Kazimir Majorinc

Moćan koliko je god moguće

$$
\begin{aligned}
&eval[e; a] = \\
&\quad [atom[e] \rightarrow assoc[e; a]; \\
&\qquad atom[car[e]] \rightarrow [eq[car[e]; \texttt{QUOTE}] \rightarrow cadr[e]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{ATOM}] \rightarrow atom[eval[cadr[e]; a]]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{EQ}] \rightarrow eq[eval[cadr[e]; a]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad eval[caddr[e]; a]]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{COND}] \rightarrow evcon[cdr[e]; a]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{CAR}] \rightarrow car[eval[cadr[e]; a]]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{CDR}] \rightarrow cdr[eval[cadr[e]; a]]; \\
&\qquad\qquad\qquad eq[car[e]; \texttt{CONS}] \rightarrow cons[eval[cadr[e]; a]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad eval[caddr[e]; a]]; \\
&\qquad\qquad\qquad \texttt{T} \rightarrow eval[cons[assoc[car[e]; a]; cdr[e]]; a]]; \\
&\qquad eq[caar[e]; \texttt{LABEL}] \rightarrow eval[cons[caddar[e]; cdr[e]]; \\
&\qquad\qquad\qquad\qquad\qquad cons[list[cadar[e]; car[e]]; a]]]; \\
&\qquad eq[caar[e]; \texttt{LAMBDA}] \rightarrow \\
&\qquad\quad eval[caddar[e]; append[pair[cadar[e];evlis[cdr[e];a]]; a]]]
\end{aligned}
$$

KAZIMIR MAJORINC: *Moćan koliko je god moguće*
Glavne ideje Lispa u McCarthyjevom periodu

Multimedijalni institut
ISBN 978-953-7372-28-6

CIP zapis je dostupan u računalnome katalogu
Nacionalne i sveučilišne knjižnice u Zagrebu pod brojem 000910180.

Zagreb, srpanj 2015.

Kazimir Majorinc

Moćan koliko je god moguće

Glavne ideje Lispa u McCarthyjevom periodu

Multimedijalni institut

Sadržaj

Moćan koliko je god moguće

Glavne ideje Lispa
u McCarthyjevom periodu

1.

Uvod

> *"In developing LISP our first goal is to describe a language which is as powerful as possible from the point of view of the programmer."*[1]

Većina programera se prvi puta susreće s Lispom preko nekog od brojnih citata i aforizama u kojima se, ponekad i preko mjere, ističu ljepota, elegancija i moć jezika. Tako se, primjerice, razumijevanje Lispa naziva prosvjetljenjem; tvrdi se da iskustvo programiranja u Lispu čini programera boljim, čak i ako nikad u životu ne bude koristio Lisp. Jeziku se, doduše u šali, pripisuju mistična svojstva; bog (ili bogovi) su svijet napisali u Lispu, a Lisp zajednica se opisuje kao kult.

Odluči li programer proučiti Lisp, zacijelo će zaključiti da je Lisp začuđujuće moćan, ali jednako tako začuđujuće jednostavan programski jezik. Moć je manje upitna; želi li dizajner jezika stvoriti moćan jezik, uloži li dovoljno vremena i napora, teško da neće i uspjeti. Jednostavnost je čudna; ona ostavlja dojam da je Lisp *otkriven*, a ne stvoren. Sličan su dojam imali i članovi ARTIFICIAL INTELLIGENCE projekta na američkom MIT-ju koji su razvili Lisp, zamišljen kao moćan, ali praktičan i vrlo složen jezik. Suprotno onome što bi se moglo očekivati, Lisp je tijekom razvoja postajao sve jednostavniji. Tek kad je glavni dizajner, John McCarthy napisao članak kojim je Lisp predstavljen javnosti, članovi projekta su prepoznali Lisp kao "predmet ljepote" i sam vrijedan proučavanja.

Narednih desetljeća, neki članovi Lisp zajednice su pokušali jezik učiniti još općenitijim, moćnijim, jednostavnijim, korektnijim opisom načina na koji programeri razmišljaju. Drugi, brojniji pokušaji unapređenja potaknuti su željom da se jezik učini efikasnijim, praktičnijim, popularnijim, možda čak i profitabilnijim. Čak i kad su radili kompromise, dizajneri su nastojali identificirati, zadržati, možda i unaprijediti bitne kvalitete jezika. Lisp je utjecao na druge programske jezike. U manjoj mjeri, i drugi programski jezici su utjecali na Lisp. Različiti motivi, prioriteti i rješenja doveli su do raspada i fragmentacije zajednice i razvoja brojnih, znatno različitih dijalekata i zajednica okupljenih oko dijalekata.

1 McCarthy, *Programs in Lisp,* AIM-012, 1959., str. 4.

U uvjetima fragmentacije zajednice i postojanja brojnih tumačenja osnovnih ideja, onaj tko želi razumjeti Lisp teško može izbjeći povijesni pristup; upoznavanje s idejama u obliku koji su imale kad su nastajale. Najzanimljiviji period je pri tome, u pravilu, onaj najraniji. Ova knjiga pokušava izložiti nastajanje i razvoj glavnih ideja Lispa tijekom prvih nekoliko godina u kojima je John McCarthy vodio razvoj jezika.

Pokušaja sistematičnog izlaganja glavnih ideja Lispa u ranom periodu je već bilo. Sam je McCarthy održao nekoliko predavanja i napisao nekoliko članaka o toj temi. Ranu povijest Lispa je daleko najviše istraživao Herbert Stoyan. Osim Stoyanove knjige *LISP, Anwendungsgebiete, Grundbegriffe, Geschichte* iz 1980., izdane samo na njemačkom jeziku i danas teško dostupne, svi radovi o povijesti Lispa pisani su u obliku za čije je razumijevanje potrebna već dobra upućenost u Lisp. Ova knjiga pisana je tako da bude razumljiva svakom programeru s malo iskustva u bilo kojem programskom jeziku.

Tijekom pisanja knjige, autor je održao nekoliko desetaka izlaganja u Hacklabu Mama u Zagrebu. Diskusije s članovima Hacklaba su u velikoj mjeri utjecale na sadržaj i oblik knjige.

U izvornim dokumentima, ime Lisp je obično pisano velikim slovima: LISP. U knjizi se koristi danas uobičajen i praktičniji način pisanja: Lisp. Izvorni način pisanja zadržan je samo pri citiranju.

2.

Rani utjecaji na McCarthyja

U razvoju Lispa sudjelovale su stotine ljudi. Ipak, autorom jezika se bez iznimke smatra Amerikanac John McCarthy (1927-2011), istraživač umjetne inteligencije. Lisp, programski jezik pogodan za rješavanje problema u oblasti umjetne inteligencije jedan je od McCarthyjevih prvih projekata.

2.1. TEHNOLOŠKI ENTUZIJAZAM

McCarthyja se često smatra vizionarem,[2] čovjekom čije je djelo uvelike okrenuto budućnosti. Među mnogim projektima, napisao je i održavao web stranice o futurističkim temama.[3]

Svoje znanstvene i životne interese McCarthy je smatrao posljedicom odgoja. Majka Ida i otac Jack bili su komunistički aktivisti među kojima je tada vladalo povjerenje u znanost, tehnologiju i nezadrživi napredak čovječanstva. Djeca su čitala sovjetsku popularno znanstvenu literaturu a McCarthy je posebno volio knjigu Mikhaila Ilina *100 000 Whys*.[4] Politička uvjerenja, a i povjerenje u znanost roditelji su uspješno prenijeli na Johna. Slične interese McCarthy je primjetio i kod druge djece odgojene u komunističkim obiteljima. McCarthy je sam sebe smatrao "radikalnim optimistom:" vjerovao je da će ishod biti dobar čak i ako ljudi ne slušaju njegove savjete.[5]

2.2. KONAČNI AUTOMATI I EKSPLICITNE REPREZENTACIJE ČINJENICA

McCarthy je bio iznimno dobar učenik. Srednju školu završio je dvije godine prije roka. Studij matematike započeo je 1944. i odmah upisao treću godinu, ali je izbačen sa studija zbog izostajanja s tjelesnog odgoja. Nakon kraće pauze, ipak mu je odobren nastavak studija.[6]

2 Lord, *John McCarthy has passed*, 2011.

3 McCarthy, *Progress and its sustainability*, 1995.

4 Shasha & Lazere, *Out of their minds ...*, 1995., str. 23.

5 McCarthy Susan, *What your dentist doesn't want you to know*, 2012.

6 Nilsson, *John McCarthy 1927-2011*, 2012., str. 3.

Konferencija *Hixon Symposium fon Cerebral Mechanisms in Behaviour* u Pasadeni, 1948., a posebno predavanje John von Neumanna o automatima koji bi se mogli reproducirati, mutirati i evoluirati[7] zainteresirali su McCarthyja za umjetnu inteligenciju.[8, 9]

Prva značajna McCarthyjeva ideja je predstavljanje *inteligentnog bića i njegove okoline* konačnim automatima. Unatoč von Neumannovoj preporuci da objavi svoje ideje, McCarthy odustaje zbog nemogućnosti predstavljanja činjenica o okolini u konačnom automatu.[10] Do završetka studija istraživao je parcijalne diferencijalne jednadžbe. Od 1951. do 1958. promijenio je nekoliko radnih mjesta, uglavnom na visokoobrazovnim institucijama.

Drugu značajnu ideju McCarthy je razvio 1952. Problemi su, smatrao je, određeni funkcijom koja provjerava rješenja problema. Tada se rješenje problema može naći primjenom inverza te funkcije.[11] McCarthy je objavio jedan članak, ali je naišao na iste probleme kao i s prethodnom idejom.[12] Zakratko i posljednji put, vraća se diferencijalnim jednadžbama.

7 von Neumann, *The general and logical theory of automata*, 1951.

8 McCarthy, *The logic and philosophy of artificial intelligence*, 1988., str. 2.

9 "Q. What is artificial intelligence? A. It is the science and engineering of making intelligent machines, especially intelligent computer programs."
McCarthy, *What is Artificial intelligence?*, 2007., str. 2.

10 "It considered a brain as a finite automaton connected to an environment also considered as an automaton. To represent the fact that the brain is uncertain about what the environment is like, I considered an ensemble (i.e. a set with probabilities) of environment automata. Information theory applied to this ensemble permitted defining an entropy at time 0 when the brain was first attached to the environment and later when the system had run for a while and the state of the brain was partially dependent on which environment from the ensemble had been chosen. The difference of these entropies measured how much the brain had learned about the environment."
McCarthy, *The logic and philosophy of artificial intelligence*, 1988., str. 2.

11 McCarthy, *The inversion of functions defined by Turing machines,* 1956., str. 177.

12 McCarthy, *The logic and philosophy of artificial intelligence*, 1988., str. 3.

Prvi su neuspjesi uvjerili McCarthyja da inteligentni stroj mora procesirati eksplicitne reprezentacije činjenica.[13] Ta je spoznaja temelj kasnijeg McCarthyjevog rada, "logičke umjetne inteligencije" i jasna inspiracija za razvoj Lispa.

2.3. DIGITALNA RAČUNALA KAO INTELIGENTNI STROJEVI

Digitalna računala mogu oponašati sve strojeve s konačnim brojem stanja. Ako su inteligentni strojevi uopće mogući, to će zacijelo biti digitalna računala opremljena odgovarajućim programima. Ta se ideja danas čini nepotrebnom trivijalnošću, ali prvim istraživačima nije bila očigledna. Alan Turing počeo ju je zastupati 1947.,[14] a objavio tek 1950.[15] McCarthy je do istog zaključka došao u ljeto 1955., radeći u IBM-ovom centru u Poughkeepsieju pod vodstvom Nathaniela Rochestera, konstruktora računala.[16]

13 "McCarthy thought that even if the "brain automaton"could be made to act intelligently, its internal structure wouldn't be an explicit representation of human knowledge. He thought that somehow brains did explicitly represent and reason about "knowledge," and that's what he wanted computers to be able to do." Nielsson, *John McCarthy 1927-2011*, 2012., str. 4.

14 Turing, *Lecture to the London Mathematical Society on 20 February 1947.*

15 Turing, *Computing machinery and intelligence*, 1950.

16 McCarthy, *The philosophy of AI and AI of philosophy*, 2008., str. 713.

3.

Dartmouthški projekt

McCarthy, tada zaposlen u DARTMOUTHU COLLEGE, njegov poznanik sa studija i izumitelj "neuralnih mreža" Marvin Minsky, Rochester, utemeljitelj teorije informacija Claude Shannon i nepotpisani Oliver Selfridge[17] napisali su u kolovozu 1955. "A proposal for the Dartmouth summer research project on artificial intelligence" te ga uputili mogućem financijeru. Atraktivna i pretenciozna, prethodno gotovo[18] nekorištena fraza "umjetna inteligencija" koju je smislio upravo McCarthy[19] uskoro je prihvaćena kao ime za cijelo područje računalne tehnologije.

3.1. JEZIK INTELIGENTNIH RAČUNALA

Predlagači su vjerovali da računalo može oponašati svaki aspekt ljudske inteligencije i najavili pokušaje rješavanja mnogih teških problema. Naveli su i pojedinačne interese i planove. McCarthy je smatrao da je za razvoj inteligentnih strojeva nužno primijeniti standardne metode pokušaja i pogreške na "višem nivou apstrakcije". Kao što ljudi koriste jezik za rješavanje složenih problema postavljajući pretpostavke i isprobavajući ih, to bi činila i inteligentna računala. Namjeravao je razviti i jezik pogodan za takvu upotrebu.[20] Već razvijeni formalni jezici lako se opisuju neformalnom matematikom i neformalna matematika se lako u njih prevodi, a lako se i provjerava korektnost dokaza. Jezik inteligentnih računala trebao bi imati i neke prednosti prirodnih jezika: konciznost, univerzalnost (unutar prirodnog jezika se može definirati i prigodno koristiti bilo koji jezik), mogućnost samoreferencije i izražavanja pretpostavki.

Vjerojatno iz istog perioda sačuvan je kratak, nedatiran McCarthyjev rukopis *The programming problem* koji

17 McCarthy, *The logic and philosophy of artificial intelligence*, 1988., str. 3.

18 "Do not develop your artificial intelligence, but develop that intelligence which is from God. From the latter results virtue; from the former, cunning."
Giles, *Chuang Tzu – mystic, moralist and social reformer*, 1889., str. 232.

19 Andresen, *John McCarthy: father of AI*, 2002., str. 84.

20 McCarthy et al., *A proposal ...* , 1955., str. 10.

sadrži gotovo iste ideje, ali ističe i potrebu da jezik bude eksplicitan: ne smije ostati mogućnosti za različito tumačenje značenja procedure.[21]

Lisp se može vidjeti kao pokušaj ostvarenja McCarthyjeve ambicije iz 1955.

3.2. IZMEĐU IPL-A I FORTRANA

Ljetni projekt je prihvaćen i ostvaren sljedeće, 1956. godine. Unatoč sudjelovanju desetak najpoznatijih istraživača, do očekivanih unapređenja nije došlo. Razloge relativnog neuspjeha McCarthy je kasnije objasnio nedostatkom financijskih sredstava, slabom suradnjom istraživača koji su se držali vlastitih projekata i težinom problema koje su predlagači podcijenili. Minsky je predstavio ideju za dokazivač teorema u geometriji, Ray Solomonoff je započeo rad na složenosti algoritama (engl. *algorithmic complexity*) a Alex Bernstein je predstavio program za šah. Umjesto rada na jeziku, McCarthy je predstavio "alfa-beta heuristiku" za igre poput šaha.[22]

Ako i nije razvio najavljeni jezik, McCarthy se upoznao s radom Allena Newella, Cliffa Shawa i Herberta Simona koji su predstavili program LOGIC THEORIST (LT) pisan u INFORMATION PROCESSING LANGUAGE (IPL).[23] IPL je podržavao jednostruko vezane liste i rekurzije. Naredbe su bile pozivi potprograma i nisu se mogle direktno komponirati. McCarthy je tada osjetio potrebu, čak žudnju, za "algebarskim jezikom" u kojem bi se izrazi pisali kao u matematici ili Fortranu, ali koji bi, poput IPL-a, omogućavao procesiranje lista i rekurziju. Takav bi jezik znatno pojednostavnio analizu formula i izdvajanje podformula u odnosu na IPL.[24]

21 Stoyan, *Early LISP History (1956-59)*, 1984., str. 300.

22 McCarthy, *Dartmouth and beyond*, 2006.

23 Newell & Shaw, *Programing the Logic Theory Machine*, 1957.

24 McCarthy, *History of LISP*, 1981., str. 174.

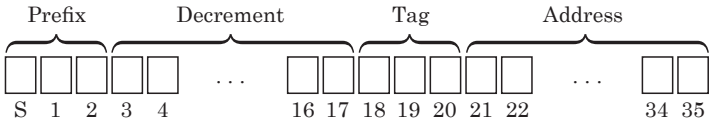
4.

FLPL

Jedva pola godine kasnije, McCarthy je dobio priliku sudjelovati u stvaranju jezika kakvog je želio. Početkom 1957. Rochester je pokrenuo projekt GEOMETRY THEOREM MACHINE prema spomenutoj ideji Minskog. Voditelj projekta bio je Herbert Gelernter, a McCarthy je bio konzultant. McCarthy je predložio[25] da se umjesto planirane implementacije IPL za računalo IBM 704 koristi Fortran, razvijen za numerička, ali smatran upotrebljivim i za logička izračunavanja.[26] "Ugnježđavanje poziva funkcija" omogućuje opis vrlo složenih operacija nad brojevima jednom naredbom u Fortranu. Gelernter i McCarthy si međusobno pripisuju otkriće da se tako može procesirati i liste.[27, 28] Fortran proširen mnoštvom specijalnih funkcija smatran je novim jezikom i nazvan "*FORTRAN-compiled list-processing language*" – FLPL. McCarthy je surađivao na projektu do jeseni 1958.

4.1. OPERACIJE NAD RIJEČIMA

Memorija računala IBM 704 podijeljena je u riječi ("registre") od 36 bitova. Bitovi su podijeljeni u grupe nazvane po najčešćem korištenju u mašinskom jeziku; *prefix* sadrži bitove S, 1 i 2; *decrement* sadrži bitove 3–17; *tag* sadrži bitove 18–20; konačno, *address* sadrži bitove 21–35.[29]



SLIKA 1. Grafički prikaz riječi računala IBM 704.

25 Gelertner et al., *A Fortran-compiled list-processing language*, 1960., str. 88.

26 Backus et al., *The Fortran – automatic coding system for the IBM 704*, 1956., str.2.

27 Gelertner et al., *A Fortran-compiled list-processing language*, 1960., str. 88.

28 McCarthy, *An algebraic language* ..., AIM 001, 1958., str. 3.

29 *704 electronic data-processing machine – manual of operation*, IBM, New York, 1955.

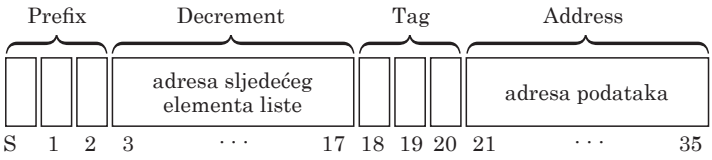
Neke od funkcija definiranih u FLPL-u su izdvajale dijelove riječi. Primjerice, ako je *j* adresa riječi u memoriji računala, onda su pozivi funkcija `XCPRF(`*j*`)`, `XCDRF(`*j*`)`, `XCTRF(`*j*`)` i `XCARF(`*j*`)` vraćali vrijednost sadržanu u odgovarajućim dijelovima riječi na adresi *j*: *prefix*, *decrement*, *tag*, *address* redom. Neke funkcije su definirane kao kompozicije funkcija `XCDR` i `XCAR`. Primjerice, `XCDADF(`*j*`)` je ekvivalent `XCDRF(XCARF(XCDRF(`*j*`)))`.

Druge funkcije su, obratno, upisivale vrijednosti u riječi. Primjerice, pozivi funkcija `XSTORAF(`*j*`,`*k*`)` i `XSTORDF(`*j*`,`*k*`)` su upisivali vrijednost *k* u adresni ili dekrementni dio riječi na adresi *j*.

