije skine sa steka, a vrati njegov rezultat. To je i trenutak završetka evaluacije na steku.
  - U slučaju da na steku ima još elemenata, evaluira se vršni redeks i nakon njegove evaluacije, primjeni se metod 'općeg grafa' (vidi str. 21). Naime, u drvu povratne adrese lijevi rep je redeks (po algoritmu redukcije) sa vrha steka, pa se zamjeni drvom njegovog rezultata. Kažemo da se pointeri 'prespoje'.[19]
  - Ova zamjena je ustvari reducirala graf izraza. Zato je element na vrhu steka nepotreban i skine sa sa steka, a također i povratna adresa.
  - Redukcija se nastavi na drvu sa vrha steka.

[18] Npr. (progn (setq x 2) ((lambda (x) (setq x x)) 4)) će dati rezultat 4, ali će ostati $x = 2$. Naime, x u tijelu funkcije je lokalna i ima samo trenutnu vrijednost $x = 4$.

[19] Drugi način je da se umjesto zamjene pointera $p = q$, zamjene njihove vrijednosti $*p = *q$. Ovo je tzv. *destruktivni* način zamjene

Detaljnije, algoritam steka je:

{

if term x je atom ili lambda_funkcija **then** evaluiraj ih i ne stavi na stek;

if lijeva grana tačkastog para je atom i nije ključna riječ **then** prijavi grešku;

else stavi na stek drvo terma x.

Odredi povratnu adresu i redeks (*pov_adr* i *redeks*) drva izraza x;

while stek nije prazan

{

if ima još redeksa na drvu terma x

{

stavi na stek njegovu povratnu adresu;

stavi na stek taj redeks;

odredi novu povratnu adresu i redeks;

}

else evaluiraj vrh steka;

if dostignuto je dno steka

{

obriši vrh steka;

vrati rezultat evaluacije;

}

else zamjeni evaluiranu granu drva s rezultatom evaluacije;

}

else

{

obriši vrh steka;

obriši novi vrh steka;

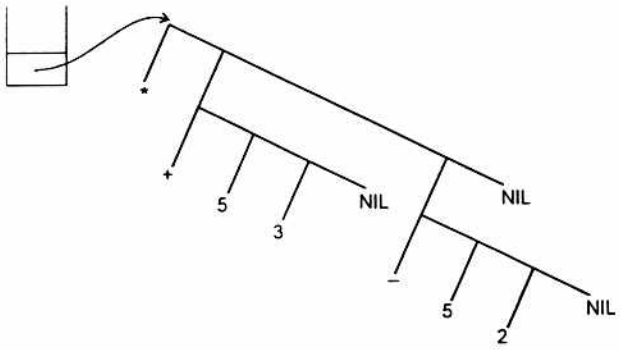
}

}

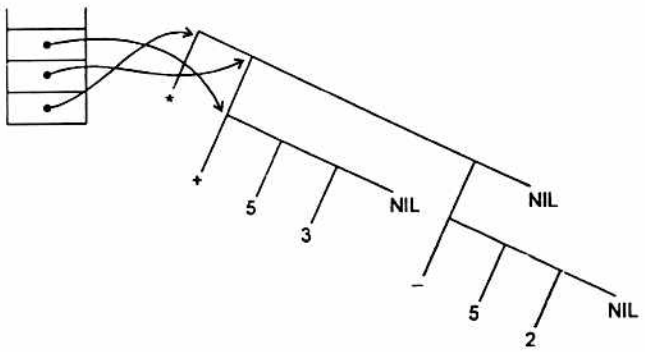
Primjer:

Razmotrimo redukciju (evaluaciju) na jednostavnom primjeru (* (+ 5 3) (- 5 2)).

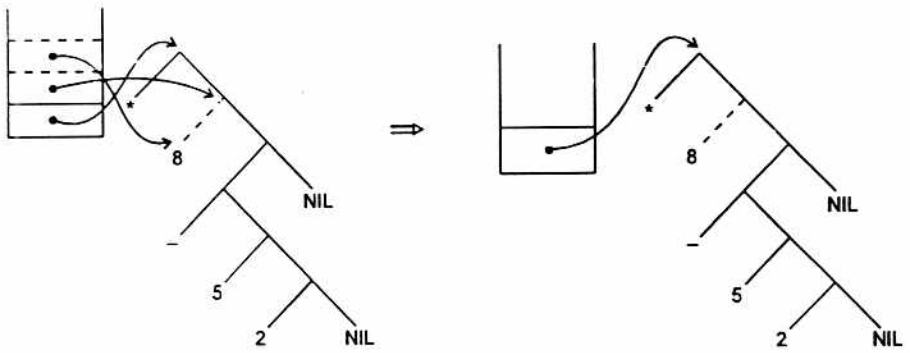
1. Na samom početku evaluacije, vrh steka sadrži pointer na korjenski vrh grafa.



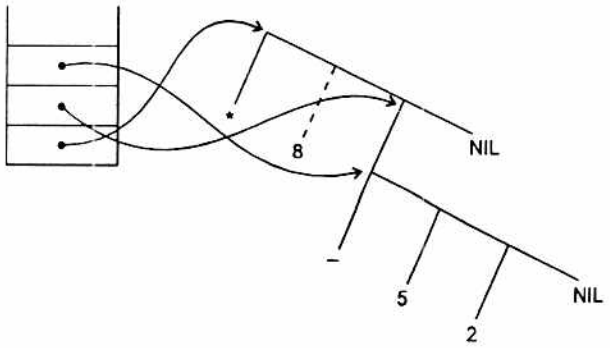
2. Postupak redukcije kreće od vršnog elementa steka. Odmah se gleda desna grana i na stek stavi pointer na ono desno poddrvo koje ima redeks u lijevoj grani, a zatim na stek stavi i pointer na taj redeks.



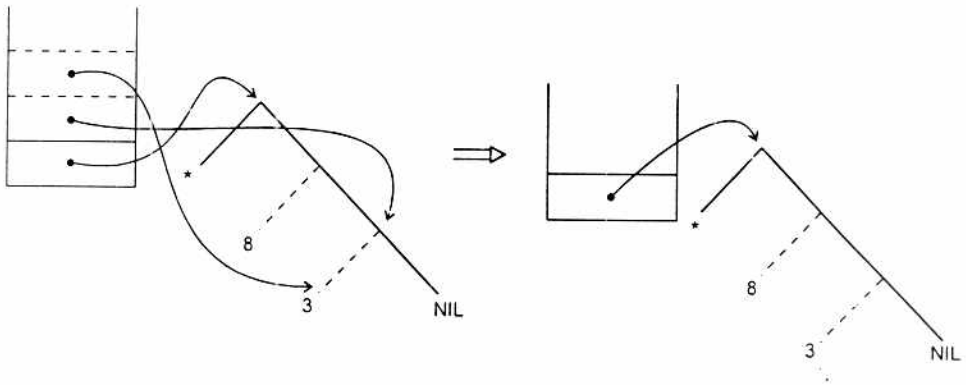
3. Postupak redukcije se pokušava opet na vršnom elementu steka. Međutim, ovdje nema redeksa, pa se pristupa evaluaciji vršnog elementa. Nakon evaluacije izraza (+ 5 3) u 8, pointer na izraz (+ 5 3) se zamjeni sa pointerom na izraz 8 i oba pointera skinu sa steka.



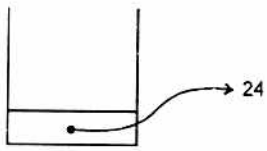
4. Redukcija se nastavi opet sa elementom na vrhu steka. Stek je sada sljedeći:



5. Nakon evaluacije vrha steka i skidanja pointera sa steka, imamo na vrhu steka samo pointer na drvo izraza $(* \ 8 \ 3)$.



6. Vršni element $(* \ 8 \ 3)$ steka je normalna forma, pa se može izvršiti redukcija.



Tako se dobije rezultat 24.

U radu sa stekom postoji još jedan problem. To je funkcija *eval*. Ova funkcija zahtjeva dodatnu evaluaciju izraza. Zato se redeks u kome se ona pojavljuje mora opet podvrgnuti algoritmu steka, tj. ako je u toku evaluacija na steku, ponašati se kao da taj postupak tek započinje. To je omogućeno korištenjem pojma *dna*. U trenutku svakog pristupa steku funkcijom *eval*, vrh steka se proglasi za dno steka i na njemu radi dalje kao da je bio prazan. Ako se u pokušaju silaska sa steka dođe do dna, a ispod dna ima još elemenata steka, postupak redukcije se nastavlja, a inače završi.

8.3.3 Algoritam funkcije

Nakon evaluacije izraza na steku, dolazi se do trenutka kada se poziva funkcija eval koja treba evaluirati vrh steka.

Algoritam funkcije ima dvije grane:

- ako je funkcija sistemska,

- ako je funkcija λ-apstrakcija.

Ako treba primjeniti sistemsku funkciju na argumente, algoritam je jednostavan:

- Prvo se pogleda da li je broj zadanih argumenata funkcije onakav kako je definirano semantikom funkcije. Ako nije javi se greška i prekine evaluacija, inače se nastavi dalje
- Ako je sistemska funkcija *quote*, *setq*, *defun*, *cond*, *if*, *and* i *or*, argumenti se evaluiraju sa zadrškom[20], onako kako je to definirano u semantici jezika.
- Inače se odmah evaluiraju svi argumenti funkcije.[21]
- Nakon toga se odgovarajuća funkcija primjeni nad danim argumentima.