4.2. LISTE

Podrška procesiranju lista u FLPL-u je prilagodba podrške listama u IPL-u za računalo IBM 704. Autori FLPL-a govore o "*NSS memoriji*" i "*NSS listama*"[30], gdje su NSS inicijali Newella, Simona i Shawa, autora IPL-a. Procesiranje lista se svodi na procesiranje pojedinačnih riječi u memoriji.

Liste u FLPL-u se sastoje od jednostavnijih "*elemenata liste*". Relativno dugačka riječ IBM 704 računala omogućivala je predstavljanje elemenata liste u jednoj riječi računala. U *adresnom dijelu* se nalazi adresa podatka u memoriji. U *dekrementnom dijelu* se nalazi adresa sljedećeg elementa liste ili 0 — ako sljedeći element ne postoji.



SLIKA 2. Element liste

Elementi liste u pravilu nisu smješteni u memoriji jedan iza drugoga.

30 Gelernter et al, *A Fortran-compiled list processing language*, 1959., str. 37-1-2.

Adresa riječi	Dekrementni dio riječi	Adresni dio riječi	
1001	1004	2005	← 1. element liste
1002	0	2003	← 3. element liste
1004	1002	2006	← 2. element liste
	...		
2003	3.14259		← 3. podatak u listi
2005	0.71412		← 1. podatak u listi
2006	2.71828		← 2. podatak u listi

SLIKA 3. Lista (0.71412, 2.71828, 3.141259) na adresi 1001.

Radi optimizacije, podaci koji stanu u petnaest bita mogu se spremiti i direktno u adresni dio riječi. U FLPL-u, liste su samo apstrakcije kojima se programeri služe, ne poseban tip podataka. Funkcije za procesiranje lista kao argumente primaju adrese prvog elementa liste.

Tako predstavljene liste su prilagodljive, promjenjive duljine i omogućuju brzo umetanje i uklanjanje podataka. S druge strane, nije moguć direktan pristup, primjerice, stotom podatku u listi nego do podatka valja doći postepeno, u sto koraka.

4.3. KREIRANJE LISTA

Slobodna memorija se na početku izvršavanja programa nalazi spremljena u posebnoj, internoj listi *lavst* (engl. *list of available storage*). Iz *lavst* se uzimaju riječi potrebne za formiranje lista i odlažu riječi koje više nisu potrebne za izvršavanje programa.

Najvažnija funkcija za kreiranje lista je `XLWORDF`. Primjerice, poziv funkcije `XLWORDF(1,2,3,5)` uklanja riječ iz *lavst*, upisuje 1, 2, 3 i 5 u odgovarajuće dijelove riječi (*prefix*, *decrement*, *address* i *tag* redom) te vraća, kao vrijednost, adresu riječi. Postojale su i druge, slične funkcije. `XLWORDF` se, kao funkcija, mogla komponirati s drugim funkcijama, što u ono vrijeme nije bila očigledna ideja.[31]

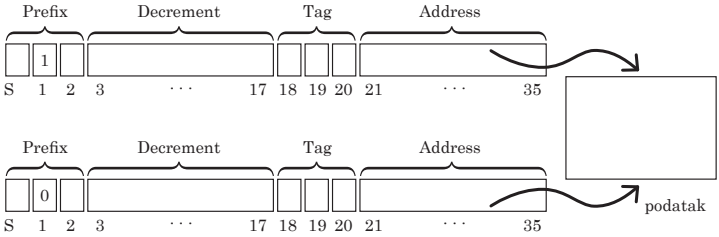
31 McCarthy, *The logic and philosophy of AI*, 1988., str. 4.

Posebno, elementi liste mogu sadržavati, kao podatke, i druge liste. Tako nastaju složenije *strukture liste* (engl. *list structures*) — naziv preuzet iz IPL-a. Primjerice, izraz

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

bi prilikom izračunavanja kreirao strukturu liste i vratio je kao rezultat.

Kako se u "adresnom dijelu" elementa liste nalazi adresa podatka, moguće je da više različitih lista sadrže iste podatke. Ta korisna mogućnost čini neke operacije nad listama složenijima. Želi li se, primjerice, izbaciti podatak iz liste, nije jasno može li se osloboditi memoriju koju taj podatak zauzima, jer je moguće da je podatak još uvijek sadržan u nekoj drugoj listi. Rješenje primijenjeno u FLPL-u je uvođenje svojevrsnog vlasništva nad podacima. Ako se u prvom bitu elementa liste nalazi 1, onda brisanjem elementa liste valja obrisati i podatak. Ako se u prvom bitu elementa liste nalazi 0, onda je podatak "posuđen" i brisanje elementa liste ne uključuju i brisanje podatka.



SLIKA 4. Dva elementa liste koji sadrže isti podatak. Donji element liste "posuđuje" podatak.

Vodeći računa o prethodnom, poziv funkcije `XERASEF(`*j*`)` briše element liste na adresi *j*, a poziv funkcije `XTOERAF(`*j*`)` briše cijelu listu na adresi *j*; brišu se i odgovarajući podaci – ako je tako određeno sadržajem bita 1.

Primjer FLPL programa je program koji ispituje članstvo u listi.[32]

32 Stoyan, *List processing*, 1992., str. 151.

```
FUNCTION MEMBER(X, L)
  LI=L
1 IF LI=0 2, 4, 2
2 EL=XCARF(LI)
  IF EL=X 3, 5, 3
3 LI=XCDRF(LI)
  GOTO 1
4 MEMBER=0
  GOTO 6
5 MEMBER=1
6 RETURN
  END
```

FLPL nije podržavao rekurzivne funkcije. Autori jezika su vjerovali da se rekurzija može emulirati spremanjem međurezultata u liste ili čak tako da pozvana funkcija treba modificirati program koji ju je pozvao,[33] ali se pri pisanju GEOMETRY THEOREM MACHINE nije za time pokazala potreba.

Uočena je i sličnost između izraza koji procesiraju liste i lista samih, ali nije jasno jesu li tu sličnost autori jezika nekako iskoristili.[34]

McCarthy je u ljeto 1958. neuspješno pokušao napisati program za diferenciranje izraza u FLPL-u. Osim nedostatka podrške za rekurziju[35], smetalo mu je i nezgrapno grananje toka programa po kojem je rani Fortran danas poznat. Kako je Gelernterova grupa bila zadovoljna FLPL-om McCarthy je zaključio da mora razviti nov jezik.[36] Mnogi elementi FLPL-a su našli mjesto u Lispu.

33 McCarthy, *History of Lisp*, 1981., str. 189.

34 "The authors have since discovered a further substantial advantage of an algebraic list-processing language ... It is the close analogy that exists between the structure of an NSS list and a certain class of algebraic expressions that may be written within the language."
Gelernter et al., *A Fortran-compiled list-processing language*, 1959., str. 37-3.

35 "If FORTRAN had allowed recursion, I would have gone ahead using FLPL. I even explored the question of how one might add recursion to FORTRAN. But it was much too kludgy."
Shasha & Lazere, *Out of their minds* ..., 1998., str. 27

36 McCarthy, *History of Lisp*, 1981., str. 176.

5.

Prijedlog kompajlera

Nakon Ljetnog projekta, McCarthy se zaposlio na MASSACHUSETTS INSTITUTE OF TECHNOLOGY (MIT). Krajem 1957., nakon početnih iskustava s FLPL-om, uputio je voditelju računskog centra dvadesetak strana dug memorandum *A proposal for a compiler*. Predloženi kompajler bio je iznimno ambiciozan, zanimljiv i bogat mogućnostima. Ipak, nije imao jedinstvenu koncepciju niti ljepotu kakvu će Lisp, posebno "čisti Lisp" imati.

U nekim dijelovima *Prijedloga*, McCarthyjevo razmišljanje je apstraktno i možda ne sasvim precizno. Razvoj jezika je, čini se, uskoro zaustavljen. Najavljeni nastavci *Prijedloga kompajlera* nikad nisu napisani.

5.1. PROGRAMSKI JEZIK KAO KOORDINATNI SUSTAV

McCarthy ponovo propituje što je dobar programski jezik. Na tragu razmatranja iz *Prijedloga Darmouthškog projekta*, programski jezik mora biti kombinacija prirodnog i matematičkog jezika. Sam prirodni jezik nije dovoljno precizan, a postojeći matematički jezik izražava deklarativne umjesto potrebnih imperativnih rečenica i ne omogućuje definiranje. Za definiranje se koristi prirodni jezik.

Programski jezik se, po McCarthyju, može promatrati kao koordinatni sustav. Program je određen kombinacijom "varijabli", pri čemu se ne misli na simbole s vrijednostima, kao inače u kontekstu programskih jezika, nego na različite "atribute" ili "aspekte" programa, po mogućnosti takve da se željene promjene u programu mogu postići promjenom što manjeg broja varijabli. Primjerice, tipografske konvencije i same naredbe u jeziku su za McCarthyja varijable. Postoje četiri vrste varijabli:

1. *varijable sustava*, čije vrijednosti ne može mijenjati ni programer ni program, nego se mijenjaju ako se mijenja programski sustav;

2. *varijable programa*, čije vrijednosti određuje programer, a koje se ne mogu mijenjati tijekom izračunavanja;

3. *varijable segmenta programa*, čije vrijednosti mogu biti različite u različitim dijelovima programa i

4. *varijable izračunavanja* (engl. *computation variables*) koje mijenjaju vrijednost tijekom rada programa.

Sistem postaje moćniji ako, varijable postaju lakše promjenjive. Primjerice, "tipografske konvencije" pa i same naredbe (engl. *statements*) trebaju biti "varijable izračunavanja", tj. mogu se mijenjati tijekom izvršavanja programa.[37]

5.2. SLOBODA PROGRAMERA

Najvažnije svojstvo predloženog programskog jezika je, po samom McCarthyju, sloboda programera pri definiranju novih izraza.[38] Naredbe ekvivalencije (engl. *equivalence statements*) bi omogućile uvođenje pokrata za bilo koji izraz. Kompajler bi se mogao mijenjati i proširivati naredbama programa koji se kompajlira. Najavljeni su i elementi deklarativnog programiranja.[39] Programi bi mogli generirati i kompajlirati druge programe i mijenjati kod kompajlera, napisan u istom programskom jeziku. Posebno, mogli bi kompajlirati i interpretere, a onda i generirati kod koji bi ti interpreteri izvršavali.

5.3. LOGIČKE VRIJEDNOSTI I FUNKCIJE

Predloženi jezik bi podržavao logičke vrijednosti (u originalu *propositional quantities*) 0 i 1 i uobičajene iskazne

37 "The statements themselves … are computational variables here since the program can generate source language program in the course of operation and can call in the compiler to compile it." McCarthy, *A proposal for a compiler*, 1957., str. 4.

38 "The most important feature of the source language of this system is the freedom it gives the programmer to define new ways of expressing himself."
McCarthy, *A proposal for a compiler*, 1957., str. 4.

39 "The ability to describe a computation by giving final state of the machine in terms of the initial state without haying to worry about intermediate changes to the variabless used in the computation."
McCarthy, *A proposal for a compiler*, 1957., str. 5.

operatore, uključujući implikaciju i ekskluzivnu disjunkciju. Primjerice, bili bi mogući izrazi poput

```
P = Q ∧ ((A = B) ∨ P).
```

5.4. FUNKCIJA IF

Važna inovacija je funkcija `IF`, ugodnija za rad od istoimene naredbe u Fortranu. Primjerice, izraz

```
A = IF(P,X+Y: Q,U+V: (A=B),A+B: OTHERWISE,B)
```

je ekvivalentan naredbi u suvremenom pseudokodu

if P **then** $X + Y$
else if Q **then** $U + V$
else if $A = B$ **then** $A + B$
else B

Također, ekvivalentan je kasnijim uvjetnim izrazima u Lispu.

5.5. SINTAKSA

Sintaksa predloženog jezika je neobična i zanimljiva, iako samo grubo naznačena. Karakteristična je podjela programa u dva do četiri stupca. Primjerice, program

```
X | Y
Y | X
```

je predstavljao "paralelnu" naredbu za pridruživanje te bi zamijenio vrijednost varijablama `X` i `Y` bez uobičajene upotrebe treće, privremene varijable. Program u suvremenom pseudokodu

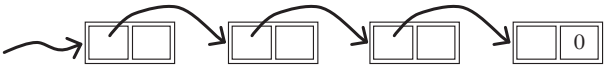
for j **in** L **do**
  if $B[\,j\,] > 0$ **then** $A[B[\,j\,]+C[\,j\,]] := R[\,j\,] \cdot S[\,j\,]$
end.

bi u *predloženom jeziku* bio zapisan kao

Quantifier	Quantity	Condition	Value
j ∈ L	A(B(J)+C(J))	B(J)>0	R(J)*S(J).

5.6. LISTE

Jezik bi podržavao i algebarske izraze, logičke vrijednosti i operacije, "short-circuit" izračunavanje, jednostruko vezane liste, poznate iz FLPL-a. U dokumentu se nalazi i prvi grafički prikaz lista:



SLIKA 5. Grafički prikaz liste.[40]

Definirane su i funkcije za izdvajanje dijelova riječi, kao u FLPL-u, samo s pojednostavljenim imenima poput `CAR` i `CDR`.

5.7. FUNKCIJE S VIŠESTRUKIM VRIJEDNOSTIMA I KOMPOZICIJA FUNKCIJA

Osim neuobičajene sintakse, posebnost prijedloga je podržavanje funkcija s višestrukim vrijednostima. Dijeljenje s ostatkom je možda najjednostavniji primjer potrebe za takvom funkcijom. Definirana je i kompozicija funkcija, s istom oznakom kao i u matematici. Opis tih mogućnosti u prijedlogu je prilično štur, ali primjeri su dovoljno ilustrativni.

Funkcija `PI` preraspoređuje vlastite argumente. Primjerice, vrijednosti

```
(PI(1,2,2,3))(X,Y,Z)
```

40 Prema ilustraciji iz McCarthy, *A proposal for a compiler*, 1957., str. 15.

su vrijednosti `X`, `Y`, `Y` i `Z` redom. Funkcija `PI` je korisna za pisanje izraza koji uključuju funkcije s više vrijednosti. Isto vrijedi i za `I1`, funkciju identiteta jedne varijable i jedne vrijednosti.

Primjerice, neka je `Q(A,B,C)` funkcija koja ima dvije vrijednosti: rješenja kvadratne jednadžbe $\mathtt{A} \cdot x^2 + \mathtt{B} \cdot x + \mathtt{C}$. Neka je `PLUS` funkcija koja zbraja proizvoljan broj argumenata. Izraz koji izračunava sumu obaju rješenja jednadžbe $\mathtt{A} \cdot x^2 + \mathtt{B} \cdot x + \mathtt{C}$ i koeficijenata `A` i `C` može biti

```
(PLUS ∘ (Q,I1,I1) ∘ PI(1,2,3,1,3))(A,B,C).
```

McCarthy nije pokušao objasniti kako bi takve funkcije mogle biti korisne i na ideju funkcija s višestrukim vrijednostima se više nije vratio.

6.

Prijedlog programskog jezika

U okviru međunarodne inicijative za stvaranjem "univerzalnog programskog jezika" koja će kasnije iznjedriti programski jezik Algol, američka *Association for Computing Machinery* (ACM) početkom 1958. osnovala je *Ad hoc komitet za programske jezike*. KOMITET je odlučio da novi jezik treba biti viši, "algebarski programski jezik." Fortran je već zadovoljavao taj uvjet, ali kao intelektualno vlasništvo IBM-a nije bio prihvatljiv. Podkomitet čiji su članovi, pored McCarthyja, predstavnika MIT bili i autori tada aktualnih programskih jezika John Backus, Alen Perlis i William Turanski sastavio je *Prijedlog programskog jezika*, polaznu točku delegacije ACM na sastanku u Zürichu, u ljeto 1958.[41] *Prijedlog programskog jezika* nije bio upadljivo ambiciozan kao godinu dana stariji *Prijedlog kompajlera,* ali je sadržavao nekoliko važnih ideja.

6.1. UVJETNA NAREDBA

Umjesto nezgrapne naredbe uvjetnog grananja u Fortranu, *uvjetna naredba* (engl. *conditional statement*) imala je oblik

$$p_1 \to e_1, p_2 \to e_2, \ldots, p_n \to e_n$$

gdje su p_1, ..., p_n logički izrazi, a e_1, ..., e_n bilo koje naredbe i izvršavala se tako da se izračunavaju p_1, ..., p_n dok se ne pronađe prvi istinit izraz p_i. Tada se izvrši pripadajuća naredba e_i.

Ovdje, za razliku od *Predloženog kompajlera*, uvjetna naredba nije izraz.

6.2. GO TO NAREDBA

Naredbe su mogle imati imena. Imena su simboli, nizovi znakova koji počinju slovom. Primjerice, `TR` je ime naredbe

```
(TR) X = 2 + 2.
```

41 Perlis, *The American side of the development of ALGOL*, 1981., str. 77.

Vrlo izražajna je bila naredba

```
GO TO e.
```

"*Designacijski izraz*" *e* mogao je imati nekoliko oblika. Najjednostavnije, mogao je biti simbol korišten kao ime u programu. Primjerice,

```
GO TO TR.
```

Mogao je biti i oblika *s*(*I*), gdje je *s* simbol, a *I* izraz koji se izračunava u broj. Primjerice, postoji li u programu "deklarativna naredba"

```
SWITCH Q(A, B, C, D, E)
```

tada se nakon

```
GO TO Q(2+2)
```

program nastavlja izvršavati od naredbe s imenom `D`. Konačno, izraz *e* mogao je imati oblik `(`*c*`)` gdje je *c* svojevrsni izraz uvjetnog grananja, primjerice,

```
GO TO (D<0 → NOSOLUTIONS,
       D=0 → ONESOLUTION,
       D>0 → TWOSOLUTIONS).
```

6.3. MAKRO NAREDBA

Predviđena je i neka vrst *makro naredbe*. Primjerice, izvršenjem naredbe

```
LABEL P(A, B)
```

dio programa između naredbi označenih imenima `A` i `B` postaje vijednost `P`. Tada naredba

```
P(L1 → X, L2 → Y)
```

izvršava dio programa `P`, ali uz privremenu zamjenu simbola `X` i `Y` simbolima `L1` i `L2`.

McCarthy nije nazočio sastanku u Zürichu, a u konačni oblik jezika, danas poznatog kao Algol 58, ušli su neki od izloženih prijedloga. Prihvaćen je uvjetni `GO TO`, dok je funkcionalnost `LABEL` izraza prenesena na drugu naredbu. McCarthyju drage[42] uvjetne naredbe nisu prihvaćene.

Tijekom svibnja 1958., McCarthy je na MIT-ju održao predavanje pod naslovom *An algebraic coding system* i tom prilikom, prema bilješci studenta James R. Slaglea, govorio o proširenju Fortrana Churchovim lambda izrazima, kompozicijom funkcija i funkcijama s više vrijednosti.[41]

U ljeto 1958., u pismu Perlisu i Turanskom McCarthy predlaže promjene u *Prijedlogu*. Promjene su značajne i dalekosežne. Najveći napor u dizajnu sljedeće verzije jezika treba biti "mogućnost promjene jezika u jeziku samom".[43] Treba uvesti međujezik (engl. *intermediate language*) u prefix formi. Cijeli program treba biti jedan izraz međujezika.[44]

6.4. FUNKCIJE VIŠEG REDA

Imena funkcija trebaju biti varijable, tako da je, primjerice, moguće pisati

```
f = sin
g = f + cos
a = g(3).
```

Predlaže se definiranje funkcija višeg reda; zbrajanje, oduzimanje, množenje i dijeljenje aritmetičkih funkcija, diferenciranje i integriranje kao i mogućnost definiranja drugih funkcija višeg reda. Posebno, uvođenje kompozicije, tako da, primjerice,

```
(f ∘ g)(x) = f(g(x)).
```

42 Stoyan, *Early LISP history (1956-59)*, 1984., str. 300.

43 McCarthy, *Some proposals for the Volume 2 (V2) language*, 1958., str. 1.

44 Stoyan, *Early LISP history (1956-59)*, 1984., str. 303.

McCarthy ističe da izrazi u elementarnoj matematici, primjerice $x + y$ ponekad označavaju vrijednost, a ponekad funkciju. U programskom jeziku se takva dvosmislenost mora izbjegavati, pa McCarthy, možda prvi put u povijesti programskih jezika[45] predlaže uvođenje Churchove lambda notacije. Primjerice, izraz

```
lambda(x, y)(x + y)
```

označavao bi funkciju

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

Tako definirane funkcije bi se primjenjivale na brojeve, ali i na druge "forme". Primjerice, izraz

```
lambda(x, y)(x + y)(3, 4)
```

bi se izračunavao u broj `7`, a izraz

```
lambda(x, y)(x + y)(a + 1, b)
```

u izraz `(a + 1) + b`. Osnovne operacije nad formama također trebaju biti podržane.

McCarthy je, u manjoj mjeri, nastavio sudjelovati u razvoju Algola i kasnije, predlažući ideje koje je već bio primjenio u Lispu.[46]

45 Stoyan, *Early history of LISP (1956-59)*, 1984., sl. 22.

46 McCarthy, *On conditional expressions and recursive functions*, 1959.

7.

Imperativni Lisp

McCarthy i Minsky, tada zaposleni na američkom MIT-ju su u rujnu 1958.[47] započeli rad na projektu ARTIFICIAL INTELLIGENCE. Rad je relativno dobro dokumentiran objavljenim člancima i predstavljanjima na konferencijama i internim dokumentima, *Artificial Intelligence Memo* (AIM), *Research Laboratory of Electronic, Quaterly progress report* (RLE QPR) i studentskim radovima. "Ujak"[48,49] McCarthy je slijedeći primjere projekata LOGIC THEORY MACHINE – IPL i GEOMETRY THEOREM MACHINE – FLPL namjeravao razviti "ekspertni sustav" ADVICE TAKER[50] i programski jezik za "manipuliranje simboličkim izrazima" u kojem bi sustav bio napisan.[51] Nakon predstavljanja javnosti, iako Mccarthy od ADVICE TAKERA nije odustao,[52] razvoj projekta je zamro i jedva da se i spominje u brojnim internim i objavljenim dokumentima.[53,54] Programski jezik je, naprotiv, intenzivno razvijan.

Isprva samo opisivan kao "*an algebraic language for the manipulation of symbolic expressions*"[55] i "*symbol manipulating language*"[56], jezik je uskoro dobio ime "LISP (*List Processer*)"[57], i nešto kasnije, "LISP (*List Proces-*

47 Stoyan, *Early LISP history (1956-59)*, 1984., str. 304.

48 Levy, *Hackers*, 2010., str. 36.

49 "The teacher was a distant man with a wild shock of hair and an equally unruly beard — John McCarthy. A master mathematician, McCarthy was a classically absent-minded professor; stories abounded about his habit of suddenly answering a question hours, sometimes even days after it was first posed to him."
Levy, *Hackers*, 2010., str. 11.

50 McCarthy, *Programs with common sense*, 1959., str. 75-92.

51 McCarthy & Minsky, *Artificial Intelligence* in RLE QPR 052, 1959., str. 129.

52 McCarthy, *Situations, actions and causal laws*, 1963., str. 1.

53 "The main problem in realizing the Advice Taker has been devising suitable formal languages covering the subject matter about which we want the program to think."
McCarthy, *A basis for mathematical theory of computation*, 1963., str. 69.

54 Stoyan je nezavisno rekonstruirao ADVICE TAKER u *Programmiermethoden der Künstlichen Intelligenz*, 1988., str. 193-231.

55 McCarthy, *An algebraic language …*, AIM-001, 1958., str. 1.

56 McCarthy, *A revised definition of maplist*, AIM-002, 1958., str. 1.

57 McCarthy, *Revisions of the language*, AIM-004, 1959., str. 9.

sor)"[58]. Ponekad je korišteno i ime "List Processing Language[59]". Asocijacija imena LISP i "List Processer" je uskoro oslabila pa se poneķad piše da ime LISP potječe od "*List processing*"[60] ili se čak koristi termin "*LISP Processor*"[61]. Ime jezika se uglavnom pisalo velikim slovima, ali McCarthy povremeno koristi i danas češći oblik "Lisp".[62]

John McCarthy smatra se autorom jezika.[63] Članovi projekta su bili prvi "hakeri"[64] koji su radili "u atmosferi … neograničene ambicije i entuzijazma."[65] McCarthy nije priječio drugim članovima projekta ugrađivati vlastite ideje u jezik.[66] Steven "Slug" B. Russell je "bio kompajler":[67] osobno je prevodio McCarthyjeve Lisp programe u asembler, razvijao interpreter i sudjelovao u dizajnu jezika. Robert Brayton i David C. Luckham[68] su bili prvi studenti koji su radili na projektu, te su uspješno napisali prvi kompajler u asembleru. David M. R. Park je pomagao u pisanju kompajlera te je doprinio dizajnu jezika[69], kao i Rochester[70], tada privremeno zaposlen na MIT-ju Klim Maling je pisao kompajler u Lisp-u. Daniel J. Edwards je napisao prvi "garbage collector" i sudjelovao u rješavanju "funarg problema". Phyllis Fox napisala je Lisp I. priručnik. Prvi korisnici su bili Rochester, Slagle, Paul W. Abrahams, Louis Hodes, Seymor Z. Rubenstein te Solomon H. Goldberg. Konačno, Minsky, Dean Arden, Shannon, Har-

58 McCarthy, *Recursive functions* …, AIM-008, 1959., str. 1.

59 McCarthy et al., *Artificial intelligence* u RLE QPR 053, 1959., str. 122.

60 Berkeley i Bobrow, *LISP – its operations and applications*, 1964., str. 4.

61 Edwards, *Secondary storage* in LISP, AIM-063, 1963., str. 13.

62 McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 13-17.

63 Abrahams, *Discussant's remarks* in HOPL I, 1981., str. 192.

64 Levy, *Hackers*, 2010.

65 Minsky, *Introduction to COMTEX Microfische edition* …, 1983., str. 10.

66 McCarthy, *Guy Steele interviews John McCarthy, father of Lisp*, 2009.

67 Russell, *Adventures and pioneering with John*, 2012.

68 Osobna komunikacija s Braytonom i Luckhamom, 2012.

69 Osobna komunikacija s Luckhamom, 2012.

70 Rochester, AIM-005, 1959.

tley Rogers, Roland Silver i Alan Tritter su bili zainteresirani promatrači i komentatori.[71]

Prvi nacrt jezika opisan je u rujnu 1958. Uslijedio je postepeni, kontinuirani razvoj te veliki broj promjena do studenog[72] 1962. kad McCarthy zbog neslaganja s nadređenima oko razvoja "time sharinga"[73] odlazi raditi na sveučilište Stanford. Jedini iskorak iz kontinuiteta je "čisti Lisp", podskup stvarno implementiranog Lispa koji je McCarthy opisao u proljeće 1959. radi predstavljanja osnovnih ideja jezika. Iako je McCarthy želio nastaviti voditi razvoj Lispa[74], središte razvoja jezika ostalo je na MIT-ju.

7.1. imperativni i algebarski jezik

Već početkom rujna[75] McCarthy je napisao prvi memo, *An algebraic language for the manipulation of symbolic expressions*. Naslov je tek opis jezika koji tada još nije imao ime. Neki detalji su bolje opisani u kasnijim memoima, posebno u trećem.

Prvi dizajn jezika se prema McCarthyjevom opisu "*language of imperative statements*"[76] u ovoj knjizi naziva "imperativni Lisp". Taj je dijalekt kasnije McCarthy opisao kao "*a Fortran-like language with list structures*"[77], dakle, ne puno više od FLPL-a. "Imperativni Lisp" je bio, po shvaćanju samih članova tima, pragmatično dizajniran jezik.[78]

Fraza "algebarski jezik" je za McCarthyja označavala viši programski jezik u kojem je za razliku od asemblera i

71 Stoyan, *Early History of LISP (1956-1959)* (slides), 1984., str. 24-6.

72 Stoyan, *LISP, Anwendungsgebiete, Grundbegriffe, Geschichte*, 1980.

73 McCarthy, *An interview with John McCarthy*, 1989., str. 4.

74 "Maintenance and further development of LISP will be continued by Professor J. McCarthy, who is now at Stanford University. We plan to continue close association with his group."
Minsky, *Artificial intelligence*, RLE QPR 068, 1963., str. 159.

75 Stoyan, *Early LISP History (1956-1959)*, 1984., str. 304.

76 McCarthy, *Revisions of the language*, AIM-003, 1958., str. 1.

77 McCarthy, *The logic and philosophy of AI*, 1988., str. 5.

78 Abrahams, *Transcript of discussant remarks*, in McCarthy, *History of Lisp*, 1981., str. 193.

mašinskih jezika moguće pisati složene izraze. Prednosti takvih jezika još nisu bile općeprihvaćene, pa McCarthy objašnjava da su programi kraći i jednostavniji. Ističe da se "izlaz" jedne procedure može iskoristiti kao "ulaz" druge pa ne treba davati imena međurezultatima.[79]

Čini se da je McCarthy dvojio treba li jezik biti specijaliziran za simboličko procesiranje. Tako na samom početku Memoa[80] piše da jezik nije pogodan za predstavljanje "lista fiksne duljine" i pristupanje podacima drugačijim redoslijedom od onoga u listi što je krupan, neprihvatljiv nedostatak za programski jezik opće namjene. Ipak, već nakon nekoliko stranica McCarthy uvodi tip "polja" pa se čini da je ideja specijaliziranog jezika odmah odbačena.[81]

7.2. OSNOVNI TIPOVI I NAREDBE

Jezik podržava nekoliko uobičajenih tipova podataka. Aritmetika je trebala biti jednaka kao u Fortranu. *Cijele riječi* u memoriji računala su zaseban tip. *Iskazne vrijednosti* (u originalu *propositional quantities*) *istina* i *laž*, predstavljene su jednim bitom: 1 i 0. *Iskazni izrazi* su izrazi koji imaju iskazne vrijednosti. Funkcije koje vraćaju iskazne vrijednosti, primjerice `<` i `=` nazivaju se *predikati*. "Imperativni Lisp" je baziran na naredbama; posebno na naredbi za pridruživanje (orig. engl. *arithmetic* ili *replacement statement*), najvažnijoj naredbi za procesiranje lista.[82] Naredba za pridruživanje ima uobičajeni oblik,

79 McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 3.

80 "... this language is best suited for representing expressions whose number and length may change … It is not so convenient for representing lists of fixed length where one frequently wants the n-th element where n is computed rather than obtained by adding 1 to $n - 1$."
McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 2.

81 McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 7-8.

82 "Programs for manipulating list structure are written mainly in terms of replacement statements (i.e. of the form `a = b`)."
McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 11.

$$l = r,$$

pri čemu je *r* bilo koji izraz a *l* ime varijable ili "indeksirane varijable", primjerice `a=15` ili `A(i)=15`, ili poziv funkcije. Primjerice, `cwr(3)=15` upisuje vrijednost 15 u riječ na adresi 3.

Naredba za iteriranje `do`, preuzeta iz Fortrana, trebala je podržavati iteriranje kroz segmente cijelih brojeva, ali i kroz liste. Detalji u vrijeme pisanja memoranduma još nisu bili određeni.

Više naredbi se može organizirati u *složenu naredbu* (engl. *compound statement*) koristeći "vertikalne zagrade" `/` i `\`. Primjerice,

```
/ t = a
  a = b
\ b = t.
```

Uvjetni izrazi (engl. *conditional expressions*) koje valja razlikovati od uvjetnog grananja programa imaju oblik

$$(p_1 \to e_1,\ p_2 \to e_2, \ldots, p_n \to e_n)$$

gdje su p_1, ..., p_n iskazni izrazi a e_1, ..., e_n bilo koji izrazi. Redom se izračunavaju p_1, ..., p_n dok vrijednost jednog od njih, primjerice p_i, ne bude jednaka 1. Vrijednost e_i je tada vrijednost cijelog uvjetnog izraza. Ako niti jedan p_i nije istinit, onda se naredba koja sadrži uvjetni izraz ne izvršava.

Kontrola toka programa preuzeta je iz *Prijedloga ad hoc komiteta.* Mjesta u programu se označavaju simbolima i smatraju se vrijednostima posebnog, lokacijskog tipa. Tijek programa se prenosi naredbom `go(`*e*`)` gdje je *e izraz* koji se izračunava u lokacijsku vrijednost. Primjerice,

```
/       i = 0
  loop  i=i+1
\       go(loop).
```

Operacije nad lokacijskim vrijednostima su ograničene, no moguće je koristiti uvjetne izraze. Moguće je i set-naredbom, primjerice

`set(A;` q_1`, ...,` q_m`)`

definirati polje `A` koje sadrži lokacijske vrijednosti q_1, ..., q_m i onda koristiti naredbu `go(A(`*e*`))`, gdje je *e* izraz koji se izračunava u prirodni broj.

7.3. POTPROGRAMI I "FUNKCIJE"

Jezik uključuje dvadesetak već definiranih funkcija i potprograma, ali programeri mogu definirati i vlastite potprograme i funkcije. Definicija funkcija i potprograma se sastoji se od zaglavlja, primjerice

`subroutine eralis(J)` ili

`function copy(J)`,

iza kojeg slijedi jedna složena naredba. Izvršavanje potprograma i funkcija se prekida naredbom `return`. Funkcija vraća, kao rezultat izračunavanja, posljednju vrijednost izračunatu prije naredbe `return`. Primjerice,

```
function abs(x)
/  (x<0 → -x, x=0 → 0, x>0 → x)
\  return.
```

Funkcije se mogu definirati i jednostavnijim izrazima, primjerice,

`abs(x) = (x<0 → -x, x=0 → 0, x>0 → x).`[83]

Potprogrami i funkcije se pozivaju na uobičajen način, primjerice, `abs(-3)`, `copy(L)`.

Potprogrami i funkcije mogu biti *rekurzivni*, tj. pozivati sami sebe. Rekurzija je posebno korisna pri procesiranju lista. Ta je mogućnost "imperativnog Lispa" važno unapređenje u odnosu na FLPL.

83 Obje definicije funkcije `abs` su napisane za potrebe ove knjige. Svi McCarthyjevi primjeri u originalnom memou su presloženi da bi se koristili na ovom mjestu.

Funkcije su vrijednosti posebnog, *funkcijskog tipa* te mogu biti i argumenti u pozivima potprograma i funkcija. McCarthy piše da mogućnosti funkcijskog tipa nisu u potpunosti iskorištene u "ranom sistemu".[84]

Funkcije u "imperativnom Lispu" razlikuju se od uobičajenih matematičkih funkcija. Vrijednost funkcije za iste argumente može biti različita jer ovisi o stanju memorije. Funkcije mogu mijenjati vrijednost varijabli i memorijskih registara.

7.4. SIMBOLIČKI IZRAZI

Simbolički izrazi, čijem procesiranju je "imperativni Lisp" namijenjen, su nizovi znakova posebnog oblika, pogodnog za prevođenje matematičkih i logičkih izraza, ali i za procesiranje pomoću računala. Primjerice, matematički izraz

$$x \cdot (x + 1) \cdot \sin y$$

se može zapisati kao simbolički izraz

```
(times,x,(plus,x,1),(sin,y)).
```

Taj oblik podsjeća na tzv. *poljsku notaciju*, ali zagrade omogućuju korištenje funkcija s varijabilnim brojem argumenata, primjerice `times`. Simbolički izrazi se mogu koristiti kao podaci u programima za "manipuliranje rečenicama formalnog jezika", dokazivanje teorema, razvoj *Advice takera*, pojednostavljivanje formula, simboličko deriviranje i integriranje, kompajlerima, programima za procesiranje izraza čiji broj i duljina variraju na teško predvidljive načine.

Simbolički izrazi su definirani i formalnije. *Simboli* su nizovi od jednog ili više znakova. Posebno, i nizovi znakova koji predstavljaju brojeve su simboli. Primjerice, `times`, `x`, `plus`, `1`, `sin` i `y` su simboli.

84 McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 7.

1. Svi simboli su *simbolički izrazi*.

2. Ako su e_1, ..., e_m simbolički izrazi onda je i `(`e_1`,...,`e_m`)` simbolički izraz.

Primjerice, `(plus,x,1)` je simbolički izraz jer su `plus`, `x` i `1` simbolički izrazi. Slično, `(sin,y)` i `(times,x,(plus,x,1),(sin,y))` su simbolički izrazi.

7.5. LISTE SVOJSTAVA

Simboli su u memoriji računala predstavljeni strukturama podataka koje se nazivaju "*liste svojstava*" (engl. *property list*.) Liste svojstava, generalizacija "*tablice simbola*" (engl. *symbol table*) u asembleru IBM 704, sadrže osnovne podatke o simbolu: ime koje se koristi pri ispisu, adresu same liste svojstava u memoriji, informaciju je li simbol broj, varijabla ili konstanta i slično. Informacije iz liste simbola može koristiti kompajler, ali i programer.

Liste svojstava se mogu mijenjati *deklarativnim naredbama* neočekivanog oblika

```
I declare (...)
```

gdje točke (...) predstavljaju izraze oblika `(`a`;` p_1`, ...,` p_n`)` koji dodaju izraze p_1, ..., p_n u listu svojstava a, ili oblika `(`a_1`, ...,` a_n`,` p`)` koji dodaju izraz p u liste svojstava simbola a_1, ..., a_n.

Sintaksa bliska prirodnom jeziku se nije više javljala za vrijeme razvoja jezika pod McCarthyjevim vodstvom, a McCarthyju se nije svidjelo ni kad se pojavila u kasnijim verzijama Lispa.[85]

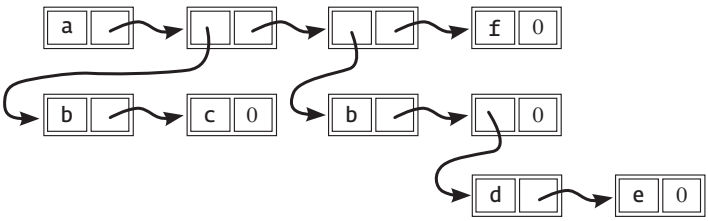
7.6. LISTE I REPREZENTACIJA SIMBOLIČKIH IZRAZA

Simbolički izrazi su u memoriji računala predstavljeni strukturama podataka koje se nazivaju *liste* (engl. *list*.)

85 McCarthy, *Guy Steele interviews John McCarthy, father of Lisp,* 2009.

Štoviše, oblik simboličkih izraza je, čini se, inspiriran upravo jednostavnošću prevođenja u liste.[86,87]

Implementacija lista identična je onoj u FLPL, osim što su imena funkcija za procesiranje lista nešto drugačija. Jedina McCarthyjeva inovacija glede lista u "imperativnom Lispu" je pregledan grafički prikaz lista koji se koristi do danas.



SLIKA 6. Grafički prikaz strukture liste koja predstavlja simbolički izraz `(a,(b,c),(b,(d,e)),f)`[88]

Pravokutnici predstavljaju "elemente liste" (engl. *list elements*), pojedinačne riječi u memoriji računala. Lijeve i desne polovine pravokutnika predstavljaju redom adresni i dekrementni dio riječi. Redoslijed adresnog i dekrementnog dijela riječi u grafičkim prikazima je obrnut u odnosu na redoslijed u riječi računala, zato što su to članovi projekta smatrali prirodnim[89] ili zato što je u asembleru IBM 704 to bio uobičajen redoslijed.[90] Strelica zamjenjuje adresu riječi. Simbol upisan u "adresnom" ili "dekrementnom dijelu" pravokutnika označava da se u odgovarajućem dijelu riječi nalazi *adresa liste svojstava* tog simbola. Prvi element u listi svojstava, u adresnom dijelu, ima vrijednost nula – na taj način programi mogu prepoznati listu svojstava od ostalih lista.

86 McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 18.

87 Stoyan, *Lisp history*, 1979., str. 45.

88 Prema ilustraciji u McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 5.

89 Faase, *The origin of CAR and CDR in LISP*, 2006.

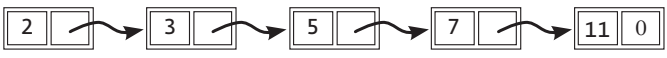
90 Abrahams, *Transcript of discussant's remarks*, u McCarthy, *History of Lisp*, 1981., str. 192.

Isti simbolički izraz se može predstaviti različitim strukturama liste. Dovoljno je da se elementi liste nalaze na različitim mjestima u memoriji.