Pitanje primjene funkcije λ-apstrakcija na njene argumente, riješeno je sljedećim algoritmom:

- Primjeniti λ-apstrakciju na njene argumente, znači evaluirati njeno tijelo
- Da bi se tijelo evaluiralo, formalnim argumentim funkcije se moraju respektivno pridružiti stvarni argumenti.
- Prvo se stvarni argumenti evaluiraju[22], a zatim se za svaku lokalnu varijablu napravi kopija i njoj se pridruži stvarna vrijednost argumenta. Ovaj proces se naziva *ukešenje* i obavlja ga funkcija *ukesenje*.
- Kako formalni argumenti nisu globalne varijable, smještamo ih u posebnu listu, koju nazivamo *kesa*. Kesa ima ulogu male liste 'listatom', ali su u njoj jedino varijable odgovarajuće funkcije. Gdje god ide funkcija, sa njom ide i njena 'kesa'.
- Kako nakon ukešenja može doći do poziva druge funkcije, to se evaluacija tijela prve funkcije odgađa, pa se kesa za to vrijeme mora negdje sačuvati. Kesa se zato stavi na stek podataka i tamo čuva do poziva njene funkcije.
- Ako nema poziva nove funkcije, pristupa se evaluaciji tijela funkcije.
- Ako tokom evaluacije tijela dođe do zahtjeva za vrijednošću atoma, prvo se pogleda da li je atom u dostupnoj kesi, a ako nije, onda u 'listatom'. U zavisnosti od izabranog atoma, vrati se njegova vrijednost.

[20] 'Lijena' evaluacija

[21] Evaluacija aplikativnog poretka ili energična evaluacija

[22] Način da se prvo formalnim argumentima pridruže stvarni argumenti. bio bi neefikasan, jer bi se pritom morala svaka kopija formalnog argumenta izračunavati više puta. Vidi 1.4.3

- Nakon evaluacije, 'kesa' se skine sa steka i izbriše, jer je kesa lokalnih varijabli privremenog karaktera.

- Rezultat evaluacije tijela se vrati kao rezultat primjene λ-apstrakcije. Ako funkcija ima više tijela, rezultat evaluacije funkcije je rezultat evaluacije posljednjeg tijela.

- Ako je tokom evaluacije došlo do prekida zbog prijave greške, kese sa steka se izbrišu prije završetka evaluacije.

Dakle, jedino što ostaje je, prije evaluacije odrediti da li je prvi element izraza sistemska funkcija ili λ-apstrakcija i prema tome odabrati put u algoritmu.

Algoritam funkcije je dakle:

{

if funkcija f je sistemska

{

if broj argumenata funkcije funkcije f nije sintaksno ispravan **then** prijavi grešku;

if funkcija f je quote, setq, set, defun, and, or, cond, if **then** primjeni evaluaciju sa zadrškom;

else primjeni evaluaciju aplikativnog poretka.

Primjeni odgovarajuće algoritme za odgovarajuće funkcije;

}

else if f je λ-apstrakcija

{

Napravi kopiju formalnih argumenata;

Kopijama formalnih argumenata pridruži stvarne argumente;

Tako formirane formalne argumente stavi u listu lokalnih varijabli 'kesa';

if 'kesa' nije prazna **then** 'kesu' stavi na stek podataka;

else za 'kesu' uzmi trenutačni vrh steka podataka.

Napravi kopiju tijela λ-apstrakcije;

Evaluiraj sva tijela funkcije;

Nakon evaluacije skini kesu sa steka podataka;

}

}

Primjer 1.
Zadana je funkcija

$$(\text{if } (\text{not } (\text{zerop } x)) \ (+ \ x \ (- \ x \ 5)) \ y),$$

pri čemu je $x = 6$, a y je slobodna varijabla.

1. Kako je prvi element liste sistemska funkcija *if*, provjeri se da li je broj stvarnih argumenata funkcije 3 i kako jest. nastavlja dalje.
2. Poziva se odgovarajuća funkcija koja evaluira izraz prema definiciji funkcije *if*.
   (a) Prvo se evaluira prvi argument.
   (b) Kako je rezultat T, evaluira se drugi argument i dobije se 7.
3. Prema semantici naredbe, evaluacija je završena i rezultat funkcije je 7. Do prijave greške za slobodnu varijablu y dakle nije moglo ni doći.

Primjer 2.
Uzmimo sada primjer primjene λ-funkcije na argumente u slučaju

$$((\text{lambda } (x \ y) \ (\text{cons } x \ y)) \ 1 \ 2).$$

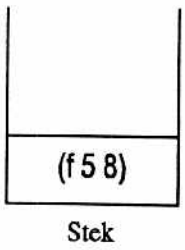
1. Očigledno je da ovaj izraz nije atom. Njegov prvi element je λ-funkcija

   $$(\text{lambda } (x \ y) \ (\text{cons } x \ y)).$$

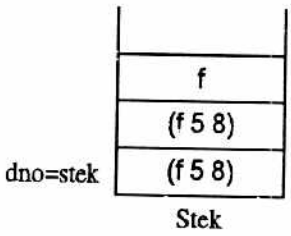
   Primjeniti ovu funkciju na njene argumente, znači evaluirati njeno tijelo $(\text{cons } x \ y)$.
2. Prije toga se kopijama formalnih argumenata (varijablama) x i y pridruže stvarni argumenti 1 i 2.
3. Od njih se formira kesa i stavi na stek podataka.
4. Pristupi se evaluaciji tijela $(\text{cons } x \ y)$. Varijable x i y se prvo potraže u tekućoj kesi i kako se tamo nađu, odatle se uzmu vrijednosti argumenata funkcije.
5. Nakon zamjene argumenata x i y sa 1 i 2. prilazi se evaluaciji tijela (cons 1 2) i vraća rezultat (1 2).

Primjer 3.
Sada razmotrimo primjer redukcije na izrazu $(f \ 5 \ 8)$, gdje je $f \equiv (\lambda x \lambda y \, (+ \ x \ y))$.

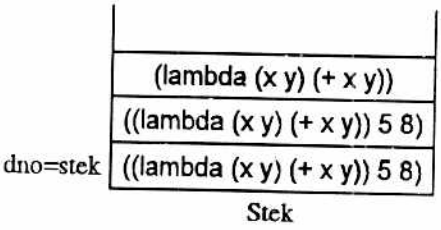
1. Na samom početku evaluacije, na stek se stavi redeks (pointer na redeks) $(f\ 5\ 8)$.



2. Kako se varijabla također može reducirati, na stek zatim stavljamo pointer na drvo izraza $(f\ 5\ 8)$ (jer je redeks u njegovoj lijevoj grani) i pointer na taj redeks, tj. pointer na varijablu f.

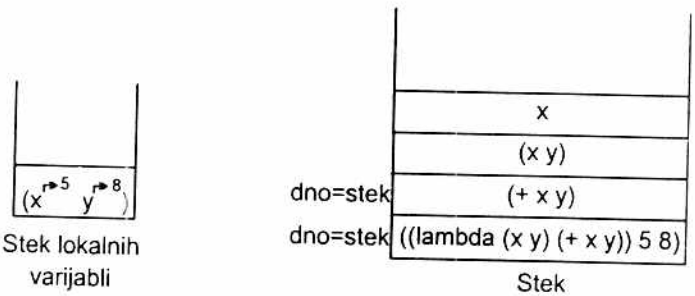


3. Vrh f se evaluira u svoju vrijednost $(\lambda x \lambda y(+\ x\ y))$. Zatim se pointer na drvo od f zamjeni sa pointerom na vrijednost od f, tj. pointerom na drvo od $(\lambda x \lambda y(+\ x\ y))$, a nakon toga oba pointera skinu sa steka.

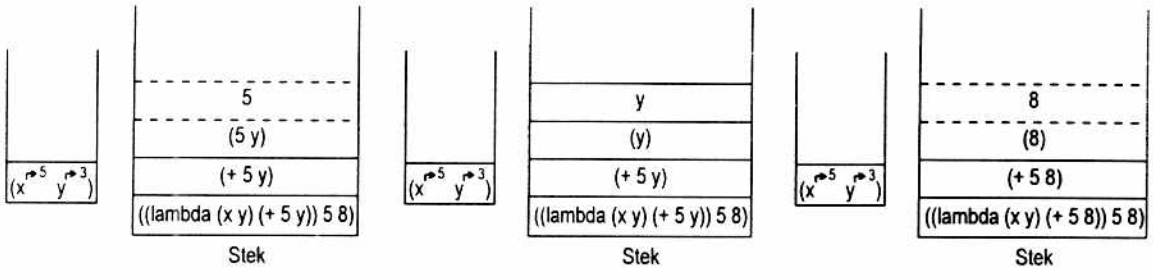


4. Redukcija se nastavlja dalje po desnoj grani, međutim ne nailazi na nove redekse. Zato se pristupa evaluaciji. Evaluacija tog izraza uključuje sljedeće:

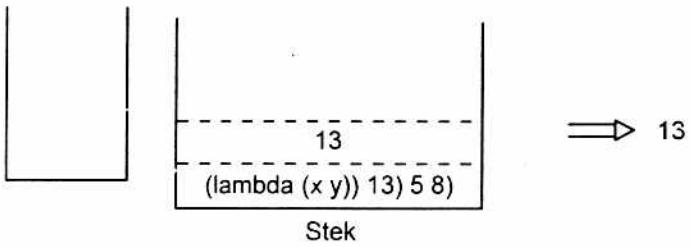
   (a) Kopijama formalnih argumenata se pridruže stvarni argumenti, a zatim te kopije (kesa!) stave na stek lokalnih varijabli.



(b) Nad tijelom funkcije izvrši se redukcija i evaluacija.



c) Nakon korištenja argumenata oni se skinu sa steka lokalnih varijabli, a rezultat evaluacije vrati kao rezultat jer je dostignuto dno steka.