Iako u grafičkim prikazima lista nema ostalih dijelova riječi osim adresnog i dekrementnog, u "imperativnom Lispu" ti se dijelovi još uvijek koriste. Ako je u adresnom dijelu elementa liste adresa podliste, informacija u prefiksnom dijelu (preciznije, u bitovima 1 i 2, koje McCarthy naziva *indikator*) elementa liste određivala je može li se prilikom brisanja liste obrisati i podlista. U kasnijem razvoju Lispa se odustalo od takvog upravljanja memorijom. Liste svojstava simbola ne pripadaju niti jednoj listi, i ne brišu se ako se lista briše.

Kao i u FLPL-u, liste nisu implementirane kao poseban tip podataka nego su apstrakcija, način razmišljanja programera. Programi procesiraju adrese elemenata liste, a zamišljene liste su u programima predstavljene adresom prvog elementa liste. Takvo svođenje lista na memorijske adrese je kasnije kritizirano kao "previše blisko hardveru".[91] McCarthy se nadao da bi se operacije mogle definirati na nivou cijelih lista, ali mu je iskustvo govorilo da većina operacija "još uvijek" mora biti definirano na nivou elemenata lista.[92]

Listama se mogu "modelirati" i nizovi (engl. *sequences*), matematički objekti koji nemaju veze s nizovima znakova. Primjerice, 2, 3, 5, 7, 11 je konačni niz.



SLIKA 7. Grafički prikaz liste koja modelira niz 2, 3, 5, 7, 11 i čija je eksterna reprezentacija `(2,3,5,7,11)`.

Jednom modelirani kao liste, nizovi imaju i svoju "eksternu reprezentaciju" u obliku simboličkih izraza. Primjerice, lista koja modelira konačni niz 2, 3, 5, 7, 11 ima eksternu reprezentaciju `(2,3,5,7,11)`. Konačni nizovi se često zapisuju u sličnom obliku, kao uređene n-torke, i u matematičkim tekstovima.

91 Landin, *Next 700 programming languages*, 1966., str. 160.

92 McCarthy, *An algebraic language* ..., AIM-001, 1958., str. 5.

McCarthy nije direktno izrazio namjeru da se i cijeli programi predstavljaju u obliku lista. Stoyanova teza je da je, uzme li se u obzir ideja da bi kompajler bio napisan u Lispu samom iznesena u uvodu memoa, McCarthy takve ideje već imao.[93] Ta se teza čini opravdanom, pogotovo uzme li se u obzir i spomenuto *Pismo Perlisu i Turanskom*.

7.7. FUNKCIJE ZA ANALIZU I SINTEZU RIJEČI I REFERENCIRANJE

Definiran je niz funkcija za izdvajanje dijelova riječi. Primjerice, pozivi funkcija `add(`*e*`)`, `dec(`*e*`)` i `ind(`*e*`)` vraćaju vrijednost adresnog i dekrementnog dijela te indikatora riječi koja je rezultat izračunavanja izraza *e*, redom. Neke funkcije su izdvajale vrijednosti proizvoljnih bitova ili segmenata riječi.

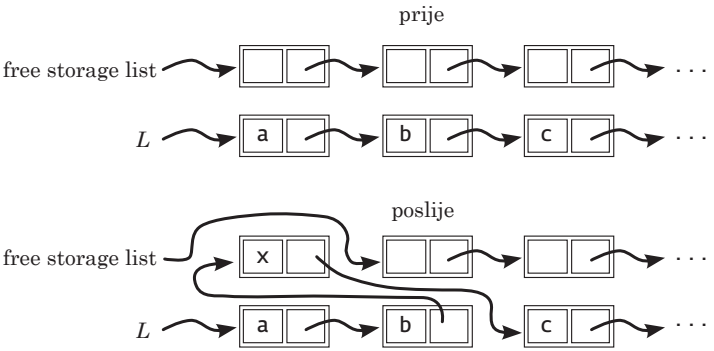
Nekoliko funkcija je sintetiziralo vrijednost riječi. Primjerice, `comb 4(`*p*`,`*d*`,`*t*`,`*a*`)` vraća *vrijednost* riječi dobivene upisivanjem vrijednosti *p*, *d*, *t* i *a* u odgovarajuće dijelove riječi (prefix, decrement, tag i address.)

Funkcije za referenciranje (engl. *reference functions*) izračunavaju vrijednost riječi ili dijelova riječi na zadanoj adresi. Primjerice, pozivi funkcija `cwr(3)`, `car(3)`, `cdr(3)` i `cir(3)` vraćaju vrijednost riječi na adresi 3 te adresnog i dekrementnog dijela i indikatora riječi, redom. Imena funkcija za referenciranje su skraćenice, primjerice, "content of the address part of register".

Podaci se mogu upisivati direktno na memorijske adrese, korištenjem naredbe za pridruživanje i funkcija za referenciranje. Alternativu čine funkcije `stwr`, `star`, `stdr` … takve da, primjerice, `stwr(3,15)` ima isti rezultat kao `cwr(3) = 15`.

93 Stoyan, *Early LISP history (1956-1959)*, 1984., str. 305.

Upravljanje slobodnom memorijom je preuzeto iz FLPL-a. *Lista slobodnog prostora* (engl. *free storage list*) ima istu strukturu kao i druge liste i sadrži memoriju koja trenutno nije iskorištena u programu. Alocira se na početku izvršavanja programa. Funkcije koje potrebuju memoriju za stvaranje nove liste ili dodavanje novih elemenata u već postojeće liste koriste i uklanjaju riječi iz liste slobodnog prostora.



SLIKA 8. Ubacivanje simbola x na treće mjesto u u listi L.[94]

Funkcije za konstruiranje (engl. *construction functions*) odabiru prvu riječ iz liste slobodnog prostora, uklanjaju je iz liste slobodnog prostora, upisuju neku vrijednost u nju i kao rezultat vraćaju *adresu* te riječi. Poziv funkcije `consw(`e`)` upisuje vrijednost e u riječ. Poziv funkcije `consel(`a`,`d`)` upisuje vrijednost a u adresni dio i vrijednost d u dekrementni dio riječi. Čini se da imena funkcija `consw` i `consel` dolaze od fraza "construct word" i "construct element".[95] Postoje i druge slične funkcije.

Funkcija `list` konstruira novu listu od argumenata funkcije. Vrijednost `list(`i_1`,...,`i_n`)` je lista koja sadrži vrijednosti i_1, ..., i_n.

94 Prema McCarthy, *Revisions of the language*, AIM-003, 1958., str. 7.

95 Slagle, *A heuristic program* ..., 1961., str. 17.

"Imperativni Lisp" još nije imao "garbage collector". Funkcija `erase` vraća memorijske adrese koje više nisu potrebne u izračunavanju u listu slobodne memorije. Primjerice, poziv funkcije `erase(3)` vraća treću riječ u memoriji u *listu slobodne memorije*. Vrijednost funkcije `erase` je stari sadržaj upravo oslobođene riječi.

7.9. OSNOVNE OPERACIJE NAD CIJELIM LISTAMA

Način na koji su liste definirane omogućuje definiranje operacija nad cijelim listama. Poziv potprograma `eralis(`*J*`)` briše cijelu *strukturu liste* čiji je prvi element na adresi *J* i oslobođenu memoriju vraća u *listu slobodnog prostora*. McCarthy definira potprogram `eralis` u "imperativnom Lispu"; to je prvi primjer programa u memou pa ga se može smatrati prvim zabilježenim programom u Lispu.

```
subroutine eralis(J)
/    J = 0 → return
     go (a(cir(J))
a(1) jnk = erase(car(J))
a(0) eralis(dec(erase(J)))
     return
a(2) eralis(car(J))
\    go (a(o))
```

Potprogram `eralis` analizira prvi element liste. Ako je lista prazna, potprogram završava rad. Ako nije, izvršavanje se grana ovisno o vrijednosti indikatora prvog elementa liste.

1. Ako indikator ima vrijednost 0, tada je vrijednost u adresnom dijelu elementa liste adresa "posuđene" podliste koju ne treba obrisati iz memorije. Potprogram `eralis` briše element liste pozivanjem funkcije `erase` i primjenjuje sam sebe na ostatak liste.

2. Ako indikator ima vrijednost 1, u adresnom dijelu elementa liste nalazi se adresa podatka; `eralis` briše taj podatak pozivanjem funkcije `erase`, briše prvi element liste pozivanjem funkcije `erase` i primjenjuje sam sebe na ostatak liste.

3. Konačno, ako indikator ima vrijednost 2, tada je podatak lista koja nije "posuđena;" `eralis` primjenjuje sam sebe na taj podatak, zatim briše prvi element liste pozivajući funkciju `erase` i primjenjuje sam sebe na ostatak liste.

Kuriozitet je prva sintaktička greška u Lisp programu: u zadnjem retku umjesto nule McCarthy je napisao malo slovo `o`.

Funkcija `copy` konstruira i vraća kao vrijednost izračunavanja kopiju *strukture liste* na zadanoj adresi. Memorija u koju se sprema kopija uzima se iz liste slobodnog prostora. Funkcija je definirana uvjetnim izrazom u kojem se osnovni slučaj rješava direktno, a ostali slučajevi rekurzivnim pozivom iste funkcije, primijenjene na ostatak liste. Taj uzorak je kasnije primijenjen na mnoge druge funkcije. Funkcija `search` traži elemente strukture liste koji zadovoljavaju zadani kriterij. Funkcija `equal` provjerava jednakost struktura liste na zadanim adresama.

7.10. FUNKCIJA MAPLIST

Funkcija `maplist` koja primjenjuje zadanu funkciju *na podatke u listi* ima važnu ulogu u daljnjem razvoju Lispa. U prvom memou `maplist` je šturo i neprecizno definirana[96], ali se dvoznačnosti mogu izbjeći temeljem primjera[97] i kasnijeg McCarthyjevog pojašnjenja.[98] Primjerice, vrijednost simboličkog izraza

96 "Maplist (L,J,f(J)). The value or this function is the address of a list formed from the list L by mapping the element J into f(J)." McCarthy, *An algebraic language* …, AIM-001, 1958., str. 17.

97 Funkcija za diferenciranje:

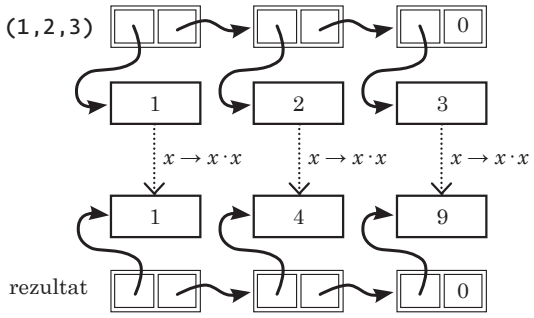
```
function diff(L)
/ diff=(…
        car(L)="x" → 1,
        car(L)="plus" → consel("plus",
                               maplist(cdr(L),
                                       J,
                                       diff(J))),
\       …)
```

McCarthy, *An algebraic language* …, AIM-001, 1958., str. 20.

98 "The version of maplist in memo 1 was written "maplist(L,J,f(J))" where J is a dummy variable which ranges over the address parts of the words in the list L and f(J) was an expression in J." McCarthy, *Symbol manipulating language*, AIM-004, 1958., str. 4.

```
maplist(list(1,2,3),x,x*x)
```

se izračunava tako da varijabla `x` poprima vrijednost svih podataka čija se adresa nalazi u adresnom dijelu *elemenata liste* `list(1,2,3)`. Za sve te vrijednosti se izračuna `x*x`, tj. 1, 4 i 9. Formira se nova lista, od elemenata liste uzetih s *liste slobodnog prostora* u čiji se adresni dio upišu adrese u kojima se nalaze rezultati prethodnog izračunavanja. Ta se lista vraća kao vrijednost funkcije.



SLIKA 9. Rezultat primjene `maplist(list(1,2,3),x,x*x)`.

Općenito, izraz `maplist(`L`,`J`,`$f(J)$`)`, gdje je L izraz koji se izračunava u listu, J simbol, $f(J)$ izraz dobiven uvrštenjem simbola J u funkciju f, izračunava se tako da (1) varijabla J poprima sve vrijednosti u L; (2) za svaku od vrijednosti J se računa vrijednost $f(J)$ i (3) formira se nova lista pri čemu se u adresnim dijelovima elemenata liste nalaze vrijednosti $f(J)$. Ta se lista vraća kao rezultat izračunavanja.

8.

Elementi funkcionalnog programiranja

U periodu od rujna 1958. do ožujka 1959. Lisp se razvijao postepeno i kontinuirano, bez većeg redizajna. Promjene u tom periodu su, uglavnom, vodile ka jeziku koji će se kasnije nazivati "čisti Lisp" i koji je opisan u prvom McCarthyjevom članku o Lispu.[99]

8.1. NOVA FUNKCIJA MAPLIST

U drugom memou McCarthy kritizira prvu verziju funkcije `maplist`[100,101] koja se pozivala izrazom `maplist(`L`,`J`,`$f(J)$`)`. Prvo, bilo bi bolje da u pozivu funkcije vrijednosti J budu *elementi liste* L umjesto adresnih dijelova elemenata liste.[102] Ako je poznat element liste, njegov adresni dio se može izračunati. Obrat ne vrijedi.

Drugi, važniji prigovor je da se ime varijable ne može "uvjerljivo" koristiti kao argument funkcije. Primjerice, pozove li se prva verzija funkcije `maplist`, izrazom

```
maplist(list(1,2,3),x,x*x)          (*)
```

podizrazi `x` i `x*x` bit će izračunati a rezultati izračunavanja proslijeđeni funkciji `maplist`. Pri tome, prilikom izračunavanja `x*x` doći će do greške, jer varijabla `x` nema definiranu vrijednost, a ako je negdje u programu ta vrijednost i definirana, rezultat neće sadržavati informaciju potrebnu za izračunavanje izraza (*).

Stoga McCarthy definira novu verziju funkcije `maplist`. Funkcija se poziva izrazom oblika

`maplist(`L`,`f`)`,

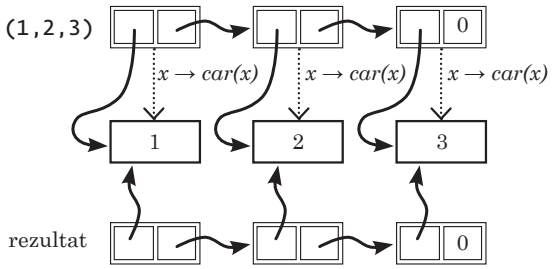
99 "The development of the LISP system went through several stages of simplification in the course ot its development and was eventually seen to be based on a scheme for representing the partial recursive functions of a certain class of symbolic expressions." McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 1.

100 McCarthy, *A revised version of MAPLIST*, AIM-002, 1958. U vrijeme pisanja knjige bila je javno dostupna samo prva stranica AIM-002. Ta je stranica identična početku drugog poglavlja AIM-004 pa je, možda, cijeli AIM-002 sadržan u AIM-004.

101 McCarthy, *Revisions of the language*, AIM-004, 1958.

102 McCarthy, *Revisions of the language*, AIM-004, 1958., str. 4.

gdje je *L* izraz čija je vrijednost lista, a *f* izraz čija je vrijednost funkcija. Primjerice, vrijednost poziva funkcije `maplist(list(1,2,3),car)` je lista sastavljena od vrijednosti dobivenih primjenom funkcije `car` na sve *elemente liste* dobivene izračunavanjem `list(1,2,3)`.



SLIKA 10. Rezultat izračunavanja `maplist(list(1,2,3),car)`.

Definicija funkcije `maplist` razrješava moguće dvojbe.

```
maplist(L,f) = (L = 0 → 0,
                1 → cons(f(L),
                         maplist(cdr(L),f))).
```
[103]

Funkcija `cons`[104] uzima riječ iz liste slobodnog prostora, u adresni dio riječi upisuje adresu `f(L)` a u dekrementni dio adresu `maplist(cdr(L),f)`.

8.2. MCCARTHYJEVI LAMBDA-IZRAZI

Druga verzija funkcije `maplist`, ovakva kakva je, ograničenija je od prethodne verzije. Primjerice, s drugom verzijom nije moguće napisati ekvivalent izraza

```
maplist(list(1,2,3),x,x*x)
```

103 Ova i mnoge druge funkcije u ovoj knjizi formatirane su prelamanjem i uvlačenjem redova, kako je uobičajeno danas, ali ne u vrijeme u kojem su originalni dokumenti pisani. Formatiranje znatno pridonosi čitljivosti.

104 Ta se funkcija u AIM-001 zvala `consel`.

s prvom verzijom funkcije `maplist`. Zato se McCarthy vraća ideji iz *Pisma Perlisu i Turanskom*. Definira "funkcionalnu apstrakciju" kao izraz koji daje pravila za izračunavanje primjene funkcija na argumente. Izraz poput `x*x+y` nema to svojstvo. Pokuša li se, primjerice, izračunati

```
(x*x+y)(3,4)
```

neće biti jasno koje vrijednosti treba pridružiti parametrima `x` i `y`. McCarthyjevi *lambda-izrazi*[105], primjerice `λ(x,y,x*x+y)`, inspirirani Churchovima[106] zadovoljavaju taj kriterij. Vrijednost izraza

```
λ(x,y,x*x+y)(3,4)
```

je jednaka vrijednosti `3*3+4`. Općenitije, vrijednost `λ(`*J*`,`*E*`)(`*e*`)` je vrijednost izraza dobivenog supstitucijom argumenta *e* umjesto *J* u *E*. Slično vrijedi i za lambda-izraze s više varijabli. Primjerice, poziv druge verzije funkcije

```
maplist(list(1,2,3),λ(J,car(J)))
```

ekvivalentan je pozivu prve verzije funkcije `maplist(list(1,2,3),J,J)`.

Važan McCarthyjev primjer je definicija funkcije za deriviranje `diff`[107] koja spretno koristi drugu verziju `maplist` i lambda-izraze.

105 McCarthy nije koristio termin "McCarthyjevi lambda-izrazi", nego "funkcije" koji danas više nije dovoljno precizan. Pogodniji termin "lambda-izrazi" počeo se koristiti kasnije, a pridjev "McCarthyjevi" potreban je radi razlikovanja od drugih vrsta lambda-izraza, posebno Churchovih i Rochesterovih.

106 Church, *The calculi of λ-conversion*, 1941.

107 McCarthy, *Revisions of the language*, AIM-004, 1958., str. 7.

```
diff(L,V)=
  (L=V → C1,
   car(L)=O → C0,
   car(L)=plus → cons(plus,
                      maplist(cdr(L),
                              λ(J,
                                diff(car(J),
                                     V)))),
   car(L)=times →
     cons(plus,
          maplist(cdr(L),
             λ(J,
               cons(times,
                    maplist(cdr(L),
                            λ(K,
                             (J≠K →
                                copy(car(K)),
                              1→diff(car(K),
                                     V)))))))),
   1 → error)
```

`C0` i `C1` predstavljaju konstante 0 i 1. Valja uočiti da je posljednji uvjet u prethodnoj definiciji `1`, tj. izračunava se u "svim preostalim slučajevima". Program je neobično vješto i lijepo napisan, to više jer McCarthy u to vrijeme jedva da je mogao imati iskustava s pisanjem programa u Lispu.

S povijesne distance, nije očigledno da su lambda-izrazi najsretnije rješenje problema s funkcijom `maplist`. Lambda-izrazi su dosljedan razvoj ideje da funkcije mogu biti argumenti drugih funkcija. Ipak, potreba da se neizračunati simboli i izrazi koriste kao argumenti funkcija je ostala i kasnije je riješena općenitije, uvođenjem "specijalnog operatora" `QUOTE`. Zbog toga, moglo bi se reći da su lambda-izrazi uvedeni u Lisp preuranjeno, ili čak slučajno. Ipak, gotovo svi dijalekti Lispa su zadržali lambda-izraze.[108]

108 Značajna iznimka je PicoLisp Nijemca Alexandra Burgera. Burger, *The PicoLisp reference*.

Postoje stavovi da su razlike između lambda računa i Lispa velike[109, 110] kao i da je λ-račun osnova Lispa[111, 112, 113] Sam McCarthy je ove druge odbacio kao "mit".[114]

8.3. POJEDNOSTAVLJENJE JEZIKA

McCarthy je ubrzo primijetio[115] da je za sve potrebne operacije dovoljno koristiti samo adresni i dekrementni dio riječi. Od velikog broja funkcija za procesiranje pojedinačnih riječi ostalo je svega nekoliko. Neke funkcije su dobile nova imena.