8.3.4 Algoritam rekurzije

Pojam rekurzije

Pod rekurzijom se shvaća proces kada funkcija poziva samu sebe neposredno ili posredstvom neke druge funkcije. Njena osnova je u pojmu matematičke indukcije. Postupak programiranja zasnovan na rekurzivnim funkcijama zove se *rekurzivni* postupak, a zasnovan na raznim programskim petljama *iterativni* postupak. U LLisp nije ugrađen mehanizam za rad sa petljama, pa je rekurzivni postupak u LLispu osnovni postupak rada sa funkcijama. U odnosu na iterativni postupak, rekurzivni postupak je prirodan i jednostavan, međutim izrazito je neefikasan. Ali se zato neki programski zadaci, kao što je obilazak dinamičkih struktura podataka, ne mogu riješiti bez rekurzije.

Međutim, problem sa rekurzijom je u tome što ona generira čitav niz varijabli istog imena, od kojih svaka vrijedi samo na jednom nivou rekurzije. To je za svaki programski jezik manji ili veći problem, ovisno o memoriji računala i načinu rada. Npr. Fortran je tu nemoćan, dok je Prolog koncipiran na rekurziji.

Na LLispu je teško rekurziju simulirati iterativno, jer mu nedostaju uobičajene strukture za kontrolu toka programa.

Objašnjenje principa rada rekurzije

Osnovno je za rekurziju sljedeće:

1. Funkcija koja poziva drugu funkciju, nastavlja svoj rad tek kad pozvana funkcija izvrši zadatak.

2. Prilikom rekurzije, ista funkcija je aktivirana na više nivoa.

3. Svako aktiviranje funkcije otvara novu biblioteku varijabli. Te varijable žive koliko i poziv funkcije, osim ako nisu globalne.

Da bi se omogućila primjena rekurzije u LLisp-u, neophodno je korištenje ranije opisane strukture u obliku niza pointera, koja simulira stek podataka i gdje se čuvaju lokalne varijable između dva poziva funkcija. Algoritam rekurzije je sljedeći:

1. Na stek lokalnih varijabli se prije poziva funkcije, stave kopije svih lokalnih varijabli funkcije.[23]

2. Na stek upravljanja se stavi povratna adresa funkcije.

3. Na stek upravljanja se zatim stave argumenti funkcije i nađeni redeksi.

4. Gornji postupak se ponavlja sve dok se ne ispune uvjeti za izlazak iz rekurzije. Uvjet za izlazak mora postojati, inače će se stek prepuniti i prijaviti grešku.

5. U trenutku ispunjenja uvjeta za izlazak iz rekurzije, izvršava se tijelo funkcije koje je na vrhu steka, izvrši potrebna redukcija izraza, a adresa tijela i povratna adresa skinu sa steka.

Postupak se nastavlja dalje sve do trenutka kad na steku više ne bude elemenata. Pritom treba razlikovati:

- uvjet izlaska iz rekurzivne funkcije prekida stavljanje argumenata funkcije i lokalnih varijabli na stek,
- završetak rekurzije dolazi kad se svi podaci rekurzivne funkcije skinu sa steka.

Algoritam rekurzije je:

```
{
while stek nije prazan
    {
    do
        {
        Napravi kopije formalnih argumenata λ-apstrakcije;
        Kopijama formalnih argumenata pridruži stvarne argumente;
        Tako formirane formalne argumente stavi u 'kesu';
```

[23] Rekurzivnim pozivom se ne stvara nova kopija funkcije, već su samo argumenti u pozivu funkcije novi.

if 'kesa' nije prazna **then** stavi 'kesu' na stek podataka;
else 'kesa' je tekući vrh steka podataka.
Na stek upravljanja stavi zatim tijelo funkcije, povratnu adresu i redeks;
}

while nije ispunjen uvjet za izlazak iz rekurzije;

if ispunjen je uvjet za izlazak iz rekurzije **then** evaluiraj tijelo na vrhu steka, reduciraj izraz i skini povratnu adresu i redeks;

}

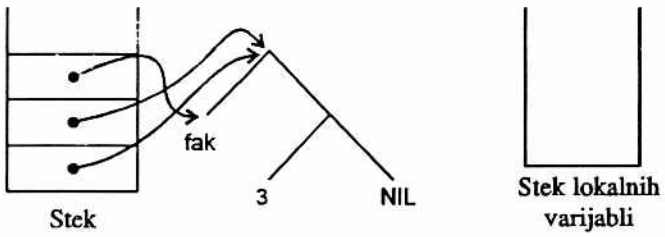
}

Primjer:
Pretpostavimo da je u fak definicija funkcije 'faktorijel':

$$(\text{defun } fak\ (n)\ (\text{cond } ((=\ n\ 0)\ 1)\ (\text{T } (*\ n\ (fak\ (-\ n\ 1)))))).$$

Tada se pozivom $(fak\ 3)$ dešava sljedeće:

1. Nakon poziva funkcije $(fak\ 3)$, na stek se stavlja povratna adresa $(fak\ 3)$ i redeks fak.

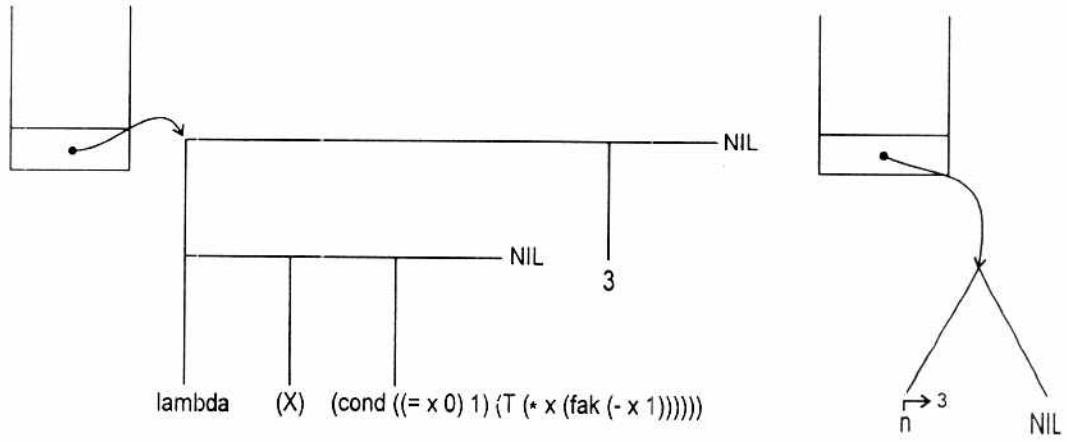


2. Varijabla fak je globalna i njena vrijednost je λ-apstrakcija

$$(\text{lambda } (n)\ (\text{cond } ((=n\ 0)\ 1)(\text{T } (*\ n\ (\text{fak } (-\ n\ 1)))))).$$

fak se zamjeni vrijednošću funkcije.

3. Nakon toga se pokuša primjeniti λ-apstrakcija na argument 3, pa dolazi do ukešenja. Time lokalna varijabla n dobija vrijednost 3 i kesa sa varijablom $n = 3$ stavi na stek podataka.



4. Pokuša se evaluirati tijelo λ-apstrakcije. Zato se prvo provjerava uvjet izlaska iz rekurzije. Kako on nije dostignut, pokušava se evaluirati izraz $(* \; n \; fak \; (- \; n \; 1))$.

5. Tako se dolazi do poziva funkcije $(fak \; (- \; n \; 1))$. Ponovo se na stek stavlja varijabla fak i evaluira u svoju λ-definiciju.

6. Da bi se ona primjenila na $(- \; n \; 1)$, prvo se evaluira argument i dobije vrijednost 2. Nakon toga dolazi do ukešenja. Time formalni argument (lokalna varijabla) n dobija novu vrijednost 2 i kesa sa varijablom $n = 2$ stavi na stek podataka.

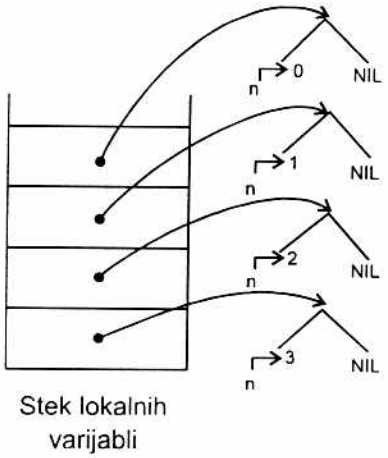
7. Sada dolazi do ponovnog poziva funkcije fak kroz $(fak \; 2)$.

8. Zato se gornji postupak ponovi još za $(fak \; 2)$ i $(fak \; 1)$. Sada dolazi do poziva $(fak \; 0)$ i sve je isto kao i ranije do trenutka kada se ne postigne uvjet izlaska iz rekurzije, tj. kad postane $(= \; n \; 0)$.

9. Pošto je uvjet izlaska iz rekurzije sada postignut, evaluira se tijelo funkcije i dobija vrijednost 1.

10. Kako se izraz $(fak \; 0)$ nalazi u povratnoj adresi (nadređenom tijelu), zamjeni se rezultatom 1, i tek sada varijabla $n = 0$ skida sa steka, skida redeks i povratna adresa.

11. Izračunava se vrijednost novog tijela za vrijednost 1 i dobija rezultat 1.



12. Kako se izraz $(fak\ 1)$ nalazi u povratnoj adresi (nadređenom tijelu), zamjeni se rezultatom 1, i tek sada varijabla $n = 1$ skida sa steka, skida redeks i povratna adresa.

13. Izračunava se vrijednost novog tijela za vrijednost 1 i dobija rezultat 2.

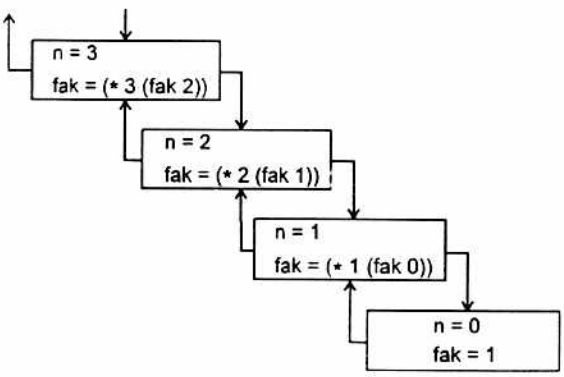
14. Kako se izraz $(fak\ 2)$ nalazi u povratnoj adresi (nadređenom tijelu), zamjeni se rezultatom 2, i tek sada varijabla $n = 2$ skida sa steka, skida redeks i povratna adresa.