Funkcije[116] `add(`w`)` i `dec(`w`)` izdvajaju adresni i dekrementni dio riječi koja je vrijednost izraza w.

Funkcija `comb(`a`,`d`)` izračunava vrijednost riječi koja sadrži vrijednosti a i d u adresnom i dekrementnom dijelu.

Funkcija `cwr(`n`)`, `car(`n`)` i `cdr(`n`)` vraćaju vrijednost riječi, te adresnog i dekrementnog dijela riječi na adresi koja je vrijednost n.

109 Cianalgini & Hindley, *Lambda calculus*, in Wiley Enc. of Comp. Sci., 2008., str. 5.

110 Barendregt & Barensen, *Introduction to λ-calculus*, 2000., str. 30.

111 Weizenbaum, *Review of "The LISP 2 programming language …"*, 1967., str. 236.

112 Moore & Mertens, *The nature of computation*, 2011., str. 295.

113 Mac Lane, *Group extensions for 45 years*, 1988., str. 29.

114 "That was fine for that recursive definition of applying a function to everything on the list. No new ideas were required. But then, how do you write these functions? And so, the way in which to do that was to borrow from Church's Lambda Calculus, to borrow the lambda notation. Now, having borrowed this notation, one of the myths concerning LISP that people think up or invent for themselves becomes apparent, and that is that LISP is somehow a realization of the lambda calculus, or that was the intention. The truth is that I didn't understand the lambda calculus, really. In particular, I didn't understand that you really could do conditional expressions in recursion in some sense in the pure lambda calculus. So, it wasn't an attempt to make lambda calculus practical, although if someone had started out with that intention, he might have ended up with something like LISP."
McCarthy, *History of LISP*, 1981., str. 190.

115 McCarthy, *Revisions of the language*, AIM-003, 1958., str. 1.

116 Preciznije bi bilo pisati "poziv funkcije", ali zbog jednostavnosti, uobičajeno je samo "funkcija". Isto vrijedi u ostatku teksta.

Funkcija `consw(`w`)` upisuje vrijednost w u riječ uzetu iz liste slobodnog prostora i vraća adresu te riječi.

Funkcija `cons(`a`,`d`)` upisuje vrijednost a i d u adresni i dekrementni dio riječi uzete iz liste slobodnog prostora i vraća adresu te riječi.

Funkcija `erase(`L`)` vraća riječ na adresi L u listu slobodnog prostora, te kao rezultat vraća prethodnu vrijednost riječi L.

Podržano je i nekoliko operacija nad cijelim listama. Poziv funkcije `copy(`L`)` kopira cijelu strukturu liste na adresi L u novu listu, uzimajući potrebne riječi iz liste slobodnog prostora.

```
copy(L) = (L=0 → 0,
           car(L)=0 → L,
           1 → cons(copy(car(L)),
                    copy(cdr(L))))
```

Funkcija `equal(`L_1`,`L_2`)` uspoređuje strukture liste na adresama L_1 i L_2 i vraća 1 ako su jednake, 0 inače.

```
equal(L1,L2) = (L1=L2 → 1,
                car(L1)=0 ∨ car(L2)=0 → 0,
                1 → equal(car(L1), car(L2)) ∧
                    equal(cdr(L1), cdr(L2)) )
```

Uvjet u drugom retku je potreban jer se u adresnom dijelu prvog elementa u listama svojstava nalazi broj 0. Ako su `L1` i `L2` na različitim adresama (što je osigurano uvjetom `L1=L2`) i makar jedan od `L1` i `L2` je lista svojstava, onda `equal(L1,L2)` ima vrijednost 0.

Potprogram `eralis(`L`)` briše cijelu *strukturu liste* na adresi L.

```
      subroutine (eralis(L))
    / L = 0 ∨ car(L) = 0 → return
      M = erase(L)
      eralis(add(M))
      eralis(dec(M))
    \ return
```

Potprogram je znatno jednostavniji od onoga u prethodnom poglavlju, ali više ne može prepoznati jesu li podaci u listi koju se briše "posuđeni".

Definiran je i potprogram `print` i funkcija `read` koja kao rezultat vraća adresu liste konstruirane na temelju zapisa simboličkog izraza s bušene kartice ili drugog medija.

Funkcije koje modificiraju vrijednost elemenata liste, `star` i `stdr` izbačene su iz jezika, pokazat će se, privremeno.

8.4. IMPLEMENTACIJA REKURZIVNIH FUNKCIJA

Funkcije `maplist`, `diff` i neke druge su rekurzivne; pozivaju same sebe. Implementacija rekurzivnih funkcija predstavljala je određenu poteškoću jer različite instance istih funkcija koriste iste varijable.

Problem je riješen spremanjem "zaštićene privremene memorije" (engl. *protected temporary storage*), koja sadrži simbole i vrijednosti koje trebaju biti sačuvane prilikom poziva iste funkcije na "javni stog", (engl. *stack*, tada zvan *public push down list*).[117] Pozvana funkcija prvo spremi sadržaj zaštićene privremene memorije na stog, izračuna vrijednost koju treba vratiti, a zatim sa stoga rekonstruira "zaštićenu privremenu memoriju". Rješenje je originalno, ali nije složeno i do njega je došao već Turing.[118]

8.5. FUNKCIONALNI I IMPERATIVNI STIL

Danas je raširen stav da su programi pisani u funkcionalnom stilu "elegantniji", ali manje efikasni od programa pisanih u imperativnom stilu. Iako termini poput "funkcionalnog" i "imperativnog stila" još nisu bili korišteni, te su razlike vrlo brzo uočene. Osim prethodne, "funkcionalne definicije" funkcije `maplist`

117 Russell, *Explanation of big "P"*, AIM-009, 1959., str. 2.

118 Turing, *Proposed electronic calculator*, 1945., str. 12.

```
maplist(L,f) = (L = 0 → 0,
                1 → cons(f(L),
                         maplist(cdr(L),f))).
```

McCarthy je u četvrtom memou definirao i drugu, imperativnu verziju.

```
maplist(L,f) = / L = 0 → return(0)
                 maplist = cons(f(L),0)
                 M = maplist
               a1 L = cdr (L)
                 cdr(M) = cons(f(L),0)
                 cdr(L) = 0 → return(maplist)
                 M = cdr(M)
                \ go(a1)
```

Imperativna verzija je bila oko četiri puta brža. McCarthy, ipak, nije bio voljan preporučiti pisanje programa u manje jasnom, imperativnom stilu.[119] Spekulirao je da bi kompajler mogao prevoditi programe iz jednog u drugi stil, ali nije vidio kako napisati takav kompajler. Umjesto toga, pragmatično je zaključio da mali broj često korištenih programa kod kojih je brzina bitna treba pisati u imperativnom, a sve ostale u funkcionalnom stilu. Uočljiva je i razlika u načinu uporabe naredbe `return`.

8.6. NEDEFINABILNOST FUNKCIJE LIST

McCarthy proširuje diskusiju o već ukratko opisanoj funkciji `list`. Funkcija može biti definirana jednakostima

$$\texttt{list}(i) = \texttt{cons}(i,0),$$

$$\texttt{list}(i_1,\ldots,i_n) = \texttt{cons}(i_1, \texttt{list}(i_2, \ldots,i_n)).$$

119 "One is very reluctant to say that routines like `maplist` should be described by programs like the above which is certainly much less clear than the previous description."
McCarthy, *Revisions of the language*, AIM-004, 1958., str. 8.

McCarthy uočava da je `list` definirana rekurzivno po broju argumenata, te da se takav `list` ne može definirati u Lispu, nego se mora implementirati u kompajleru za Lisp.[120]

8.7. SUPSTITUCIJSKE FUNKCIJE I FUNKCIJA APPLY

McCarthy uvodi pomalo zagonetan pojam supstitucijskih funkcija koje se "primjenjuju" (engl. *apply*) na listu argumenata.[121] Supstitucijske funkcije su simbolički izrazi, primjerice,

```
(subfun,(x,y),(times,x,y)).
```

"Primjeni" li se ta supstitucijska funkcija na listu argumenata

```
((plus,a,b),(minus,a,b))
```

rezultat je

```
(times,(plus,a,b),(minus,a,b)).
```

Općenito, rezultat "primjene" supstitucijske funkcije

$$(\texttt{subfun},(x_1,\ldots,x_n),e)$$

gdje su x_1, …, x_n simboli, e bilo koji izraz, na listu argumenata

$$(e_1,\ldots,e_n)$$

je vrijednost izraza e' koji je dobiven simultanom supstitucijom svih slobodnih pojavljivanja x_1, …, x_n u e argumentima e_1, …, e_n redom.

McCarthy nije napisao što točno znači da se supstitucijske funkcije "primjenjuju" na listu argumenata. Danas je uobičajeno reći da, primjerice, izraz `car(`e`)` označava primjenu funkcije `car` na izraz e. No, u pedesetak stranica memoa koje je do tada napisao, McCarthy je samo jednom

120 McCarthy, *Revisions of the language*, AIM-004, 1958., str. 16.

121 McCarthy, *Revisions of the language*, AIM-004, 1958., str. 19.

koristio riječ "apply" u tom značenju, i to prilično neformalno.[122] Gotovo sigurno je da je pod "primjenom" supstitucijskih funkcija mislio na izračunavanje izraza oblika

`apply(`*l*`,`*f*`)`

gdje je *l* lista argumenata, *f* supstitucijska funkcija a `apply` Lisp funkcija. Primjerice, ako

f ima vrijednost `(subfun,(x,y),(times,x,y))` a

l ima vrijednost `((plus,a,b),(minus,a,b))`

onda

`apply(`*l*`,`*f*`)` ima vrijednost `(times,(plus,a,b),(minus,a,b))`.

8.8. POMOĆNE FUNKCIJE I IMPLEMENTACIJA APPLY

Iako rani oblik funkcije `apply` nije pretjereno složen, za implementaciju `apply` potrebne su pomoćne, manje važne, ali zanimljive funkcije `search`, `subst`, `sublis` i `error`.

Funkcija `search` traži podatak koji zadovoljava zadani kriterij u listi; nađe li ga, vraća zadanu funkciju od tog podatka; ako ga ne nađe, vraća zadanu vrijednost. Primjerice, poziv funkcije

```
search(list(1,2,3), λ(x,x+x=2), λ(x,x*x*x), 0).
```

ima vrijednost 8. Argumenti poziva funkcije su lista u kojoj se traži podatak, predikat koji podatak treba zadovoljiti, funkcija koja se primjenjuje na pronađeni podatak koji zadovoljava predikat i vrijednost koja se vraća ako ne postoji podatak koji zadovoljava predikat.

```
search(L,p,f,u)=(L=0 → u,
                 p(L) → f(L),
                 1 → search(cdr(L),p,f,u))
```

122 McCarthy, *Revisions of the language*, AIM-003, 1958., str. 6.

Funkcija `subst` provodi supstituciju. Primjerice, neka je l izraz koji ima vrijednost `(A,B)`, v izraz koji ima vrijednost `(X,Y)`, m izraz koji ima vrijednost `((X,Y),C,(X,Y))`. Tada `subst(`l`,`v`,`m`)` ima vrijednost `((A,B),C,(A,B))`. Općenitije, ako su l, v i m izrazi, izraz `subst(`l`,`v`,`m`)` ima za vrijednost rezultat supstitucije vrijednosti l umjesto vrijednosti v u vrijednost m.

```
subst(L,V,M)= (M=0 → 0,
               equal(M,V) → copy(L),
               car(M)=0 → M,
               1 → cons(subst(L,V,car(M)),
                        subst(L,V,cdr(M))))
```

Uvjet `car(M)=0` znači da je vrijednost `M` simbol (liste svojstava imaju vrijednost 0 u adresnom dijelu prvog elementa liste.)

Funkcija `sublis` provodi više "supstitucija, kodiranih u listi oblika `((`l_1`,`v_1`),...,(`l_n`,`v_n`))`, pri čemu je l_i simbolički izraz koji se supstituira a v_i simbolički izraz umjesto kojeg se supstituira. Primjerice, ako je

p izraz koji ima vrijednost `((X,A),((Y,Z),(B,C)))`,

e izraz koji ima vrijednost `((X,Y),X,Y,(Y,Z))`

onda poziv funkcije

`sublis(`p`,`e`)` ima vrijednost `((A,Y),A,Y,(B,C))`.

Definicija funkcije `sublis` je vrlo sofisticirana.

```
sublis(P,E) =
  maplist(E,
          λ(J,search(P,
                 λ(K,equal(car(J),
                           car(car(K)))),
                 λ(K,copy(car(cdr(car(K))))),
                 (car(car(J))=0 → car(J),
                  1 → subst(P,car(J))))))
```

Definicija ima dvije manje greške. U posljednjem retku umjesto `subst` treba pisati `sublis`. Drugo, program neće raditi ako je izraz u kojem treba izvršiti supstituciju, `E`, trivijalan – simbol. Primjerice, ako je p izraz koji ima vrijednost `((X,A),(A,B))`, e izraz koji ima vrijednost `(X,A,X,A)` onda poziv funkcije `sublis(`p`,`e`)` ima vrijednost `(A,B,A,B)`.

Funkcija `pair` kreira listu parova, potrebnu za upotrebu u funkciji `sublis`. Ako su l i v izrazi čija je vrijednost `(`l_1`,...,`l_n`)`, `(`v_1`,...,`v_n`)` onda poziv funkcije `pair(`l`,`v`)` ima vrijednost `((`l_1`,`v_1`),...,(`l_n`,`v_n`))`.

```
pair(Ll,L2)=( Ll=0 ∧ L2=0 → 0,
              Ll=0 ≠ L2=0 → error,
              1 → cons(cons(copy(car(Ll)),
                            cons(copy(car(L2)),
                                 0)),
                       pair(cdr(Ll),
                            cdr(L2))))
```

Funkcija `error` se poziva u iznimnim slučajevima i ispisuje poruku o grešci i podatke korisne za analizu greške.

Nakon svih pomoćnih definicija, definicija funkcije `apply` je kratka.

```
apply(L,f) =
  (car(f) = subfun → sublis(pair(car(cdr(f)),
                                 L),
                            car(cdr(cdr(f)))),
   1 → error)
```

Supstitucijske funkcije su značajan korak u razvoju Lispa. Zbog njih je definirana vrlo općenita i značajna funkcija `apply` koja je kasnije shvaćena kao "univerzalna funkcija".

Moglo bi se pomisliti da je funkcija `apply` jednostavnija od funkcije `maplist` te

```
apply(L,f) = car(maplist(list(L),f)).
```

Prethodni izraz, ipak, vrijedi samo za funkcije jedne varijable, dok se `apply` može primijeniti na funkcije s proizvoljnim brojem varijabli.

8.9. AUTOMATIZIRANO UPRAVLJANJE MEMORIJOM

Funkcije za brisanje elemenata lista i cijelih lista `erase` i `eralis` opisane u prvom memou, u sljedeća tri memoa nisu korištene. Indikativan je McCarthyjev komentar uz gornju definiciju funkcije `apply`. On uočava da funkcija `pair` stvara novu listu i da ta lista nije pridružena niti jednoj varijabli, pa se onda ne može niti obrisati funkcijom `eralis`. Umjesto da korištenjem naredbe za pridruživanje riješi taj problem, McCarthy primjećuje da, ako se ne napiše kompajler koji bi ubacio instrukcije za brisanje takvih, "pomoćnih lista", one će uvijek "krasti memoriju".[123,124,125]

8.10. ROCHESTEROVI LAMBDA IZRAZI

Rochester je uočio da korištenjem McCarthyjevih lambda-izraza nije moguće definirati rekurzivne funkcije. Kao što će se kasnije pokazati, to je ipak moguće – ali nije jednostavno. Zato uvodi drugi, izražajniji oblik lambda-izraza (on ih naziva "funkcijske apstrakcije") u kojem je specificirano ime funkcije. Primjerice, funkcija koja provjerava je li u listi svojstava simbola upisan i simbol `var` definirana je lambda izrazom

```
λ(F(K),(K=0 → 0,
        car(K) = var → 1,
        1 → F(cdr(K)))).
```
[126]

123 McCarthy, *Revisions of the language*, AIM-004, 1958., str. 20.

124 "When I wrote this program for differentiation … it was just much too awkward to write Erasure, that is erasing things. ... So, I thought very hard "Is there some way in which we can eliminate having to have explicit erasure, in order to be able to write the differentiation function in a way corresponds to the way that mathematicians describe it?" They don't describe, they don't mention erasing anything in the calculus textbook, so I worked hard on that and came up with garbage collection."
McCarthy, *Guy Steele interviews John McCarthy, father of Lisp*, 2009.

125 McCarthy, *The logic and philosophy of AI*, 1988., str. 5.

126 Rochester, AIM-005, 1958., str. 15.

Varijabla `F` nema vrijednost izvan lambda-izraza. Rochesterovi lambda-izrazi će biti neznatno izmjenjeni i preimenovani u *label-izraze* u "čistom Lispu".

8.11. FORMATIRANJE PROGRAMA

Rochester prvi počinje formatirati na danas uobičajen način, uvlačenjem redaka i čak eksplicitno preporučuje takvo zapisivanje. Primjerice, gornji je program formatiran gotovo identično Rochesterovom originalu. Drugi članovi Lisp zajednice još prilično dugo nisu prihvatili tu ideju. Uvlačenje redaka se eksplicitno spominje u Lisp 1.5. priručniku.[127]

8.12. KOMPOZICIJE FUNKCIJA CAR I CDR

Rochesterov prilog je i kraći način zapisivanja kompozicija funkcija `car` i `cdr`. Primjerice, umjesto `car(cdr(cdr(L)))` može se pisati `caddr(L)`.[128] No, ta ideja nije nova; primijenjena je već u FLPL-u.

Iako su imena funkcija poput `caddr` praktična, i prihvaćena su u gotovo svim dijalektima Lispa, autoru ove knjige ideja se ne čini neupitnom. Simboli poput `caddr` nisu tek imena, nego sadrže informaciju koju čovjek, programer, mora dekodirati. Te informacije su onda nedostupne, ili makar teže dostupne procesiranju od strane programa.

8.13. FUNKCIJA COMPUTE

U Memou 7. McCarthy opisuje, ali ne definira, potprogram `compute(L,C)` gdje je `L` izraz koji treba izračunati, a `C` je adresa na kojoj treba biti spremljen rezultat izračunavanja `L`. Vrijednost poziva `compute(L,C)` je program u asembleru, u formi liste.[129] Implementiranje funkcija koje procesiraju izraze jezika u samom jeziku je posebnost Lispa.

127 McCarthy et al., *LISP 1.5 programmer's manual*, 1962., str. 16.

128 Rochester, AIM-005, 1958., str. 14.

129 McCarthy, *Notes on the compiler*, AIM-007, 1958., str. 1.

9.

Čisti Lisp

Razvoj od "imperativnog Lispa" do kraja 1958. je bio vrlo brz i odlično dokumentiran. Tijekom tog perioda, McCarthy je shvatio da kombinacija iznesenih ideja ne čini samo praktičan programski jezik, nego i "elegantan matematički sistem" za opisivanje izračunljivih funkcija, "uredniji" od Turingovih strojeva i teorije rekurzivnih funkcija. Takvo razumijevanje potaklo je daljnja pojednostavljenja, dijelom iz estetskih razloga, a dijelom zbog želje za razvojem tehnika za dokazivanje korektnosti programa.[130, 131] U nekoliko internih memoa i izvješća napisanih tijekom ožujka i travnja 1959. pod istim naslovom, *"Recursive functions of symbolic expressions and their computation by machine"*, pojednostavljeni Lisp je već definiran i precizno opisan.[132, 133, 134] U Memou 8. McCarthy je pokušao koristiti deskriptivnija imena funkcija: *first*, *rest* i *combine* umjesto *car*, *cdr* i *cons*. On i Russell pokušali su uvjeriti ostale članove projekta da koriste nova imena, ali bez uspjeha.[135] Već u sljedećem memou McCarthy odustaje od novih imena. U rujnu 1959. Lisp je predstavljen na konferenciji ACM.[136]

Konačnu formu tekst je dobio u članku od desetak stranica objavljenome početkom 1960. pod istim naslovom.[137] Neke ideje, poglavito odnos Lispa i teorije izračunljivosti[138] ostale su zabilježene samo u internim dokumentima. Najavljeni nastavak članka trebao je sadržavati primjere izračunavanja s algebarskim izrazima, ali nikad nije napisan.[139]

130 McCarthy, *History of LISP*, 1981., str. 173.

131 McCarthy, *History of LISP*, 1981., str. 178.

132 McCarthy, *Recursive functions* ..., AIM-008, 1959.

133 McCarthy, *Recursive functions* ..., AIM-011, 1959.

134 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., p 124-152.

135 Faase, *The origin of CAR and CDR in LISP*, 2006.

136 McCarthy, *LISP: A programming system for symbolic manipulation*, 1990.

137 McCarthy, *Recursive functions ...,* CACM, 1960., str. 193.

138 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959.

139 McCarthy, *History of LISP*, 1981., str. 178.

Iznenađujuće čak i za članove AI projekta, Lisp se pokazao jezikom čija je važna osobina — ljepota.[140] Čini se da su neki čitaoci imali problema s razumijevanjem osnovnih ideja jezika.[141]

U vrijeme kad je razvijen, novi dijalekt nije imao posebno ime. Nazivan je jednostavno "LISP". Tek s pojavom znatno različitog Lispa 1.5, dijalekt se počinje nazivati "čisti" (engl. *pure*)[142] a ponekad i "vanilla Lisp".[143] "Čisti Lisp" se ponekad pogrešno identificira s Lispom I., internom verzijom praktičnog Lispa koja je prethodila poznatijem Lispu 1.5. Interno, članovi grupe su ga nazivali i RSFE Lisp.[144] Zasebna implementacija "čistog Lispa" nije postojala, nego ga se moglo shvatiti kao podjezik implementiranog Lispa.

McCarthy je nekoliko puta pokušao sasvim okvirno odrediti "osnovu Lispa". Opisuje ga kao "programski sistem za korištenje IBM 704 računala za računanje simboličkih informacija u formi S-izraza" a "osnova sistema je program koji izračunava S-funkcije"[145] ili "način generiranja rekurzivnih funkcija nad simboličkim izrazima".[146]

Definicije i teze iz članka i internih dokumenata mogu se podijeliti u desetak grupa.

140 "And then, two years later came John's paper … That changed the whole ball game, and it changed how people perceived LISP. Now all of a sudden, LISP was not merely a language you used to do things. It was now something you looked at: an object of beauty. It was something to be studied as an object in and of itself."
Abrahams, *Transcript of discutant's remarks* in McCarthy, *History of LISP*, 1981., str. 193.

141 Woodward i Jenkins, *Atoms and lists*, 1961., str. 51.

142 McCarthy et al, *LISP 1.5 programmer's manual*, 1962., str. 20.

143 Talcott, *Rum. An intensional theory of function …*, 1988., str. 15.

144 Osobna komunikacija s Abrahamsom., 2014.

145 "The Lisp programming system is the system for using the lBM 704 computer to compute with symbolic information in the form of S-expressions (…) The basis of the system is a way of writing computer programs to evaluate S-functions."
McCarthy, *Recursive functions …*, CACM, 1960., str. 191.

146 "The LISP programming system will be shown … to be based mathematically on a way of generating the general recursive functions of symbolic expressions."
McCarthy et al., *The LISP programming system*, 1959., str. 122.

1. Uvodi se niz pravila za formiranje matematičkih izraza, posebno izraza kojima se definiraju funkcije.

2. Definiraju se *simbolički izrazi* ili *S-izrazi* kao posebna vrsta nizova znakova. Primjerice, `A`, `B`, `DA`, `E1`, `(A,B)`, `(A,(B,C),(B,(DA,E1)))` su simbolički izrazi. Simbolički izrazi su jedina vrsta podataka u "čistom Lispu".

3. Definiraju se *funkcije na simboličkim izrazima* ili *S-funkcije*. Nekoliko je osnovnih S-funkcija. Ostale S-funkcije su definirane uz pomoć malog broja sredstava popisanih u 1. Primjerice, *append* je funkcija koja paru `(A,B)` i `((A))` pridružuje `(A,B,(A))`.
   Svaka S-funkcija je definirana nekim matematičkim izrazom. Radi izbjegavanja miješanja definicija S-funkcija sa simboličkim izrazima, okrugle zagrade su zamijenjene uglatima, a zarezi točka-zarezima. Tako pisani izrazi nazivaju se meta-izrazi. Primjerice, $\lambda[[x]; append[x; x]]$ je meta-izraz kojim se definira S-funkcija koja simboličkom izrazu x pridružuje $append[x; x]$.
   Iz definicije svake S-funkcije proizlazi i način kako se za zadane argumente može izračunati vrijednost funkcije.
4. Definira se prevođenje meta-izraza u S-izraze. Koristi se "Cambridge poljska notacija". Primjerice, meta-izraz $append[x; x]$ se prevodi u `(APPEND,X,X)`. To se piše i

$$append[x; x]^* = \texttt{(APPEND,X,X)}.$$

5. Analogno univerzalnom Turingovom stroju, definira se univerzalna S-funkcija *apply* koja može zamijeniti svaku drugu S-funkciju. Preciznije, ako je f S-funkcija, te f^* prijevod meta-izraza kojim je f definirana onda

   $$f[arg_1; arg_2; ...; arg_n] = apply[f^*; (arg_1, arg_2, ..., arg_n)]$$

   za sve S-izraze arg_1, ..., arg_n.

6. Definira se druga važna funkcija, *eval*, koja izračunava "vrijednost" S-izraza uz zadane "vrijednosti" varijabli. Primjerice, ako S-funkcija *append* "pripisuje" S-izraze onda

eval[(APPEND,X,Y); ((X,(A)),(Y,((B))))] = (A,(B)).

7. Opisana je, u grubim crtama, implementacija sistema. Od posebnog značaja je implementacija simboličkih izraza u obliku koji se danas naziva jednostruko vezane liste, već opisana u prvom memou.

8. Dokazuje se da se svaki program može predstaviti S-funkcijom.

9.1. MATEMATIČKI IZRAZI

Na prvi pogled začuđujuće, znatan dio članka McCarthy je posvetio matematičkim idejama i notaciji.[147] Neke vrste izraza koji su prethodno bili uvedeni kao dio Lispa, sada su uvedeni kao matematički izrazi.

Parcijalne funkcije su funkcije koje nisu definirane na cijelom području domene. *Iskazni izrazi* su izrazi koji mogu biti istiniti ili lažni, odnosno, čije vrijednosti su istina (označena znakom *T*) ili laž (označena znakom *F*). *Predikati* su funkcije koje, za one vrijednosti argumenata za koje su definirane, imaju vrijednost *T* ili *F*.

9.2. UVJETNI IZRAZI

U definicijama funkcija koje se različito izračunavaju za različite argumente obično se koriste fraze na prirodnom jeziku ili izrazi dvodimenzionalnog oblika poput

$$|x| = \begin{cases} -x & \text{za } x < 0 \\ 0 & \text{za } x = 0 \\ x & \text{za } x > 0 \end{cases}$$

Za istu svrhu, McCarthy uvodi *uvjetne izraze*. Primjerice, apsolutna vrijednost može se definirati uvjetnim izrazom

$$|x| = (x < 0 \rightarrow -x,\ x = 0 \rightarrow 0,\ x > 0 \rightarrow x).$$

147 McCarthy, *Recursive functions* ..., CACM, 1960., str. 184.

Uvjetni izrazi se mogu koristiti i van konteksta definicije funkcija. Primjerice,

$$(2 + 2 = 5 \rightarrow 2^2 + 3^3, 2 + 2 = 4 \rightarrow 4).$$

Želi li se izračunati vrijednost uvjetnog izraza, nije potrebno izračunati sve podizraze. Primjerice, u prethodnom izrazu potrebno je izračunati je li $2 + 2 = 5$. Kako to nije slučaj, izračunavanje $2^2 + 3^3$ nije potrebno.

Općenitije, uvjetni izrazi imaju oblik

$$(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n).$$

gdje su $p_1, ..., p_n$ iskazni izrazi, a $e_1, ..., e_n$ bilo koji izrazi. Vrijednost uvjetnog izraza izračunava se tako da se izračunavaju $p_1, ..., p_n$ redom, dok se ne pronađe prvi p_i koji ima vrijednost T. Tada se izračunava odgovarajući e_i i dobiveni rezultat je vrijednost cijelog uvjetnog izraza. Ako su svi $p_1, ..., p_n$ lažni ili neki od podizraza koje treba izračunati nema definiranu vrijednost, onda vrijednost uvjetnog izraza nije definirana.

Umjesto posljednjeg uvjeta se često koristi T. Primjerice,

$$|x| = (x < 0 \rightarrow -x, x = 0 \rightarrow 0, T \rightarrow x).$$

Uvjetni izrazi mogu, primjerice, zamijeniti sve iskazne veznike.

$$\begin{aligned} p \wedge q &= (p \rightarrow q, T \rightarrow F) \\ p \vee q &= (p \rightarrow T, T \rightarrow q) \\ \sim p &= (p \rightarrow F, T \rightarrow T) \\ p \supset q &= (p \rightarrow q, T \rightarrow T). \end{aligned}$$

9.3. REKURZIVNO I SIMULTANO REKURZIVNO DEFINIRANE FUNKCIJE

Funkcije se mogu definirati izrazima u kojima se ime funkcije pojavljuje i s lijeve i s desne strane jednakosti, primjerice

$$n! = (n = 0 \rightarrow 1,\ T \rightarrow n \cdot (n - 1)!).$$[148]

Funkcije se ponekad, iako ne često, definiraju "simultanom rekurzijom". Primjerice, funkcije f, g i h mogu se definirati izrazima

$$f(n) = g(n - 1) + h(n - 1)$$
$$g(n) = (n = 1 \rightarrow 1,\ T \rightarrow f(n) + h(n - 1))$$
$$h(n) = (n = 1 \rightarrow 2,\ T \rightarrow f(n) + g(n - 1)).$$

McCarthy je spomenuo da su simultano rekurzivne funkcije moguće i u Lispu i da će ih se koristiti ako je potrebno.[149] U važnom primjeru opisanom kasnije (funkcije *eval*, *evlis* i *evcon*) simultana rekurzija je zaista korištena.

9.4. LAMBDA-IZRAZI

Funkcije se obično definiraju prije korištenja u izrazima. Primjerice, funkcija se prvo definira izrazom poput

$$f(x) = \sin(x) + \cos(x)$$

a onda se koristi u izrazima, primjerice, $f(3)$. Da se brojevi tretiraju na isti način, umjesto $x = 2 + (3 + 4)$ morali bi pisati nešto poput

$$a = 3 + 4$$

$$x = 2 + a.$$

Pod utjecajem Churchovog lambda računa McCarthy uvodi *lambda-izraze*, koji omogućuju definiranje i korištenje funkcija u istom izrazu, bez uvođenja novog imena. Primjerice, lambda-izraz kojim se definira funkcija jednaka f je

$$\lambda((x), \sin(x) + \cos(x))$$

148 Valja razlikovati *rekurzivne definicije* od *rekurzivne aktivacije*, primjerice, izraza $f(f(x))$.

149 McCarthy, *Recursive functions* ..., CACM, 1960., str. 185.

a umjesto $f(3)$ piše se

$$\lambda((x), \sin(x) + \cos(x))(3).$$

Općenitije, ako je e matematički izraz, a $x_1, ..., x_n$ su varijable, onda je izrazom

$$\lambda((x_1, ..., x_n), e)$$

definirana funkcija koja n-torki $(c_1, ..., c_n)$ pridružuje vrijednost izraza e u kojem $x_1, ..., x_n$ imaju vrijednost $c_1, ..., c_n$ redom.

9.5. LABEL-IZRAZI

Funkcije se mogu definirati *rekurzivno*, izrazima u kojima se ime funkcije pojavljuje s lijeve i desne strane jednakosti. Primjerice,

$$fact(n) = (n = 0 \rightarrow 1, T \rightarrow n \cdot fact(n - 1))$$

Kako rekurzivna funkcija poziva samu sebe, na prvi pogled se čini da se takva funkcija ne može definirati lambda-izrazima. McCarthy zato uvodi Rochesterove lambda-izraze, nešto izmjenjene i preimenovane u *label-izraze*, kojima se i rekurzivne funkcije mogu definirati i koji se mogu koristiti unutar drugih izraza. Primjerice, funkcija koja izračunava faktorijele može se definirati izrazom

$$label(fact, \lambda((n), (n = 0 \rightarrow 1, T \rightarrow n \cdot fact(n - 1)))).$$

Ime funkcije, *fact*, korišteno u label-izrazu nema vrijednosti izvan tog label-izraza. Primjerice, u izrazu

$$label(fact, \lambda((n), (n = 0 \rightarrow 1, T \rightarrow n \cdot fact(n - 1))))(5) + fact(10)$$

drugi pribrojnik nije definiran.

Općenitije, ako je e matematički izraz u kojem se pojavljuje poziv funkcije f od n varijabli, $x_1, ..., x_n$ su varijable, onda je izrazom

$$label(f, \lambda(x_1, ..., x_n), e))$$

definirana funkcija koja n-torki $(c_1, ..., c_n)$ pridružuje vrijednost izraza e u kojem $x_1, ..., x_n$ imaju vrijednost $c_1, ..., c_n$ redom. Ime te funkcije unutar izraza e je f.

U vrijeme razvoja Lispa, McCarthy, unatoč inspiriranosti Churchovim lambda-računom, nije pročitao njegovu knjigu do kraja. Da jest, znao bi da uvođenje label-izraza nije nužno, iako su alternativne definicije funkcija bez label-izraza znatno složenije, o čemu je McCarthy kasnije i pisao.[150]

9.6. SLOBODNE I VEZANE VARIJABLE

Pojavljivanja varijabli u izrazima mogu se podijeliti na *slobodna* i *vezana*. Primjerice, sva pojavljivanja varijabli x i y u izrazu $y^2 + x$ su slobodna, a u izrazu $\lambda((x, y), y^2 + x)$ su vezana. U izrazu

$$x + \lambda((x), x^2)(3)$$

prvo pojavljivanje varijable x je slobodno, a drugo i treće je vezano.

Općenito, sva pojavljivanja varijabli $x_1, ..., x_n$ u $\lambda((x_1, ..., x_n), e)$ su vezana. Ostala pojavljivanja varijabli u $\lambda((x_1, ..., x_n), e)$ su slobodna (vezana) ako su slobodna (vezana) u izrazu e. Pojavljivanje varijable f u izrazu $label(f, \lambda((x_1, ..., x_n), e))$ je vezano.

U ostalim izrazima (onima koji nisu lambda- ili label-izrazi) pojavljivanja varijabli su slobodna (vezana) ako su slobodna (vezana) u podizrazima. Trivijalno pojavljivanje varijable (primjerice, x je varijabla u izrazu x) je slobodno.

Tipično, za izračunavanje vrijednosti izraza potrebno je znati vrijednosti slobodnih varijabli. Primjerice, za izračunavanje $f(x) \cdot f(x)$ potrebno je znati vrijednost x i definiciju f. Ponekad se izrazi mogu izračunati bez da se znaju vrijednosti slobodnih varijabli. Primjerice, za izračunavanje $f(x) - f(x)$ treba znati samo da je f definirana za vrijednost x.

150 Vidi poglavlje *Temelji matematičke teorije izračunavanja*.

9.7. PRIMJEDBA O SIMULTANO REKURZIVNIM FUNKCIJAMA

McCarthy nije razmatrao simultanu rekurziju, osim što je dao njenu definiciju. Na prvi pogled, čini se da se simultanom rekurzijom mogu definirati funkcije koje se bez nje ne bi mogle definirati. To, međutim, nije točno. Neka je

$$f_1(x_1, ..., x_n) = e_1,$$
$$...$$
$$f_k(x_k, ..., x_n) = e_k.$$

sustav simultano rekurzivnih definicija funkcija, pri čemu su e_1, …, e_k izrazi koji sadrže f_1, …, f_k. Tada je u cijelom sustavu moguće zamijeniti izraze oblika

$$f_1(a_1, ..., a_n) \text{ s } g(1, a_1, ..., a_n),$$
$$...$$
$$f_k(a_1, ..., a_n) \text{ s } g(k, a_1, ..., a_n)$$

pa sustav prelazi u rekurzivnu definiciju funkcije g

$$g(i, x_1, ..., x_n) = \{i = 1 \rightarrow es_1, ..., i = k \rightarrow es_k\}.$$

gdje su es_1, …, es_k rezultati supstitucije u izrazima e_1, …, e_k. Funkcija g se može definirati label-izrazom. Nadalje, funkcije f_1, …, f_k mogu se definirati ne-rekurzivnim definicijama

$$f_1(x_1, ..., x_n) = g(1, x_1, ..., x_n),$$
$$...$$
$$f_k(x_k, ..., x_n) = g(k, x_1, ..., x_n).$$

9.8. DEFINICIJA SIMBOLIČKIH IZRAZA

"Čisti Lisp" ima samo jednu vrstu podataka: *simboličke izraze* ili *S-izraze*. Može se reći da su simbolički izrazi osnova Lispa, onako kao što su skupovi osnova matematike. Simbolički izrazi su podskup skupa svih mogućih nizova znakova, odabran zbog pogodnosti za izražavanje matematičkih i logičkih formula.

Atomski simboli ili, kraće, simboli su konačni nizovi velikih slova, znamenki i jednostrukih razmaka. Primjerice, `A`, `ABA` i `APPLE PIE NUMBER 3` su simboli.

Niz znakova *e* je *simbolički izraz* ako:

1. *e* je atomski simbol, ili

2. $e = (e_1 \cdot e_2)$ pri čemu su e_1 i e_2 simbolički izrazi.

Primjerice, `(A·B)` i `((A·B)·XYZ)` su simbolički izrazi, ali ne i simboli. Simbolički izrazi oblika $(e_1 \cdot e_2)$ nazivaju se *uređeni parovi* (engl. *ordered pairs*) ili točkasti parovi (engl. *dotted pairs*) simboličkih izraza e_1 i e_2.

Niz znakova $(e_1, e_2, \ldots, e_n)$ gdje su e_1, e_2, ..., e_n simbolički izrazi naziva se *lista* simboličkih izraza e_1, e_2, ..., e_n. Lista $(e_1, e_2, \ldots, e_n)$ je "pokrata" (engl. *abbreviation*) za simbolički izraz

$$(e_1 \cdot (e_2 \ (\ldots(e_n \cdot \mathtt{NIL})\ldots))).$$

Primjerice, `(A,B)` je pokrata za `(A·(B·NIL))`. Po definiciji, `()` je pokrata za `NIL`.

McCarthy i liste naziva simboličkim izrazima[151] što je nepreciznije, ali kraće i nije potreban veliki napor da se izbjegnu zabune.

9.9. RAZLIKA U ODNOSU NA PRETHODNE DEFINICIJE

Definicija simboličkih izraza u članku iz 1960. razlikuje se od prethodnih. Uvedeni su uređeni parovi, manje pogodni za predstavljanje matematičkih i logičkih formula, a izrazi oblika $(e_1, e_2, \ldots, e_n)$ su degradirani u pokrate za simboličke izraze. McCarthy nije objasnio razloge za tu promjenu, što je razumljivo: "stare definicije" simboličkih izraza nisu objavljene i rasprava bi samo zbunila tadašnje čitaoce.

151 "Since we regard the expressious with commas as abbreviations for those not involving commas, we shell refer to them all as S-expressions."
McCarthy, *Recursive functions* ..., CACM, 1960., str. 187.

Razlozi se mogu naslutiti iz definicije simboličkih izraza u Memou 8. koja je odbačena u članku iz 1960.[152]

1. Atomski simboli su simbolički izrazi.

2. Nul-izraz, označen s Λ, je simbolički izraz.

3. Ako je *e* simbolički izraz onda je i `(`*e*`)` simbolički izraz.

4. Ako su e_1 i `(`e_2`)` simbolički izrazi onda je i `(`e_1`,`e_2`)` simbolički izraz.

Definicija iz Memoa 8. nije potpuno korektna. Po njoj, primjerice, `(A,B,C)` nije simbolički izraz što zacijelo nije bila McCarthyjeva intencija. Korektna definicija koja bi odgovarala McCarthyjevoj namjeri bi možda izgledala ovako:

1'. Atomski simboli su simbolički izrazi.

2'. Nul-izraz (može se koristiti i oznaka `()` umjesto Λ) je simbolički izraz.

4'. Ako su e_0 i `(`e_1`,`…`,`e_n`)`, $n \geq 0$, simbolički izrazi onda je i `(`e_0`,`e_1`,`…`,`e_n`)` simbolički izraz.