15 Izračunava se vrijednost novog tijela za vrijednost 2 i dobija rezultat 6.

16. Kako se izraz $(fak\ 2)$ nalazi u povratnoj adresi (nadređenom tijelu), zamjeni se rezultatom 6, i tek sada varijabla $n = 3$ skida sa steka, skida redeks i povratna adresa.

17. Kako je dostignuto dno steka i više nema novih redeksa, dobiven je konačni rezultat 6.

Cijeli tok izvršavanja funkcije fak je prikazan na sljedećoj slici.



Algoritam rada funkcije $(fak\ 3)$

8.3.5 Opis grešaka

0. Stek lokalnih varijabli je pun
1. Dijeljenje nulom
2. Varijabla nije vezana
3. Premalo argumenata
4. Previše argumenata
5. Negativan argument
6. Nepravilan argument
7. Nedefinirana funkcija
8. Desna zagrada prije lijeve
9. Program veći od 512 byta
10. Dozvoljen ulaz samo jednom $*$-termu
11. Leksički neispravan izraz
12. Sintaksno neispravan izraz

8.4 Algoritam sakupljača smeća

Kako evaluacija u LLispu napreduje, da bi se formirali rezultati evaluacije u obliku atoma ili liste, uzimaju se ćelije iz 'listatom'. Mnogi od ovih rezultata su privremeni pa se ćelije koje oni koriste mogu izbrisati iz liste. Tako se smanjuje zauzeće memorije. Ovaj zadatak *"sakupljanja smeća"* obavlja funkcija *ss*. Ona to može raditi u trenutku kad se memorija računala smanji na kritični nivo ili stalno. Dakle, postoje dva algoritma rada funkcije *ss* i nazivamo ih:

- Metoda udaljavanja markera,
- Metoda brojanja.

8.4.1 Algoritam metode udaljavanja markera

Ovo je efikasna metoda, ali ne za rad u realnom vremenu.

- Stalno se ispituje stanje memorije, pa kad se memorija isprazni (ili smanji na kritičan nivo), evaluacija se privremeno zaustavi.
- *'ss'* tada prolazi kroz sve alfanumeričke atome i sve njihove vrijednosti u 'listatom'. Ako su ti atomi privremeni izbriše ih se i 'listatom' se prespoji.
- Nakon što *'ss'* prođe kroz cijelu listu 'listatom', nastavi se sa evaluacijom.

8.4.2 Algoritam metode brojanja

Ova je metoda prilagođena za rad u realnom vremenu.

- Stalno se ispituje broj pokazivanja za svaki objekt iz 'listatom'.
- Čim se naiđe na objekt kod koga je broj pokazivanja jednak 0, evaluacija se zaustavi.
- Taj se objekt izbriše, a lista objekata prespoji.
- Nakon toga se nastavi sa evaluacijom.

Mi ćemo realizirati posljednju metodu

Kako se zna da su neki atomi nepotrebni?

- U trenutku kada je neka varijabla vezana, tada se ona i svi atomi koji grade njenu vrijednost (neki $*$-terni), 'označe'. Tj. u programskoj strukturi koja realizira atom ostaviće se prostor za jednu varijablu *vez* koja označava broj vezivanja tog atoma, tj. broj pointera (pokazivanja) na taj atom.
- U trenucima evaluacije $*$-terma, svako pokazivanje na neku varijablu uvećava ovu varijablu.
- U trenucima kada varijabla mijenja vrijednost u neku drugu, pri prolazu kroz listatom, u svim atomima iz prethodne vrijednosti, varijabla *vez* se umanjuje.
- U trenutku 'sakupljanja smeća' i prolaza kroz *listatom* ispituje se da li je $vez = 0$. Ako nije, prođe se.
- Ako je $vez = 0$, pointeri sa drugih objekata na taj ne postoje i on postaje smeće. U trenutku kada brojač sa 1 prelazi na 0, pokrene se algoritam dealokacije. Prostor koji je taj objekt zauzimao se obriše i memorija tako oslobodi za nove strukture. Takav proces oslobađanja je ravnomjerno raspoređen u toku računskog procesa, pa proces računanja postaje efektivniji.

Sakupljač smeća *ss* nije dostupan korisniku.

Bibliografija

[1] J. McCarty, *Recursive functions of symbolic expressions and their computation by machine*, CACM. vol. 3, pp. 184-195 (1960)

[2] P. Henderson, *Funkcional Programming Application and Implementation*, Prentice Hall (1980)

[3] H.P. Barendregt, *The Lambda Calculus - Its Syntax and Semantics*, North-Holland, 1984

[4] J.E. Stoy, *Denotational Semantics*, MIT Press, 1981

[5] H.B. Curry, R. Feys, *Combinatory Logic*, Vol.1, North-Holland, 1958

[6] C.P. Wadsworth, *Semantics and Pragmatics of the Lambda Calculus*, Chapter 4, PhD Thesis, Oxford, 1971

[7] H. Abelson and G.J. Sussman, *Structure and Implementation of Computer Programs*, MIT Press, 1985

[8] J. Peyton, *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987

[9] L. Augustsson, *A Compiler for Lazy ML*, Proc. ACM Symposium on Lisp and Functional Programming, pp.218-227, 1984

[10] M. Amamija, J. Tanaka, *Arhitekture EVM i iskustvenij intelekt*, Moskva 'Mir', 1993. (prevod s japanskog)

[11] .... *De la Logique Classique a la Programmation Logique*, Moskva 'Mir', 1990