Opisana pogreška u Memou 8. čini se previdom, nedovršenom analizom koja nam omogućuje da vidimo što je autor htio: definirati simboličke izraze tako da se izgrađuju funkcijom `cons`, a ne, kao u "imperativnom Lispu", problematičnom funkcijom `list`.

Nedostatak korigirane definicije iz Memoa 8. je da se `cons` se ne bi mogla primijeniti na sve simboličke izraze; drugi argument ne smije biti simbol. Definicija iz članka iz 1960. je gotovo sigurno posljedica želje da se funkcija `cons` definira na svim parovima simboličkih izraza, a da se pri tome zadrže liste.

Nažalost, definicijom simboličkih izraza u članku iz 1960. termin *lista* postao je višeznačan. Listom se sada nazivaju i nizovi znakova oblika `(`e_1`,`…`,`e_n`)` i interna reprezentacija takvih simboličkih izraza u memoriji pa je ponekad potrebno naglasiti o kojoj vrsti lista se radi.

152 McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 3.

9.10. META-IZRAZI

Kao što su "aritmetički izrazi" izrazi "o brojevima", tako su *meta-izrazi* (*M-izrazi*) izrazi "o" S-izrazima. McCarthy meta-izraze opisuje kao obične matematičke izraze, koji koriste "konvencionalnu funkcijsku notaciju" uz nekoliko nebitnih izmjena. Umjesto zareza pišu se točka-zarezi, umjesto okruglih zagrada pišu se uglate zagrade, u imenima funkcija i varijabli nisu dozvoljena velika slova.[153] Svrha tih izmjena je razlikovanje meta-izraza od S-izraza. Oznake za istinu i neistinu u meta-izrazima su S-izrazi T i F.

Simbolički izrazi su poseban, trivijalan oblik meta-izraza, slično kao što su brojevi trivijalan oblik aritmetičkih izraza. Tako je, primjerice, T simbolički izraz, ali ujedno i meta-izraz.

9.11. S-FUNKCIJE

McCarthy definira "klasu" *parcijalno definiranih funkcija* na S-izrazima koju naziva *S-funkcijama*. S-funkcije definira u dva koraka. Prvo, popisuje pet elementarnih S-funkcija. Zatim, pokazuje kako se uz pomoć već definiranih funkcija mogu definirati nove S-funkcije. Drugih S-funkcija, osim navedenih, nema. S-funkcije se podudaraju sa svim *izračunljivim funkcijama* na S-izrazima. Dokaz te tvrdnje slijedi iz mogućnosti definiranja Turingovih strojeva u Lispu.

153 "We now define a class of functions of S-expressions. The expressions representing these functions are written in a conventional functional notation. However, in order to clearly distinguish the expressions representing functions from S-expressions, we shall use sequences of lower-case letters for function names and variables ranging over the set of S-expressions. We also use brackets and semicolons, instead of parentheses and commas, for denoting the application of functions to their arguments. Thus we write car[x], car[cons[(A·B); x]]. In these M-expressions (meta-expressions) any S-expression that occur stand for themselves."
McCarthy, *Recursive functions* …, CACM, 1960., str. 187.

1. Elementarna S-funkcija *atom*. Primjerice,

$$atom[\texttt{X}] = \texttt{T},$$

$$atom[\texttt{(X·Y)}] = \texttt{F}.$$

Vrijednost *atom*[*x*] je definirana za sve S-izraze *x* i ima vrijednost `T` ako je *x* atomarni simbol, `F` inače. Na prvi pogled neobično, *atom*[`()`] = `T` jer je `()` tek pokrata za simbol `NIL`.

2. Elementarna S-funkcija *eq*. Primjerice,

$$eq[\texttt{X}; \texttt{Y}] = \texttt{F},$$

$$eq[\texttt{X}; \texttt{X}] = \texttt{T},$$

eq[`X`; `(X·Y)`] nije definirana.

Općenitije, vrijednost *eq*[*x*; *y*] je definirana samo ako su *x* i *y* atomarni simboli te ima vrijednost `T` ako su *x* i *y* isti simboli, inače `F`.

3. Elementarna S-funkcija *car*. Primjerice,

$$car[\texttt{(A·B)}] = \texttt{A},$$

$$car[\texttt{(A,B,C)}] = \texttt{A}.$$

Vrijednost *car*[*e*] je definirana samo ako je *e* ne-atomarni simbolički izraz, tj. *e* = `(`*x*·*y*`)`. Tada

$$car[\texttt{(}x\cdot y\texttt{)}] = x.$$

4. Elementarna S-funkcija *cdr*. Primjerice,

$$cdr[\texttt{(A·B)}] = \texttt{B},$$

$$cdr[\texttt{(A,B,C)}] = \texttt{(B,C)}.$$

Vrijednost *cdr*[*e*] je definirana samo za ne-atomarni simboličke izraze e, tj. $e = (x \cdot y)$ i

$$cdr[(x \cdot y)] = y.$$

5. Elementarna S-funkcija *cons*. Primjerice,

$$cons[A; B] = (A \cdot B),$$

$$cons[A; (B,C)] = (A,B,C).$$

Općenitije, $cons[x; y] = (x \cdot y)$ za sve S-izraze x i y.

Da su simbolički izrazi definirani kao u prvim verzijama Lispa, drugi argument ne bi smio biti simbol.

McCarthy ističe važan odnos S-funkcija *car*, *cdr* i *cons*.

$$car[cons[x; y]] = x$$

$$cdr[cons[x; y]] = y$$

Ako x nije atomarni simbol, onda i

$$cons[car[x]; cdr[x]] = x.$$

Mogli bismo se upitati zašto je McCarthy odabrao baš te elementarne funkcije, a ne neke druge. Sam McCarthy nije pokušao odgovoriti na to pitanje. Sigurno je da je sve funkcije koje je McCarthy prethodnih mjeseci pokušao definirati uspio definirati uz pomoć pet elementarnih funkcija. Znamo i da su sve elementarne funkcije "izračunljive:" za zadane argumente funkcije, čovjek može izračunati vrijednost funkcije. Štoviše, može to učiniti u ograničenom vremenu, linearno ovisnom o duljini argumenata.

Elementarne funkcije nisu nezavisne. Primjerice, *cons* se može definirati pomoću *car* i *cdr*:

$cons[x; y] = z$ takav da $car[z]=x$ i $cdr[z]=y$.

Isto tako, *car* i *cdr* mogu definirati pomoću *cons*:

$car[x] = z$ takav da postoji y takav da $cons[z; y] = x$

$cdr[x] = y$ takav da postoji z takav da $cons[z; y] = x$

Te su definicije korektne, ali ne daju postupak za izračunavanje *cons*[*x*; *y*]. Da je prihvatio takvu metodu definiranja S-funkcija, McCarthy bi "izgubio izračunljivost" S-funkcija.

9.13. PRIMJEDBE UZ MCCARTHYJEVO DEFINIRANJE S-FUNKCIJA

McCarthyjev opis načina na koji se ostale S-funkcije mogu definirati pomoću elementarnih je štur[154] i ostavlja nedoumice, koje se, srećom, otklanjaju brojnim primjerima. Primjerice, S-funkcija *ff* je definirana za sve izraze i vraća prvi element u S-izrazu, ignorirajući zagrade. Primjerice,

$$ff[((\mathtt{A}\cdot\mathtt{B})\cdot(\mathtt{C}\cdot\mathtt{D}))] = \mathtt{A}.$$

Ta se S-funkcija definira meta-izrazom

$$ff[e] = [atom[e] \rightarrow e;\ \mathtt{T} \rightarrow ff[car[e]]].$$

Općenitije, S-funkcije se definiraju meta-izrazima oblika

$$f[x_1; ...; x_n] = e,$$

gdje je *f* ime funkcije, x_1, ..., x_n su varijable, *e* je meta-izraz izgrađen kompozicijom S-funkcija, predikata jednakosti, logičkim veznicima, uvjetnim, lambda- i label- izrazima. Ako je *f* rekurzivna funkcija, onda se ime te funkcije pojavljuje i s desne strane jednakosti.

154 "Compositions of car and cdr give the subexpressions of a given expression in a given position. Compositions of cons form expressions of a given structure out of parts. The class of functions which can be formed in this way is quite limited and not very interesting. (...) We get a much larger class of functions (in fact, all computable functions) when we allow ourselves to form new functions of S-expressions by conditional expressions and recursive definition." McCarthy, *Recursive functions* ..., CACM, 1960., str. 187.

McCarthy sugerira[155] da je S-funkcije moguće definirati i izrazima poput

$$ff = label[ff; \lambda[[e]; [atom[e] \rightarrow e; \mathsf{T} \rightarrow ff[car[e]]]]].$$

Općenitije, S-funkcije se definiraju i meta-izrazima oblika

$$f = label[f, \lambda[[x_1; ...; x_n]; e]],$$

$$f = \lambda[[x_1; ...; x_n]; e],$$

gdje su f, x_1, ..., x_n i e isti kao i za oblik $f[x_1; ...; x_n] = e$.

U kasnijim tekstovima, primjerice, Slagleovoj dizertaciji[156], oba načina se eksplicitno navode, a podržani su i u mnogim, možda i svim implementacijama jezika. Čini se prikladnim ta dva načina definiranja funkcija nazvati redom, *implicitnom* i *eksplicitnom definicijom funkcije*; terminima koje McCarthy nije koristio.

Metode definiranja S-funkcija ipak ne obuhvaćaju sve metode koje se koriste u matematici. Primjerice, izrazom

$$f[x] = [[\exists y][cons[y; y] = x] \rightarrow \mathsf{T}, \mathsf{T} \rightarrow \mathsf{F}]$$

ne može se definirati S-funkcija. Zajedničko svim metodama kojima se S-funkcije mogu definirati je da su konstruktivne, tj. definicija S-funkcije daje i postupak kojim se može izračunati vrijednost funkcije, ako takva vrijednost postoji.

Valja uočiti da se ista S-funkcija može definirati različitim meta-izrazima. Primjerice, funkcija *ff* se može definirati izrazom

$$ff = label[ff; \lambda[[e]; [atom[e] \rightarrow e; \mathsf{T} \rightarrow ff[car[e]]]]]$$

155 "... function whose M-expression is *label*[*subst*; λ[[*x*; *y*; *z*]; [...]]] has the S-expression ..."
McCarthy, *Recursive functions* ..., CACM, 1960., str. 189.

156 Slagle, *A heuristic program* ..., Ph.D. thesis, 1961., str. 18.

kao i izrazom

$$ff = label[ff; \lambda[[e]; [\sim atom[e] \rightarrow ff[car[e]]; \mathtt{T} \rightarrow e]]].$$

9.14. PRIMJERI NE-ELEMENTARNIH S-FUNKCIJA

McCarthy navodi brojne primjere S-funkcija. Primjeri su važni jer pokazuju način na koji Lisp programeri razmišljaju, i omogućuju shvaćanje autorove intencije.

1. S-funkcije *caar*, *cadr*, *cdar*, *cddr*, *caaar*, *caadr* ... su definirane meta-izrazima

$$caar[x] = car[car[x]],$$
$$cadr[x] = car[cdr[x]],$$
$$cdar[x] = cdr[car[x]],$$
$$\dots$$

Primjerice,

$$cadr[\mathtt{(A,B,C)}] = car[cdr[\mathtt{(A,B,C)}]] = car[\mathtt{(B,C)}] = \mathtt{B}.$$

McCarthy S-funkcije *caar*, *cadr*, *cdar*, ... naziva i "pokratama".

2. S-funkcija *ff*. Vrijednost *ff*[*e*] je prvi element u S-izrazu *e*, ignorirajući zagrade. Primjerice,

$$ff[\mathtt{((A \cdot B) \cdot (C \cdot D))}] = \mathtt{A}.$$

S-funkcija *ff* definirana je meta-izrazom

$$ff[e] = [atom[e] \rightarrow e; \mathtt{T} \rightarrow ff[car[e]]].$$

3. S-funkcija *subst*. Vrijednost *subst*[*x*;*y*;*z*] je rezultat "uvrštavanja" izraza *x* umjesto svake pojave atomskog simbola *y* u izrazu *z*. Primjerice,

$$subst[\mathtt{(A \cdot B)}; \mathtt{C}; \mathtt{((C \cdot D) \cdot E)}] = \mathtt{(((A \cdot B) \cdot D) \cdot E)}.$$

S-funkcija *subst* definirana je meta-izrazom

$$subst[x; y; z] = [atom[z] \rightarrow [eq[z; y] \rightarrow x; \\ \mathtt{T} \rightarrow z]; \\ \mathtt{T} \rightarrow cons[subst[x; y; car[z]]; \\ subst[x; y; cdr[z]]]].$$

4. S-funkcija *equal*. Vrijednost *equal*[*x*; *y*] definirana je za sve simboličke izraze *x* i *y* i jednaka je `T` ako su *x* i *y* isti S-izraz, znak po znak, `F` inače. Primjerice,

$$equal[\mathtt{(A \cdot B)}; \mathtt{(A \cdot B)}] = \mathtt{T}.$$

S-funkcija *equal* definirana je meta-izrazom

$$equal[x; y] = [[atom[x] \wedge atom[y] \wedge eq[x; y]] \vee \\ \sim atom[x] \wedge \sim atom[y] \wedge equal[car[x]; car[y]] \\ \wedge equal[cdr[x]; cdr[y]]]].$$

5. S-funkcija *null*. Vrijednost *null*[*x*] je `T` ako je *x* jednak `NIL`; `F` inače. S-funkcija *null* definirana je meta-izrazom

$$null[x] = atom[x] \wedge eq[x; \mathtt{NIL}].$$

6. S-funkcija *append*. Primjerice,

$$append[\mathtt{(A,B,C)}; \mathtt{(A,C,A)}] = \mathtt{(A,B,C,A,C,A)}.$$

S-funkcija *append* definirana je meta-izrazom

$$append[x; y] = [null[x] \rightarrow y; \\ \mathtt{T} \rightarrow cons[car[x]; append[cdr[x]; y]]].$$

7. S-funkcija *among*. Vrijednost *among*[*x*; *y*] je `T` ako se *x* pojavljuje kao element liste *y*; `F` inače. Primjerice,

$$among[\mathtt{X}; \mathtt{(A,B,X,C)}] = \mathtt{T}.$$

S-funkcija *among* definirana je meta-izrazom

$$among[x; y] = [\sim null[y] \wedge [equal[x; car[y]] \\ among[x; cdr[y]]]].$$

8. S-funkcija *pair*. Primjerice,

$$pair[\texttt{(A,B,C)}; \texttt{(X,(Y,Z),U)}] = \texttt{((A,X),(B,(Y,Z)),(C,U))}.$$

S-funkcija pair definirana je meta-izrazom

$$pair[x; y] = [null[x] \wedge null[y] \rightarrow \texttt{NIL}; \\ \sim atom[x] \wedge \sim atom[y] \rightarrow cons[list[car[x]; car[y]]; \\ pair[cdr[x]; cdr[y]]]].$$

gdje je $list[x; y]$ pokrata za $cons[x; cons[y; \texttt{NIL}]] = \texttt{(}x\texttt{,}y\texttt{)}$.

9. S-funkcija *assoc*. Varijablama se često asociraju vrijednosti, primjerice, $x = b$, $y = \sin b$. Informacije o asocijacijama se mogu kodirati S-izrazima poput `((X,B),(Y,(SIN,B)))`. Vrijednost $assoc[s; a]$ je izraz asociran sa simbolom s u listi asocijacija a; prvi ako takvih asocijacija ima više. Primjerice,

$$assoc[\texttt{X;((X,B),(Y,(SIN,B)),(X,C))}] = \texttt{A}.$$

S-funkcija *assoc* definirana je meta-izrazom

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

Ako su S-izrazi implementirani kao jednostruko vezane liste, onda je izračunavanje vrijednosti S-funkcije *assoc* u nekim slučajevima neefikasno iz razloga na koje je McCarthy upozorio u prvom memou: primjerice, za izračunavanje

$$assoc[\texttt{X,((A,}e_1\texttt{),(B,}e_2\texttt{),...,(Z,}e_{26}\texttt{))}]$$

funkcija *assoc* treba 23 puta pozvati samu sebe.

10. S-funkcija *sub2*. Vrijednost *sub2*[*x*; *y*] je prvi simbol u listi asocijacija *x* koji je asociran s *y*. Ako takav ne postoji, onda ostaje *y*.

$$sub2[\texttt{((X,A),(Y,(B,C)),(Z,D))}; \texttt{Y}] = \texttt{(B,C)},$$
$$sub2[\texttt{((X,A),(Y,B))}; \texttt{Z}] = \texttt{Z}.$$

S-funkcija *sub2* definirana je meta-izrazom

$$sub2[a; x] = [null[a] \rightarrow x;$$
$$eq[caar[a]; x] \rightarrow cadar[a];$$
$$\texttt{T} \rightarrow sub2[cdr[a]; x]].$$

11. S-funkcija *sublis*. S-funkcija *sublis* je generalizacija *sub2*; vrijednost *sublis*[*x*; *y*] je rezultat supstitucije atomarnih simbola u S-izrazu *y* vrijednostima asociranima u listi *x*. Primjerice,

$$sublis[\texttt{((X,(A,B)),(Y,(B,C)))}; \texttt{(A,(X·Y))}] =$$
$$\texttt{(A,((A,B)·(B,C)))}.$$

S-funkcija *sublis* definirana je meta-izrazom

$$sublis[a; x] = [atom[x] \rightarrow sub2[a; x];$$
$$\texttt{T} \rightarrow cons[sublis[a; car[x]];$$
$$sublis[a; cdr[x]]]].$$

12. S-funkcije *first*, *rest*, *second*, *third* … Iako je McCarthy odustao od pokušaja da se imena *car*, *cdr* i *cons* zamijene s *first*, *rest* i *combine* ništa ne priječi definiranje tih funkcija.

$$first[l] = car[l]$$
$$rest[l] = cdr[l]$$

Analogno, često se definiraju i funkcije

$$second[l] = cadr[l]$$
$$third[l] = caddr[l]$$
$$\ldots$$

9.15. POKRATA I FUNKCIJA LIST

McCarthy uvodi i važnu *funkciju* i *pokratu list*.[157] Primjerice,

$$list[\texttt{A}; \texttt{B}; \texttt{(A,B)}; \texttt{C}] = \texttt{(A,B,(A,B),C)}.$$

Funkcija je definirana izrazom

$$list[e_1; \ldots; e_n] = cons[e_1; cons[e_2; \ldots; cons[e_n; \texttt{NIL}]\ldots]].$$

Definicija je matematički korektna, ali se razlikuje od prethodnih: sadrži tri točke – koje se ne mogu ukloniti jer *list* mora biti definirana za proizvoljan broj argumenata. Iz toga slijedi da *list* nije S-funkcija, iako McCarthy to nije eksplicitno napisao. Još uvijek, *list* se može koristiti u meta-izrazima kao pokrata, dok god ju je moguće zamijeniti kompozicijom nekoliko primjena funkcije *cons*.

9.16. FUNKCIJE KAO ARGUMENTI FUNKCIJA

McCarthy ističe da se mogu definirati mnoge "korisne" funkcije koje primaju druge funkcije za argumente. Navodi primjer funkcije *maplist* koja listi $(l_1, l_2, \ldots, l_n)$ i funkciji *f* pridružuje listu

$$(f[(l_1, l_2, \ldots, l_n)], f[(l_2, \ldots, l_n)], \ldots, f[(l_n)]).$$

Primjerice,

$$maplist[\texttt{(A,B,C)}; \lambda[[x]; x]] = \texttt{((A,B,C),(B,C),(C))}.$$

157 "Another useful abbreviation is to write $list[e_1; \ldots; e_n]$ for $cons[e_1; cons[e_2; \ldots; cons[e_n; \texttt{NIL}]\ldots]]$. This function gives the list, $(e_1, \ldots, e_n)$ as a function of its elements."
McCarthy, *Recursive functions ...*, *CACM*, 1960., str. 188.