Dodatak

Kod programa LLisp.c:

```
#include <alloc.h>
#include <stdio.h>
#include <string.h>
#include <conio.h>
#include <ctype.h>
#include <math.h>
#include <setjmp.h>
#include <stdlib.h>
#include <io.h>

#include "struct.h"
#include "macroi.h"
#include "funkct.h"

drvo UNDEF;
drvo TRUE;
drvo LAMBDA;

jmp_buf status_programa;
drvo listatom;
drvored stek;
int duz;

drvo stek_kesa[MAX_STEK];
int sp_kesa;
drvo kesa;

drvo drvce(rijec x)
{
drvo z;

NEW_DRVO(z);
z->tip=1;
NEW_RIJEC(z->atom,x);
strcpy(z->atom,x);
return(z);
}
```

```
drvo drvo_var(rijec x)
{
drvo z;

NEW_DRVO(z);
z->tip=3;
NEW_RIJEC(z->ime,x);
strcpy(z->ime,x);
NEW_POINTER(z->pointer);
*(z->pointer)=UNDEF;
return(z);
}

int alfanumerik(rijec x)
{
while(isalnum(*x)||*x==95) x++;
return(!*x?1:0);
}

int broj(rijec x)
{
int i,znak=0,tocka=0;

x=glavni_dio(x);

for (i=0;i<strlen(x);++i)
{
                              if (*(x+i)==43||*(x+i)==45) ++znak;
else if (!isdigit(*(x+i)))
{
if (IS_POINT(*(x+i))) ++tocka;
else return(0);
}
else continue;
}
return(znak<=1&&tocka<=1);
}

int slobodna_prom(drvo x)
{
if (x->tip==3&&*(x->pointer)==UNDEF) return(1);
else return(0);
```

```
}

vezana_prom(drvo x)
{
  if (x->tip==3&&*(x->pointer)!=UNDEF) return(1);
else return(0);
}


drvo dodaj_atom(drvo x,drvo y)
{
drvo pom,priv=y;

NEW_DRVO(pom);
pom->tip=2;
pom->lrep=x;
pom->drep=NULL;
if (!y) y=pom;
else
{
while (priv->drep) priv=priv->drep;
priv->drep=pom;
}
return(y);
}

drvo atom_je_u_listi(rijec x, drvo d)
{
rijec r;

while (d)
{
if (d->lrep->tip==1) r=d->lrep->atom;
else r=d->lrep->ime;
if (!strcmp(x,r)) return(d->lrep);
else d=d->drep;
}
return(NULL);
}

void zamjeni_var(drvo x,drvo y)
{
drvo rez;
```

```
rez=atom_je_u_listi(x->ime,kesa);
if (!rez) rez=atom_je_u_listi(x->ime,listatom);
*(rez->pointer)=y;
}

void stavi_kesu_na_stek(void)
{
if (sp_kesa>MAX_STEK) greska(0);
stek_kesa[++sp_kesa]=kesa;
}

void skini_kesu_sa_steka(void)
{
if (!sp_kesa) greska(100);
kesa=stek_kesa[--sp_kesa];
}

void stavi_na_stek(drvo x)
{
drvored novi_stek;

NEW_STEK(novi_stek);
novi_stek->vrh=x;
novi_stek->rep=stek;
stek=novi_stek;
duz++;
}

int duzina_steka(drvored stek)
{
if (!stek) return(0);
else return(1+duzina_steka(stek->rep));
}

drvo prvi_koji_ide_na_stek(drvo x)
{
rijec lg;
drvo lok;

if (!x) return(NULL);
if (x->tip==1) return(NULL);
```

```
if (lambda_funkcija(x)) return(NULL);
if (vezana_prom(x)) return(NULL);

if (x->tip==2)
{
lg=prvi(x);

if (!strcmp(lg,"cond")||!strcmp(lg,"if")) return(NULL);
if (!strcmp(lg,"and")||!strcmp(lg,"or")) return(NULL);
if (!strcmp(lg,"quote")) return(NULL);
if (!strcmp(lg,"defun")) return(NULL);
if (!strcmp(lg,"setq"))
{
if (!x->drep||!x->drep->drep) return(NULL);
if (x->drep->drep->drep) greska(4);
return(prvi_koji_ide_na_stek(x->drep->drep));
}
}

while (x)
{
if (x->tip!=2) greska(6);
if (x->lrep->tip==1||!x->lrep)
{
x=x->drep;
continue;
}
if (x->lrep->tip==2) return(x);
if (vezana_prom(x->lrep)) return(x);
if (x->lrep->tip==3&&sp_kesa)
{
lok=atom_je_u_listi(x->lrep->ime,kesa);
x->lrep=lok;
return(x);
}
greska(2);
}
return(x);
}

drvo _eval(drvo x)
{
```

```
drvo pov_adr,rez;
drvored dno;
rijec lg;

if (x->tip!=2) return(eval_atoma(x));
if (lambda_funkcija(x)) return(x);

lg=prvi(x);
if (x->lrep->tip==1&&!sistemska(lg)) greska(7);

stavi_na_stek(x);
dno=stek;

if (stek->vrh->tip==2) pov_adr=prvi_koji_ide_na_stek(stek->vrh);
else pov_adr=NULL;

while(duz)
{
if (pov_adr)
{
stavi_na_stek(pov_adr);
stavi_na_stek(redeks);
pov_adr=prvi_koji_ide_na_stek(stek->vrh);
}
else
{
rez=eval(stek->vrh);
if (stek==dno)
{
skini_sa_steka();
return(rez);
}
else
{
stek->rep->vrh->lrep=rez;
skini_sa_steka();

if (rez->tip==1&&stek->rep->vrh->lrep==rez&&
 (rez==LAMBDA||sistemska(rez->atom)))
pov_adr=prvi_koji_ide_na_stek(stek->vrh);
else pov_adr=prvi_koji_ide_na_stek(stek->vrh->drep);
```

```
skini_sa_steka();

if (stek->vrh->lrep->tip==1&&!sistemska(stek->vrh->lrep->atom)&&
 stek->vrh->lrep!=LAMBDA||
 stek->vrh->lrep->tip==2&&!lambda_funkcija(stek->vrh->lrep)||
 stek->vrh->lrep->tip==3)
greska(7);
}
}
}
}

void ispis_liste(drvo x)
{
if (x)
{
printf("(");
while(x)
{
printf("(");
if (x->lrep->tip==1)
printf("%s", x->lrep->atom);
else if (*(x->lrep->pointer)==UNDEF) printf("%s",x->lrep->ime);
else
{
printf("%s ",x->lrep->ime);
printf("%s",print(*(x->lrep->pointer)));
}
printf(")");
x=x->drep;
}
printf(")");
printf("\n");
}
else printf("NIL\n");
}

rijec ucitaj()
{
char buff[512];
rijec x,y;
int i,z=0;
```

```
x=malloc(1);
x[0]='\0';

do
{
if (!z) printf("\n> ");
scanf(" %[^\n]",buff);

i=0;
while (buff[i]&&buff[i]!=';')
{
buff[i]=tolower(buff[i]);
if (!dozvoljen(buff[i])) greska(14);
i++;
if (i==511) greska(9);
}
buff[i]='\0';

y=malloc(strlen(x)+i+2);
if (!strcmp(x,"")) sprintf(y,"%s",buff);
else sprintf(y,"%s %s",x,buff);
free(x);
x=y;

z=zagrade(x);
if (!strcmp(x,"')) printf("> ");
else if (z>0) printf("%d> ",z);
}
while (z>0);
if (!analizator(x)) greska(15);
return(x);
}

int analizator(rijec x)
{
while(*x)
{
if (*x=='('&&*(x+1)=='|') return(0);
if (*x=='|'&&(*(x+1)==')'||*(x+1)=='|')) return(0);
if (*x=='\''&&(*(x+1)=='|'||*(x+1)==')')) return(0);
x++;
```

```
}
return(1);
}

int dozvoljen(char x)
{
if (isspace(x)||isalnum(x)) return(1);
if (x==39||x==40||x==41||x==42||x==43||x==45||x==46||x==47||
x==60||x==61||x==62||x==95||x==124) return(1);
return(0);
}

rijec glavni_dio(rijec x)
{
int glavni[2],i=0,j=0;
char ch;

while(ch=x[j])
{
switch(ch)
{
case ' ':
case '\t':
case '\n':
break;
default:
glavni[i]=j;
if (!i)
{
i=1;
glavni[i]=j;
}
}
j++;
}
if (!i) return(x+j);
*(x+glavni[1]+1)='\0';
return(x+glavni[0]);
}

int pravilan(rijec x)
{
```

```
int i=0;
rijec w;

NEW_RIJEC(w,x);
strcpy(w,x);

w=glavni_dio(w);
if (!strcmp(w,"")) return(0);
if (!strcmp(w,"nil")) return(2);
if (*w=='(')
{
do
{
if (*w=='(') i=i+1;
else if (*w==')') i=i-1;
++w;
}
while(i&&*w);

return(!i&&!*w?2:0);
}
else if (*w=='\'')
return(pravilan(w+1)?2:0);
else while (*w)
{
if (*w==' '||*w=='\t') return(0);
if (!isalnum(*w)&&*w!=95&&!atof(w)&&!oper(x)&&!bin_rel(x)) greska(15);
w++;
}
return(1);
}

int prazna(rijec x)
{
int i=0;

x=glavni_dio(x);
if (!strcmp(x,"")) return(1);
if (!strcmp(x,"nil")) return(1);
while(*x)
{
switch(*x)
```

```
{
case ' ' :
case '\t':
case '\n':
break;
case '(' :
i++;
break;
case ')' :
i--;
break;
default  :
return(0);
}
x++;
if (i&&i!=1) return(0);
}
return(!i?1:0);
}

void otkid(rijec *prvi,rijec *drugi,rijec x)
{
rijec gd,gl;
rijec q,r=NULL,t;

gd=glavni_dio(x);

if (*gd=='\'')
{
NEW_RIJEC(*prvi,"quote");
strcpy(*prvi,"quote");

gl=malloc(strlen(gd)+2);
sprintf(gl,"(%s)",gd+1);
*drugi=gl;
}
else if (*gd=='(')
{
t=glavni_dio(gd+1);
q=t;
do
{
```

```
if (r)
{
strcat(t,r);
free(r);
}
while
(*q!=' '&&*q!='\t'&&*q!='('&&*q!=')'&&*(q-1)!=')'&&*q!='\''&&*q!='|')
q++;

r=malloc(strlen(q)+1);
strcpy(r,q);
*q='\0';
q++;
}
while(!pravilan(t));

*prvi=t;
while (*r==' '||*r=='\t'||*r=='\n') r++;
if (*r=='!')
{
q=r;
while (*q) q++;
*(q-1)='\0';
*drugi=glavni_dio(r+1);
}
else
{
*drugi=malloc(strlen(r)+2);
sprintf(*drugi,"(%s",r);
}
}
else greska(100);
}


drvo drvo_atoma(rijec x)
{
drvo d,rez;

if (broj(x)||bin_rel(x)||oper(x)||(!strcmp(x,"atomlist")))
return(drvce(x));
if (rez=atom_je_u_listi(x,kesa))
return(rez);
```

```
else if (rez=atom_je_u_listi(x,listatom))
return(rez);
else
{
if (kljucna_rijec(x)) d=drvce(x);
else d=drvo_var(x);
listatom=dodaj_atom(d,listatom);
return(d);
}
}

drvo odrvi(rijec x)
{
int sta;

if (!strcmp(x,"t")) return(TRUE);
if (!strcmp(x,"lambda")) return(LAMBDA);
if (!strcmp(x,"undef")) return(UNDEF);
if (prazna(x)) return(NULL);

sta=pravilan(x);
if (sta==1) return(drvo_atoma(x));
else if (sta==2) return(drvo_tpara(x));
else greska(13);
}

drvo drvo_tpara(rijec x)
{
rijec p,q;
drvo d;

otkid(&p,&q,x);

NEW_DRVO(d);
d->tip=2;
d->lrep=odrvi(p);
d->drep=odrvi(q);
return(d);
}

drvo kopija_formalnih_arg(drvo x)
{
```

```
drvo d;

if (!x||x==UNDEF||x==TRUE||x==LAMBDA) return(x);

switch(x->tip)
{
case 1:
NEW_DRVO(d);
d->tip=1;
NEW_RIJEC(d->atom,x->atom);
strcpy(d->atom,x->atom);
return(d);
case 2:
return(kopija_drva(x));
case 3:
NEW_DRVO(d);
d->tip=3;
NEW_RIJEC(d->ime,x->ime);
strcpy(d->ime,x->ime);
NEW_POINTER(d->pointer);
if (*(x->pointer)!=UNDEF)
*(d->pointer)=kopija_drva(*(x->pointer));
else
*(d->pointer)=UNDEF;
return(d);
default:
greska(100);
}
}

drvo kopija_drva(drvo x)
{
drvo d;

if (!x||x==UNDEF||x==TRUE||x==LAMBDA) return(x);

switch (x->tip)
{
case 1:
return(x);
case 2:
NEW_DRVO(d);
```

```
d->tip=2;
d->lrep=kopija_drva(x->lrep);
d->drep=kopija_drva(x->drep);
return(d);
case 3:
NEW_DRVO(d);
d->tip=3;
NEW_RIJEC(d->ime,x->ime);
strcpy(d->ime,x->ime);
d->pointer=x->pointer;
return(d);
default:
greska(100);
}
}

rijec print(drvo d)
{
rijec r,r1,r2;

if (!d) return("NIL");

switch(d->tip)
{
case 1:
return(d==TRUE?"T":d->atom);
case 3:
return(d->ime);
case 2:
r1=print(d->lrep);
while (d->drep)
{
if (d->drep->tip==1||d->drep->tip==3)
{
r2=print(d->drep);
r=malloc(strlen(r1)+strlen(r2)+4);
sprintf(r,"(%s|%s)",r1,r2);
return(r);
}
r2=print(d->drep->lrep);
r=malloc(strlen(r1)+strlen(r2)+2);
sprintf(r,"%s %s",r1,r2);
```

```
r1=r;
d=d->drep;
}
r=malloc(strlen(r1)+3);
sprintf(r,"(%s)",r1);
return(r);
default:
greska(100);
}
}

drvo upis(void)
{
            rijec x;
drvo d;

x=ucitaj();
d=odrvi(x);
return(d);
}

int zagrade(rijec x)
{
int z=0;

x=glavni_dio(x);
if (!strcmp(x,"'")) return(1);

while (*x)
{
if (*x=='(') z++;
else if (*x==')') z--;
if (z<0) greska(8);
x++;
}
return(z);
}

int kljucna_rijec(rijec x)
{
if (sistemska(x)||lisp_atom(x)) return(1);
else return(0);
```

```
}

int oper(rijec x)
{
if (!strcmp(x,"+")||!strcmp(x,"-")||!strcmp(x,"*")||!strcmp(x,"/"))
return(1);
else return(0);
}

int mat_fun(rijec x)
{
if (!strcmp(x,"sin")||!strcmp(x,"cos")||!strcmp(x,"ln")||!strcmp(x,"exp")||
!strcmp(x,"abs")||!strcmp(x,"sgn")) return(1);
else return(0);
}

int bin_rel(rijec x)
{
if (!strcmp(x,"<")||!strcmp(x,">")||!strcmp(x,"=")) return(1);
else return(0);
}

int un_rel(rijec x)
{
if (!strcmp(x,"numberp")||!strcmp(x,"minusp")||!strcmp(x,"zerop")||
!strcmp(x,"evenp")||!strcmp(x,"oddp"))
return(1);
else return(0);
}

int log_fun(rijec x)
{
if (!strcmp(x,"not")||!strcmp(x,"and")||!strcmp(x,"or")) return(1);
else return(0);
}

int list_fun(rijec x)
{
if (!strcmp(x,"car")||!strcmp(x,"cdr")||!strcmp(x,"cons")||
    !strcmp(x,"boundp")||!strcmp(x,"pairp")||!strcmp(x,"null")||
    !strcmp(x,"atomp")||!strcmp(x,"idp")||!strcmp(x,"constp")||
    !strcmp(x,"list")||!strcmp(x,"quote")||!strcmp(x,"apply")||
```

```
    !strcmp(x,"eval")||!strcmp(x,"cond")||!strcmp(x,"if")||
    !strcmp(x,"setq")||!strcmp(x,"set")||!strcmp(x,"append")||
    !strcmp(x,"member")||!strcmp(x,"eq")||!strcmp(x,"equal")||
                !strcmp(x,"load")||!strcmp(x,"progn")||!strcmp(x,"defun")||
    !strcmp(x,"atomlist")||!strcmp(x,"length")||!strcmp(x,"read")||
    !strcmp(x,"exit")||!strcmp(x,"cls")||!strcmp(x,"mem")||
    !strcmp(x,"listp")||!strcmp(x,"print"))
return(1);
else return(0);
}

int lisp_atom(rijec x)
{
if (!strcmp(x,"t")||!strcmp(x,"nil")||!strcmp(x,"undef")||
                !strcmp(x,"lambda"))
return(1);
else return(0);
}

drvo numberp(drvo x)
{
if (x->tip==1&&broj(x->atom)) return(TRUE);
            if (x->tip==3&&numberp(*(x->pointer))) return(TRUE);
else return(NULL);
}

drvo minusp(drvo x)
{
double r;

r=atof(x->atom);
return(r<0?TRUE:NULL);
}

drvo zerop(drvo x)
{
double r;

r=atof(x->atom);
return(!r?TRUE:NULL);
}
```

```
drvo evenp(drvo x)
{
double r;

r=atof(x->atom);
return(!r||r&&!fmod(r,2)?TRUE:NULL);
}

drvo oddp(drvo x)
{
double r;

r=atof(x->atom);
return(r&&fmod(r,2)==1?TRUE:NULL);
}

drvo null(drvo x)
{
return(!x?TRUE:NULL);
}

drvo boundp(drvo x)
{
return(*(x->pointer)!=UNDEF?TRUE:NULL);
}

drvo atomp(drvo x)
{
return(!x||x->tip!=2?TRUE:NULL);
}

drvo pairp(drvo x)
{
return(!x||x->tip==2?TRUE:NULL);
}

drvo listp(drvo x)
{
if (!x||x->tip==2&&pairp(x->drep)) return(TRUE);
else return(NULL);
}
```

```
drvo idp(drvo x)
{
return(x->tip==3?TRUE:NULL);
}

drvo constp(drvo x)
{
return(!x||x->tip==1?TRUE:NULL);
}

drvo car(drvo x)
{
              if (x&&x->tip!=2) greska(6);
return(!x?x:x->lrep);
}

drvo cdr(drvo x)
{
              if (x&&x->tip!=2) greska(6);
return(!x?x:x->drep);
}

drvo cons(drvo x,drvo y)
{
drvo d;

NEW_DRVO(d);
d->tip=2;
d->lrep=x;
d->drep=y;
return(d);
}

drvo not(drvo x)
{
return(!x->lrep?TRUE:NULL);
}

drvo and(drvo x)
{
drvo priv=TRUE;
```

```
while(priv&&x)
{
priv=x->lrep;
x=x->drep;
}
return(priv);
}

drvo or(drvo x)
{
drvo priv=NULL;

while(!priv&&x)
{
priv=x->lrep;
x=x->drep;
}
return(priv);
}

drvo member(drvo x,drvo y)
{
if (!y||y->tip==1) return(NULL);
else
{
if (equal(x,y->lrep)) return(y);
else return(member(x,y->drep));
}
}

drvo eq(drvo x,drvo y)
{
return(x==y?TRUE:NULL);
}

drvo equal(drvo x, drvo y)
{
if (x==y) return(TRUE);
else if (x->tip==1&&y->tip==1&&!strcmp(x->atom,y->atom)) return(TRUE);
else if (x->tip==2&&y->tip==2&&equal(car(x),car(y)))
                return(equal(cdr(x),cdr(y)));
else if (x->tip==3&&y->tip==3&&!strcmp(x->ime,y->ime)) return(TRUE);
```

```
else return(NULL);
}

drvo append(drvo x)
{
drvo priv;

if (!x) return x;
if (x->tip!=2) greska(6);
if (!x->lrep) return(append(x->drep));
if (x->lrep->tip!=2) greska(6);

priv=x->lrep;
while (priv->drep)
{
if (priv->drep->tip!=2) greska(6);
priv=priv->drep;
}
priv->drep=append(x->drep);
return(x->lrep);
}

drvo apply(drvo x)
{
if (!x->lrep) greska(7);
if (x->lrep->tip==1&&!sistemska(x->lrep->atom)) greska(7);
if (x->lrep->tip==2&&!lambda_funkcija(x->lrep)) greska(7);

x->drep=x->drep->lrep;
return(eval(x));
}

drvo progn(drvo x)
{
while (x->drep) x=x->drep;
return(x->lrep);
}

int duzina(drvo x)
            {
            if (!x||x->tip!=2) return(0);
            else return(1+duzina(x->drep));
```

```
            }

drvo length(drvo x)
            {
            rijec r;
            int n;

n=duzina(x);
sprintf(r,"%d",n);
return(drvce(r));
            }

int sistemska(rijec x)
{
if (mat_fun(x)||list_fun(x)||oper(x)||bin_rel(x)||un_rel(x)||log_fun(x))
return(1);
else return(0);
}

drvo lambda_funkcija(drvo x)
{
if (!x) return(NULL);
if (x->tip==2&&x->lrep==LAMBDA&&x->drep->tip==2&&
   (x->drep->lrep->tip==2||!x->drep->lrep)&&x->drep->drep)
return(x);
else return(NULL);
}

rijec prvi(drvo x)
{
if (!x||x->tip!=2) greska(6);
if (!x->lrep) return("NIL");
else if (x->lrep->tip==1) return(x->lrep->atom);
else if (x->lrep->tip==3) return(x->lrep->ime);
else if (x->lrep->tip==2) return(print(x->lrep));
else greska(100);
}


void greska(int kod_greske)
{
switch(kod_greske)
{
```

```
case 0:
printf("Stek lokalnih varijabli je pun!");
break;
                              case 1:
printf("Dijeljenje nulom");
break;
case 2:
printf("Varijabla nije vezana");
break;
case 3:
printf("Premalo argumenata");
break;
case 4:
printf("Previse argumenata");
break;
case 5:
printf("Argument negativan");
break;
case 6:
printf("Nepravilan argument");
break;
case 7:
printf("Nedefinirana funkcija");
break;
case 8:
printf("Desna zagrada prije lijeve");
break;
case 9:
printf("Program veci od 512 byta");
break;
case 13:
printf("Dozvoljen je ulaz samo jednom *-termu");
break;
case 14:
                                          printf("Leksicki neispravan izraz");
break;
case 15:
printf("Sintaksno neispravan izraz");
break;
case 100:
printf("Kopira ti se neko drvo van kontrole!!!");
break;
```

```
default:
break;
}
longjmp(status_programa,kod_greske);
}

drvo eval_atoma(drvo x)
{
drvo lok,glob;

if (!x) return(x);
if (x->tip==1)
{
if (broj(x->atom)||lisp_atom(x->atom)||sistemska(x->atom))
return(x);
else greska(2);
}
if (x->tip==3)
{
if (lok=atom_je_u_listi(x->ime,kesa))
return(*(lok->pointer));
glob=atom_je_u_listi(x->ime,listatom);
if (*(glob->pointer)==UNDEF) greska(2);
return(*(glob->pointer));
}
}

drvo eval_load(drvo x)
{
unsigned long size;
char priv[512];
rijec y,z;
FILE *prog;
drvo d;
int j=0;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
if (x->drep->lrep->tip!=3) greska(6);
y=malloc(strlen(x->drep->lrep->ime)+5);
sprintf(y,"%s.lsp",x->drep->lrep->ime);
printf("; Ucitavam %c%s%c\n",34,y,34);
```

```
prog=fopen(y,"rt");
if (!prog) return(NULL);
size=filelength(fileno(prog));
free(y);
y=malloc(size+1);
size=fread(y,1,size,prog);
y[size]='\0';
fclose(prog);

while (*y)
{
while(*y!=';'&&*y)
{
if (isspace(*y))
{
if (isspace(*(y+1)))
{
y++;
continue;
}
else *y=' ';
}
priv[j]=tolower(*y);
y++;
j++;
if (j==511) greska(9);
}
while (*y!='\n'&&*y) y++;
if (*y) y++;
}

priv[j]='\0';
y=malloc(strlen(priv)+1);
sprintf(y,"%s",priv);
z=malloc(strlen(y)+9);
sprintf(z,"(progn %s)",y);

d=odrvi(z);
_eval(d);
return(TRUE);
}
```

```
drvo eval_oper(drvo x)
{
drvo d1,d2;
rijec lg,r;
double r1,r2,i;

r=(rijec)malloc(30);

              lg=prvi(x);
if (!x->drep) greska(3);
x=x->drep;
d1=x->lrep;
if (d1->tip==1&&broj(d1->atom)) r1=atof(d1->atom);
else greska(6);
x=x->drep;

do
{
if (!x)
{
if (!strcmp(lg,"-")) r1=0-r1;
continue;
}
d2=x->lrep;
if (d2->tip==1&&broj(d2->atom)) r2=atof(d2->atom);
else greska(6);
if (!strcmp(lg,"+")) r1=r1+r2;
                            else if (!strcmp(lg,"-")) r1=r1-r2;
else if (!strcmp(lg,"*")) r1=r1*r2;
else if (!strcmp(lg,"/")&&!r2) greska(1);
else r1=r1/r2;
x=x->drep;
}
while(x);
if (!modf(r1,&i)) sprintf(r,"%-.0f",r1);
else sprintf(r,"%f",r1);
return(drvce(r));
}

drvo eval_fun(drvo x)
{
double r1,i;
```

```
rijec lg,r;
drvo d1;

r=(rijec)malloc(30);

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
if (x->drep->tip!=2) greska(6);
d1=x->drep->lrep;
if (d1->tip==1&&broj(d1->atom)) r1=atof(d1->atom);
else greska(6);
              lg=prvi(x);
if (!strcmp(lg,"sin"))
r1=sin(r1);
else if (!strcmp(lg,"cos"))
r1=cos(r1);
else if (!strcmp(lg,"exp"))
r1=exp(r1);
else if (!strcmp(lg,"ln"))
{
if (r1<=0) greska(5);
else r1=log(r1);
}
else if (!strcmp(lg,"abs"))
r1=fabs(r1);
else if (!strcmp(lg,"sgn"))
r1=SGN(r1);
if (!modf(r1,&i)) sprintf(r,"%-.0f",r1);
else sprintf(r,"%f",r1);
return(drvce(r));
}

drvo eval_bin_rel(drvo x)
{
double r1,r2;
rijec lg;
drvo d1,d2;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);
d1=x->drep->lrep;
d2=x->drep->drep->lrep;
```

```
if ((d1->tip!=1||d2->tip!=2)&&(!broj(d1->atom)||!broj(d2->atom)))
greska(6);
r1=atof(d1->atom);
r2=atof(d2->atom);

lg=prvi(x);
if (!strcmp(lg,">"))
return(r1>r2?TRUE:NULL);
else if (!strcmp(lg,"<"))
return(r1<r2?TRUE:NULL);
else
return(r1==r2?TRUE:NULL);
}

drvo eval_un_rel(drvo x)
{
rijec lg;
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);

d=x->drep->lrep;
lg=prvi(x);
if (!strcmp(lg,"numberp")) return(numberp(d));
if (d->tip!=1&&!broj(d->atom)) greska(6);
if (!strcmp(lg,"zerop")) return(zerop(d));
if (!strcmp(lg,"minusp")) return(minusp(d));
if (!strcmp(lg,"evenp")) return(evenp(d));
if (!strcmp(lg,"oddp")) return(oddp(d));
}

drvo eval_print(drvo x)
{
drvo d;
rijec r;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
r=print(d);
```

```
puts(r);
return(d);
}

drvo eval_read(drvo x)
{
drvo d;

if (x->drep) greska(4);
d=upis();
return(d);
}

drvo eval_null(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4),
d=x->drep->lrep;
return(null(d));
}

drvo eval_boundp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
if (d->tip!=3) greska(6);
return(boundp(d));
}

drvo eval_atomp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(atomp(d));
```

```
}

drvo eval_pairp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(pairp(d));
}

drvo eval_listp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(listp(d));
}

drvo eval_idp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(idp(d));
}

drvo eval_constp(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(constp(d));
}
```

```
drvo eval_car(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(car(d));
}

drvo eval_cdr(drvo x)
{
drvo d;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
d=x->drep->lrep;
return(cdr(d));
}

drvo eval_quote(drvo x)
{
if (!x->drep) greska(3);
if (x->drep->drep) greska(4);
return(x->drep->lrep);
}

drvo eval_list(drvo x)
{
return(x->drep);
}

drvo eval_length(drvo x)
              {
              if (!x->drep) greska(3);
              if (x->drep->drep) greska(4);
              if (x->drep->lrep&&x->drep->lrep->tip!=2) greska(6);

              return(length(x->drep->lrep));
              }

drvo eval_cons(drvo x)
```

```
{
drvo d1,d2;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);

d1=x->drep->lrep;
d2=x->drep->drep->lrep;
return(cons(d1,d2));
}

drvo eval_if(drvo x)
{
drvo uvjet, arg1, arg2, arg3;
drvo rez=NULL;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->tip!=2||x->drep->drep->tip!=2) greska(3);
if (x->drep->drep->drep&&x->drep->drep->drep->tip!=2) greska(3);
if (x->drep->drep->drep&&x->drep->drep->drep->drep) greska(4);

arg1=x->drep->lrep;
if (!arg1||arg1->tip==1) uvjet=arg1;
else uvjet=_eval(arg1);
if (uvjet)
{
arg2=x->drep->drep->lrep;
if (!arg2||arg1->tip==1) rez=arg2;
else rez=_eval(arg2);
}
else if (x->drep->drep->drep)
{
arg3=x->drep->drep->drep->lrep;
rez=_eval(arg3);
}
return(rez);
}

drvo eval_cond(drvo x)
{
drvo arg,uvjet,izraz,rez;
```

```
while(x->drep)
{
arg=x->drep->lrep;
if (!arg) greska(3);
if (arg->tip!=2) greska(6);
if (!arg->lrep) uvjet=NULL;
else if (arg->lrep->tip==1) uvjet=arg->lrep;
else uvjet=_eval(arg->lrep);

if (uvjet)
{
rez=uvjet;
while(arg->drep)
{
if (arg->drep->tip!=2) greska(6);
else izraz=arg->drep->lrep;
rez=_eval(izraz);
arg=arg->drep;
}
return(rez);
}
else x=x->drep;
}
return(NULL);
}

drvo eval_set(drvo x)
{
drvo d;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);
if (x->drep->lrep->tip!=3) greska(6);

zamjeni_var(x->drep->lrep,x->drep->drep->lrep);
              d=x->drep->drep->lrep;
return(d);
}

drvo eval_setq(drvo x)
{
drvo d;
```

```
if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);
if (x->drep->lrep->tip!=3) greska(6);

zamjeni_var(x->drep->lrep,x->drep->drep->lrep);
              d=x->drep->drep->lrep;
return(d);
}

drvo eval_eval(drvo x)
{
drvo d,k;

if (!x->drep) greska(3);
if (x->drep->drep) greska(4);

d=kopija_drva(x->drep->lrep);
k=_eval(d);

return(k);
}

drvo eval_append(drvo x)
{
if (!x->drep) return(x->drep);
if (x->drep->tip!=2) greska(6);

return(append(x->drep));
}

drvo eval_and(drvo x)
{
return(and(x->drep));
}

drvo eval_or(drvo x)
{
return(or(x->drep));
}

drvo eval_member(drvo x)
```

```
{
drvo d1,d2;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);

d1=x->drep->lrep;
d2=x->drep->drep->lrep;
return(member(d1,d2));
}

drvo eval_eq(drvo x)
{
drvo d1,d2;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);

d1=x->drep->lrep;
d2=x->drep->drep->lrep;
return(eq(d1,d2));
}

drvo eval_equal(drvo x)
{
drvo d1,d2;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);

d1=x->drep->lrep;
d2=x->drep->drep->lrep;
return(equal(d1,d2));
}

drvo eval_atomlist(drvo x)
{
if (x->drep) greska(4);
return(listatom);
}

drvo eval_exit(drvo x)
```

```
{
if (x->drep) greska(4);
exit(0);
}

drvo eval_defun(drvo x)
{
drvo lam,d;

if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->lrep->tip!=3) greska(6);

NEW_DRVO(lam);
lam->tip=2;
lam->lrep=LAMBDA;
lam->drep=x->drep->drep;

zamjeni_var(x->drep->lrep,lam);
d=x->drep->lrep;
return(d);
}

drvo eval_cls(drvo x)
{
if (x->drep) greska(4);
clrscr();
}

drvo eval_mem(drvo x)
{
if (x->drep) greska(4);
printf("Jos memorije: %lu bajta\n",coreleft());
return(NULL);
}


void ukesenje(drvo formalni, drvo stvarni)
{
drvo priv_l,priv_d;

if (formalni&&!stvarni) greska(3);
if (!formalni&&stvarni) greska(4);
if (!formalni&&!stvarni) return;
```

```
if (formalni->tip!=stvarni->tip) greska(6);

priv_l=formalni->lrep;
priv_d=formalni->drep;

priv_l->pointer=&(stvarni->lrep);

if (!priv_d&&!stvarni->drep) return;
else ukesenje(priv_d,stvarni->drep);
}

drvo eval_lambda(drvo y)
{
drvo formalni,stvarni;
drvo tijela,d;
drvo priv=kesa;

formalni=y->lrep->drep->lrep;
stvarni=y->drep;

if (!y->lrep->drep->drep) greska(3);

formalni=kopija_formalnih_arg(formalni);
ukesenje(formalni,stvarni);
kesa=formalni;
             if (kesa) stavi_kesu_na_stek();
             else kesa=priv;

tijela=kopija_drva(y->lrep->drep->drep);
while (tijela)
{
if (tijela->tip!=2) greska(6);
d=_eval(tijela->lrep);
tijela=tijela->drep;
}

             if (kesa) skini_kesu_sa_steka();
return(d);
}

drvo eval_apply(drvo x)
```

```
{
if (!x->drep||!x->drep->drep) greska(3);
if (x->drep->drep->drep) greska(4);
if (x->drep->drep->lrep&&x->drep->drep->lrep->tip!=2) greska(6);
if (x->drep->lrep->tip==3) x->drep->lrep=*(x->drep->lrep->pointer);

return(apply(x->drep));
}

drvo eval(drvo x)
{
rijec lg;

if (x->tip!=2) return(eval_atoma(x));
if (x->lrep==LAMBDA) return(x);

lg=prvi(x);

if (oper(lg))
return(eval_oper(x));
else if (mat_fun(lg))
return(eval_fun(x));
else if (bin_rel(lg))
return(eval_bin_rel(x));
else if (un_rel(lg))
return(eval_un_rel(x));
else if (!strcmp(lg,"cons"))
return(eval_cons(x));
else if (!strcmp(lg,"car"))
return(eval_car(x));
else if (!strcmp(lg,"cdr"))
return(eval_cdr(x));
else if (!strcmp(lg,"quote"))
return(eval_quote(x));
else if (!strcmp(lg,"list"))
return(eval_list(x));
else if (!strcmp(lg,"if"))
return(eval_if(x));
else if (!strcmp(lg,"cond"))
return(eval_cond(x));
else if (!strcmp(lg,"setq"))
return(eval_setq(x));
```

```
else if (!strcmp(lg,"set"))
return(eval_set(x));
else if (!strcmp(lg,"not"))
return(eval_null(x));
else if (!strcmp(lg,"and"))
return(eval_and(x));
else if (!strcmp(lg,"or"))
return(eval_or(x));
else if (!strcmp(lg,"null"))
return(eval_null(x));
else if (!strcmp(lg,"eq"))
return(eval_eq(x));
else if (!strcmp(lg,"equal"))
return(eval_equal(x));
else if (!strcmp(lg,"load"))
return(eval_load(x));
else if (!strcmp(lg,"progn"))
return(progn(x));
else if (lambda_funkcija(x->lrep))
return(eval_lambda(x));
else if (!strcmp(lg,"defun"))
return(eval_defun(x));
else if (!strcmp(lg,"apply"))
return(eval_apply(x));
else if (!strcmp(lg,"eval"))
return(eval_eval(x));
else if (!strcmp(lg,"print"))
return(eval_print(x));
else if (!strcmp(lg,"read"))
return(eval_read(x));
else if (!strcmp(lg,"pairp"))
return(eval_pairp(x));
else if (!strcmp(lg,"listp"))
return(eval_listp(x));
else if (!strcmp(lg,"atomp"))
return(eval_atomp(x));
else if (!strcmp(lg,"idp"))
return(eval_idp(x));
else if (!strcmp(lg,"boundp"))
return(eval_boundp(x));
else if (!strcmp(lg,"constp"))
return(eval_constp(x));
```

```
else if (!strcmp(lg,"append"))
return(eval_append(x));
else if (!strcmp(lg,"member"))
return(eval_member(x));
else if (!strcmp(lg,"length"))
return(eval_length(x));
else if (!strcmp(lg,"atomlist"))
return(eval_atomlist(x));
else if (!strcmp(lg,"exit"))
return(eval_exit(x));
else if (!strcmp(lg,"cls"))
return(eval_cls(x));
else if (!strcmp(lg,"mem"))
return(eval_mem(x));
else greska(7);
}

main()
{
drvo w1,w2;

              UNDEF=drvce("undef");
              TRUE=drvce("t");
              LAMBDA=drvce("lambda");

clrscr();
              printf("LLisp version 3.9, by Dzevad Lugic.");
              setjmp(status_programa);

while (1)
{
kesa=NULL;
stek=NULL;

printf("\nJos %lu bajta!",coreleft());
w1=upis();
w2=_eval(w1);
printf("%s",print(w2));
}
}
```

'Include' datoteka "struct.h":

```
typedef struct cvor
{
int tip;
union
{
char *Atom;
struct
{
struct cvor *Lrep;
struct cvor *Drep;
} rep;
struct
{
char *Ime;
struct cvor **Pointer;
} id;
} ukupno;
} CVOR;

typedef CVOR *drvo;
typedef char *rijec;

typedef struct Clan
{
drvo vrh;
struct Clan *rep;
} CLAN;

typedef CLAN *drvored;

#define NEW_DRVO(x) x=(drvo)malloc(sizeof(CVOR))
#define NEW_RIJEC(x,y) x=malloc(1+strlen(y))
#define NEW_POINTER(x) x=(drvo *)malloc(sizeof(drvo))
#define NEW_STEK(x) x=(drvored)malloc(sizeof(CLAN))

#define atom ukupno.Atom
#define lrep ukupno.rep.Lrep
#define drep ukupno.rep.Drep
#define ime ukupno.id.Ime
#define pointer ukupno.id.Pointer
```

'Include' datoteka "macroi.h":

```
#define IS_POINT(x) ((x)==46)
#define SGN(x) ((x)>=0?1:-1)
#define skini_sa_steka() stek=stek->rep; duz--
#define redeks pov_adr->lrep
#define MAX_STEK 64
```

'Include' datoteka "funkct.h":

```
drvo upis(void);
rijec ucitaj(void);
dozvoljen(char);
zagrade(rijec);
analizator(rijec);
pravilan(rijec);
prazna(rijec);
rijec glavni_dio(rijec);
void otkid(rijec *,rijec *,rijec);

drvo drvce(rijec);
drvo drvo_atoma(rijec);
drvo drvo_var(rijec);
drvo drvo_lokalne_var(rijec);
drvo drvo_tpara(rijec);
drvo odrvi(rijec);

alfanumerik(rijec);
broj(rijec);

slobodna_prom(drvo);
vezana_prom(drvo);
drvo lambda_funkcija(drvo);
rijec print(drvo);
duzina(drvo);

drvo kopija_formalnih_arg(drvo);
drvo kopija_drva(drvo);

oper(rijec);
mat_fun(rijec);
```

```
bin_rel(rijec);
un_rel(rijec);
list_fun(rijec);
log_fun(rijec);

drvo atomp(drvo);
drvo listp(drvo);
drvo numberp(drvo);
drvo idp(drvo);
drvo constp(drvo);
drvo minusp(drvo);
drvo zerop(drvo);
drvo evenp(drvo);
drvo zerop(drvo);
drvo quote(drvo);
drvo null(drvo);
drvo car(drvo);
drvo cdr(drvo);
drvo cons(drvo,drvo);
drvo and(drvo);
drvo or(drvo);
drvo member(drvo,drvo);
drvo eq(drvo,drvo);
drvo equal(drvo,drvo);
drvo append(drvo);
drvo eval(drvo);
drvo apply(drvo);
drvo progn(drvo);
drvo length(drvo);
drvo boundp(drvo);

lisp_atom(rijec);
kljucna_rijec(rijec);
sistemska(rijec);
rijec prvi(drvo);
void greska(int);

drvo eval_atoma(drvo);
drvo eval_oper(drvo);
drvo eval_fun(drvo);
drvo eval_bin_rel(drvo);
drvo eval_un_rel(drvo);
```

```
drvo eval_quote(drvo);
drvo eval_print(drvo);
drvo eval_set(drvo);
drvo eval_setq(drvo);
drvo eval_car(drvo);
drvo eval_cdr(drvo);
drvo eval_if(drvo);
drvo eval_cond(drvo);
drvo eval_list(drvo);
drvo eval_null(drvo);
drvo eval_listp(drvo);
drvo eval_atomp(drvo);
drvo eval_idp(drvo);
drvo eval_constp(drvo);
drvo eval_eq(drvo);
drvo eval_equal(drvo);
drvo eval_append(drvo);
drvo eval_and(drvo);
drvo eval_or(drvo);
drvo eval_member(drvo);
drvo eval_eval(drvo);
drvo eval_apply(drvo);
drvo eval_load(drvo);
drvo eval_progn(drvo);
drvo eval_lambda(drvo);
drvo eval_defun(drvo);
drvo eval_length(drvo);
drvo eval_boundp(drvo);
drvo eval_upis(drvo);
drvo eval_atomlist(drvo);
drvo eval_exit(drvo);
drvo eval_cls(drvo);
drvo eval_mem(drvo);

void ispis_liste(drvo);

drvo dodaj_atom(drvo,drvo);
drvo atom_je_u_listi(rijec,drvo);
void zamjeni_var(drvo,drvo);

void ukesenje(drvo,drvo);
void stavi_kesu_na_stek(void);
```

```
void uzmi_kesu_sa_steka(void);

void stavi_na_stek(drvo);
duzina_steka(drvored);
drvo prvi_koji_ide_na_stek(drvo);

drvo _eval(drvo);
```