Definicija *maplist* je

$$maplist[l; f] = [null[l] \rightarrow \texttt{NIL};\ \texttt{T} \rightarrow cons[f[l]; maplist[cdr[l]; f]]].$$

Funkcije koje primaju druge funkcije kao argumente su posebno korisne za definiranje drugih funkcija, što McCarthy pokazuje primjerom funkcije dvije varijable, *diff*. Vrijednost, *diff*[*y*; *x*] je derivacija izraza *y* po varijabli *x*. Izraz *y* može biti atomski simbol ili simbolički izraz oblika (PLUS,e_1,...,e_n) ili (TIMES,e_1,...,e_n), gdje su e_1, ..., e_n simbolički izrazi koji mogu biti prvi argument funkcije *diff*.

Drugi argument funkcije *diff* je simbol po kojem se derivira.

$$\begin{aligned}
diff[y; x] = [&atom[y] \rightarrow [eq[y; x] \rightarrow \texttt{ONE}; \texttt{T} \rightarrow \texttt{ZERO}\ ];\\
&eq[car[y]; \texttt{PLUS}] \rightarrow cons[\texttt{PLUS};\\
&\qquad maplist[cdr[y];\\
&\qquad\ \lambda[[z]; diff[car[z]; x]]]];\\
&eq[car[y]; \texttt{TIMES}]] \rightarrow\\
&\quad cons[\texttt{PLUS};\\
&\qquad maplist[cdr[y];\\
&\qquad\ \lambda[[z];\\
&\qquad\quad cons[\texttt{TIMES};\\
&\qquad\qquad maplist[cdr[y];\\
&\qquad\qquad\ \lambda[[w];\\
&\qquad\qquad\quad [\sim eq[z; w] \rightarrow car[w];\\
&\qquad\qquad\ \texttt{T} \rightarrow diff[car[w]; x]]]]]]]]]
\end{aligned}$$

9.17. PRIMJEDBA O ELIMINIRANJU POKRATA I NE-ELEMENTARNIH S-FUNKCIJA

Prilikom definiranja novih funkcija mogu se koristiti prethodno definirane ne-elementarne S-funkcije. Iako McCarthy to nije eksplicirao, radi razumijevanja kasnijih McCarthyjevih definicija valja uočiti da se svaka S-funkcija koja se može definirati uz korištenje pokrata, iskaznih operatora i imena prethodno definiranih ne-elementarnih funkcija može definirati i bez tih elemenata. Primjerice, iz definicije

$$pair[x; y] = [[null[x] \wedge null[y]] \rightarrow \texttt{NIL};$$
$$[\sim atom[\mathrm{x}] \wedge \sim atom[\mathrm{y}]] \rightarrow cons[list[car[x]; car[y]];$$
$$pair[cdr[x]; cdr[y]]]]$$

se mogu eliminirati operatori $\wedge$ i ~ te pokrata *list*. Dobiva se definicija

$$pair[x; y] = [[null[x] \rightarrow null[y]; \texttt{T} \rightarrow \texttt{F}] \rightarrow \texttt{NIL};$$
$$[[atom[x] \rightarrow \texttt{F}; \texttt{T} \rightarrow \texttt{T}] \rightarrow [atom[y] \rightarrow \texttt{F}; \texttt{T} \rightarrow \texttt{T}];$$
$$\texttt{T} \rightarrow \texttt{F}] \rightarrow cons[cons[car[x]; cons[car[y]; \texttt{NIL}]];$$
$$pair[cdr[x]; cdr[y]]]].$$

S-funkcija *null* "ima" M-izraz

$$\lambda[[x]; atom[x] \wedge eq[x; \texttt{NIL}]]$$

iz kojeg se eliminacijom logičkog veznika $\wedge$ dobiva

$$\lambda[[x]; [atom[x] \rightarrow eq[x; \texttt{NIL}]; \texttt{T} \rightarrow \texttt{F}]].$$

Uvrsti li se taj izraz umjesto *null* u definiciju *pair* dobiva se

$$pair[x; y] = [[\lambda[[x]; [atom[x] \rightarrow eq[x; \texttt{NIL}]; \texttt{T} \rightarrow \texttt{F}]][x] \rightarrow$$
$$\lambda[[x]; [atom[x] \rightarrow eq[x; \texttt{NIL}]; \texttt{T} \rightarrow \texttt{F}]][y];$$
$$\texttt{T} \rightarrow \texttt{F}] \rightarrow \texttt{NIL};$$
$$[[atom[\mathrm{x}] \rightarrow \texttt{F}; \texttt{T} \rightarrow \texttt{T}] \rightarrow [atom[\mathrm{y}] \rightarrow \texttt{F}; \texttt{T} \rightarrow \texttt{T}];$$
$$\texttt{T} \rightarrow \texttt{F}] \rightarrow cons[cons[car[x]; cons[car[y]; \texttt{NIL}]];$$
$$pair[cdr[x]; cdr[y]]]].$$

Konačna definicija funkcije *pair*, iako daleko manje razumljiva, ekvivalentna je početnoj i ne sadrži imena ne--elementarnih funkcija niti "pokrate".

Funkcije koje su definirane simultanom rekurzijom bi se mogle eliminirati tek nakon što se simultano rekurzivne definicije zamijene prethodno opisanim definicijama ili nekim drugim ekvivalentnim, ali ne i simultano rekurzivnim definicijama.

Naivnim eliminiranjem poziva prethodno definiranih funkcija uvrštavanjem pripadajućih lambda- ili label-izraza, originalna definicija funkcije se može eksponencijalno povećati. Primjerice, neka je

$$f_0[x] = cons[f_1[x]; f_1[cons[x; x]]]$$
$$f_1[x] = cons[f_2[x]; f_2[cons[x; x]]]$$
$$\ldots$$
$$f_{n-1}[x] = cons[f_n[x]; f_n[cons[x; x]]].$$

te na kraju

$$f_n[x] = x.$$

Uvrštanjem f_1 u definiciju f_0 dobiva se

$$f_0[x] = cons[cons[f_2[x]; f_2[cons[x; x]]];$$
$$cons[f_2[cons[x; x]]; f_2[cons[cons[x; x]; cons[x; x]]]]]$$

u kojem se f_2 pojavljuje četiri puta, a nastavi li se s takvim uvrštanjem, funkcija f_n bi se pojavila 2^n puta. Metoda korištena pri eliminaciji simultano rekurzivnih definicija može se koristiti i ovdje. Umjesto n funkcija jedne varijable treba definirati jednu funkciju dvije varijable

$$f[i; x] = [i = 0^* \rightarrow f_0[x];$$
$$\ldots$$
$$i = n^* \rightarrow f_n[x]],$$

gdje su $0^*, \ldots, n^*$ neki S-izrazi. Sada se mogu uvrstiti definicije funkcija $f_0, \ldots, f_n$ i u dobivenom izrazu zamijeniti sve pozive $fi[x]$ s $f[i; x]$. Dobiva se rekurzivna definicija funkcije f. Iz te definicije se f može definirati label izrazom $label[f; \lambda[[i; x]\ldots]]$ i iz $f_0[x]=f[0^*; x]$ slijedi

$$f_0 = \lambda[[x]; label[f; \lambda[[i; x]\ldots]][0^*; x]].$$

Nema bitne razlike i ako su funkcije $f_0, \ldots, f_n$ funkcije više varijabli.

9.18. PREVOĐENJE M-IZRAZA U S-IZRAZE.

Prijevod meta-izraza e u S-izraz se označava s e^*. Postoji šest pravila za prevođenje, ovisno o vrsti meta-izraza koji se prevodi.

1. Simbolički izrazi. Ako je e simbolički izraz onda e^* = (QUOTE,e). Primjerice,

$$\mathtt{T}^* = \mathtt{(QUOTE,T)}.$$

2. Imena funkcija i varijabli. Ako je e ime funkcije ili varijable onda je e^* isto ime, samo što su mala slova zamijenjena velikima. Primjerice,

$$xyz^* = \mathtt{XYZ}.$$

3. Primjene S-funkcija. Neka je $f[e_1; ...; e_n]$ primjena S-funkcije; f je ime funkcije ili izraz kojim se definira funkcija. Tada,

$$f[e_1; ...; e_n]^* = (f^*, e_1^*, ..., e_n^*).$$

Primjerice,

$$cons[\mathtt{T}; xyz]^* = \mathtt{(CONS,(QUOTE,T),XYZ)},$$

$$\lambda[[x]; cons[x; x]][\mathtt{A}]^* = \mathtt{(}\lambda[[x]; cons[x; x]]^*\mathtt{,(QUOTE,A))}$$

4. Uvjetni izrazi. Neka je $[p_1 \rightarrow e_1; ...; p_n \rightarrow e_n]$ uvjetni izraz, gdje su p_i i e_i bilo koji meta-izrazi. Tada

$$[p_1 \rightarrow e_1; ...; p_n \rightarrow e_n]^* = \mathtt{(COND,(}p_1^*\mathtt{,}e_1^*\mathtt{),...,(}p_n^*\mathtt{,}e_n^*\mathtt{))}.$$

Primjerice,

$$[eq[z; y] \rightarrow x; \mathtt{T} \rightarrow z]^* = \mathtt{(COND,((EQ,Z,Y),X),}$$
$$\mathtt{((QUOTE,T),Z))}.$$

Pravilo za prevođenje uvjetnih izraza je McCarthy kasnije kritizirao jer vodi do prevelikog broja zagrada.[158]

158 "Some of the decisions made rather lightheartedly for the "Recursive functions ... " paper later proved unfortunate. These included the cond notation for conditional xpressions, which leads to an unnecessary depth of parentheses, and the use of the number zero to denote the empty list NIL and the truth value false."

5. Lambda-izrazi. Neka je $\lambda[[x_1; ...; x_n], m]$ lambda-izraz; $x_1; ...; x_n$ bilo koje varijable, m bilo koji meta izraz.

$$\lambda[[x_1; ...; x_n]; m]^* = \texttt{(LAMBDA,(}x_1^*\texttt{,...,}x_n^*\texttt{),}m^*\texttt{)}.$$

Primjerice,

$$\lambda[[x]; cons[x; x]]^* = \texttt{(LAMBDA,(X),(CONS,X,X))}.$$

6. Label-izrazi. Neka je $label[a; m]$ label-izraz, a ime funkcije, m lambda izraz. Tada,

$$label[a; m]^* = \texttt{(LABEL,}a^*\texttt{,}m^*\texttt{)}.$$

Primjerice,

$$label[f; \lambda[[e]; [atom[e] \rightarrow e; \texttt{T} \rightarrow f[car[e]]]]]$$

se prevodi u S-izraz

```
(LABEL,(F,(LAMBDA,(E),
             (COND,((ATOM,E),E),
                   ((QUOTE,T),(F,(CAR,E)))))))
```

Nešto veći M-izraz

$$label[subst; \lambda[[x; y; z];
[atom[z] \rightarrow [eq[z; y] \rightarrow x;
\texttt{T} \rightarrow z];
\texttt{T} \rightarrow cons[subst[x; y; car[z]];
subst[x; y; cdr[z]]]]]]$$

prevodi se u S-izraz

McCarthy, *History of LISP*, 1981., str. 179.

```
(LABEL,SUBST,
  (LAMBDA,(X,Y,Z),
    (COND,((ATOM,Z),(COND,((EQ,Z,Y),X),
                          ((QUOTE,T),Z))),
          ((QUOTE,T),(CONS,(SUBST,X,Y,(CAR,Z)),
                           (SUBST,X,Y,(CDR,Z)))))).
```

McCarthy je bio rezerviran prema čitljivosti ovakvih, u S-izraze prevedenih M-izraza[159], dijelom možda i stoga što još nije prihvatio Rochesterovo formatiranje uvlačenjem redaka. Primjerice, gornji S-izraz je u originalu zapisan

```
(LABEL, SUBST, (LAMBDA, (X, Y, Z), (COND,
  ((ATOM, Z), (COND, (EQ,Y,Z),X),((QUOTE,
  T),Z))),((QUOTE,T),(CONS,(SUBST,X,Y,
 (CAR,Z)),(SUBST,X,Y,(CDR,Z)))))).
```

9.19. PRIMJEDBE O PREVOĐENJU M-IZRAZA

Čini se smislenim uvesti pojam "simboličkih izraza pridruženih funkcijama" o kojima McCarthy nije pisao, ali koji su prirodna i korisna analogija meta-izraza pridruženih funkcijama, te će biti potrebni u poglavlju o S-funkcijama i teoriji izračunljivosti. *S-izraz pridružen funkciji h*, oznaka $h^{(S)}$ bi bio prijevod meta-izraza pridruženog funkciji *h*. Primjerice,

$$list1 = \lambda[[x]; cons[x; \texttt{NIL}]]$$

$$list1^{(S)} = \texttt{(LAMBDA,(X),(CONS,X,(QUOTE,NIL)))}.$$

Nadalje, iako McCarthy o tome nije pisao, ne mogu se svi meta-izrazi prevesti u simboličke izraze na opisani način. Široka klasa meta-izraza kojima se McCarthy nije služio pri definiranju S-funkcija, poput

$$[[\exists y][cons[y; y] = x] \rightarrow \texttt{T}, \texttt{T} \rightarrow \texttt{F}]$$

159 McCarthy, *Recursive functions* ..., CACM, 1960., str. 189.

nisu prevodivi, jer nisu dane upute za njihovo prevođenje. Ta nemogućnost nije rezultat ograničenosti ideje prevođenja meta-izraza u S-izraze, nego samo toga što je McCarthy bio zainteresiran isključivo za prevođenje meta-izraza kojima se definiraju S-funkcije. Ustreba li u nekom konkretnom programu prevođenje nekih drugih meta-izraza, moguće je uvesti i nova pravila prevođenja.

Čak niti meta-izrazi kojima su funkcije obično definirane, oblika

$$f[x_1; ...; x_n] = e.$$

se ne mogu prevesti, jer ne postoji pravilo za prevođenje jednakosti. No, meta-izraz s desne strane jednakosti, *e*, obično je prevodiv. Prevodivi su i meta izrazi oblika

$$\lambda[[x_1; ...; x_n]; e]$$

$$label[f, \lambda[[x_1; ...; x_n]; e]].$$

za konkretne varijable x_1, ..., x_n i *e*. Značajna iznimka je *list* čija definicija

$$list[e_1; ...; e_n] = cons[e_1; cons[e_2; ...; cons[e_n; \texttt{NIL}]...]].$$

nije prevodiva jer desna strana jednakosti, pa i pripadajući label-izraz sadrže tri točkice koje se ne mogu ukloniti a za koje, isto tako, nije dana uputa o prevođenju.

Meta-izrazi koji sadrže iskazne veznike nisu prevodivi. Zamijene li se iskazni veznici ekvivalentim uvjetnim izrazima, dobiveni meta-izrazi su prevodivi.

Valja uočiti da se pravila za prevođenje meta izraza mogu proširiti, i tako omogućiti prevođenje šire klase meta-izraza u S-izraze. Doista, u kasnijem razvoju Lispa, to se i činilo. Primjerice, definicije oblika $f[x_1; ...; x_n] = e$ su prevođene s `(DEFINE,`f`*,(`x_1`*,...,`x_n`*),`e`*)`.

Mogli bi se upitati je li moguće sve meta-izraze (dakle, sve matematičke izraze) uopće prevesti nekim skupom pravila u simboličke izraze. Čini se da jest; autoru knjige nije poznat niti jedan matematički izraz koji ne bi bio prevodiv u simboličke izraze.

Neatomarni S-izrazi koji nisu "pokrativi" u oblik liste ili ugnježđene liste, primjerice, `(A·B)`, nisu prijevod nekog meta-izraza. Iznimka su S-izrazi u kojima se nepokrativi dijelovi nalaze unutar liste oblika `(QUOTE,`*e*`)`. Primjerice, `(QUOTE,(A·B))` je prijevod meta-izraza `(A·B)`.

9.20. MATEMATIČKA DEFINICIJA S-FUNKCIJE EVAL

Neki meta-izrazi imaju vrijednost. Primjerice, *cons*[`A`; `(B)`] ima vrijednost `(A,B)`. Meta-izraz *f*[*x*; `(B)`] ima vrijednost samo ako *f* i *x* imaju vrijednost. Implementirani Lisp sistem bi trebao izračunavati vrijednost meta-izraza umjesto čovjeka.

McCarthy definira S-funkciju *eval* koja, doduše, ne izračunava vrijednosti meta-izraza, ali radi nešto vrlo slično tome: izračunava vrijednosti prevedenih M-izraza.

S-funkcija *eval* ima dva argumenta:

1. simbolički izraz dobiven prevođenjem M-izraza i

2. listu parova varijabli i imena funkcija i njihovih vrijednosti. Ta lista se u Lisp literaturi naziva *lista asocijacija*.[160]

Neka je, primjerice, *x* = `(A)` i *y* = `(B)`. Tada se izračunavanje *cons*[*x*; *y*] može svesti na primjenu funkcije *eval*.

$$cons[x; y] = eval[\texttt{(CONS,X,Y)};\texttt{((X,(A)),(Y,(B)))}].$$

Izračunavanje vrijednosti meta-izraza *append*[*x*; *y*] moguće je ako se u listu parova doda i definicija ne-elementarne S-funkcije *append*. Ta S-funkcija "ima M-izraz"

$$label[append; \lambda[x; y][null[x] \rightarrow y; \texttt{T} \rightarrow cons[car[x]; append[cdr[x]; y]]]].$$

u kojem se koristi S-funkcija *null* "čiji M-izraz" je, eliminira li se veznik $\wedge$,

160 U originalnom RFSE, termin "lista asocijacija" se koristi za ono što se inače, uključujući i ovu knjigu, naziva "listom svojstava".

$$\lambda[[x]; [atom[x] \rightarrow eq[x; \texttt{NIL}]; \texttt{T} \rightarrow \texttt{F}]].$$

Tada

append[*x*; *y*] = *eval*[(APPEND,X,Y);
((X,(A)),
 (Y,(B)),
 (APPEND,
   (LABEL,APPEND,
     (LAMBDA,(X,Y),
        (COND,((NULL,X),Y),
              ((QUOTE,T),
               (CONS,(CAR,X),
                     (APPEND,(CDR,X),Y))))))),
   (NULL,(LAMBDA,(X),
            (COND,((ATOM,X),(EQ,X,NIL)),
                  ((QUOTE,T),(QUOTE,F))))))].

Općenito, neka je *e* prevodivi meta-izraz, x_1, ..., x_n su "slobodne varijable" čije vrijednosti su S-izrazi s_1, ..., s_n, a f_1, ..., f_k su imena ne-elementarnih funkcija koje imaju meta-izraze m_1, ..., m_k. Ako se na temelju navedenih informacija može izračunati vrijednost *e* tada je ta vrijednost jednaka

$$eval[e^*; ((x_1^*, s_1), \ldots, (x_n^*, s_n), (f_1^*, m_1^*), \ldots, (f_n^*, m_k^*))].$$

Vrijednost *eval*[*e*; *a*] se definira ovisno o obliku *e*.

1. Simboli. S-funkcija *eval* se svodi na *assoc*. Primjerice,

   $$eval[\texttt{X}; \texttt{((X,B))}] = assoc[\texttt{X}; \texttt{((X,B))}] = \texttt{B}.$$

   Općenito, ako je *e* simbol, onda

   $$eval[e; a] = assoc[e; a].$$

2. Quote-izrazi. S-funkcije *eval* i *quote* poništavaju. Primjerice,

   $$eval[\texttt{(QUOTE,Y)}; \texttt{((Y,B))}] = \texttt{(Y)}.$$

Općenito, ako e = (QUOTE,e_0) tada

$$eval[e; a] = eval[(\texttt{QUOTE},e_0); a] = e_0.$$

3. Atom-izrazi. *Eval* se reducira na S-funkciju *atom*. Primjerice,

$$eval[(\texttt{ATOM},\texttt{X}); ((\texttt{X},\texttt{A}))] = atom[eval[\texttt{X}; ((\texttt{X},\texttt{A}))]].$$

Općenito, ako e = (ATOM,e_0) onda

$$eval[e; a] = eval[(\texttt{ATOM},e_0); a] = atom[eval[e_0; a]].$$

4. Eq-izrazi. Primjerice,

$$eval[(\texttt{EQ},\texttt{X},(\texttt{QUOTE},\texttt{A}));a] = eq[eval[\texttt{X}; a]; eval[(\texttt{QUOTE},\texttt{A}); a]].$$

Općenito, ako e = (EQ,e_1,e_2) onda

$$eval[e; a] = eval[(\texttt{EQ},e_1,e_2); a] = eq[eval[e_1; a];eval[e_2; a]].$$

5. Cond-izrazi. Primjerice,

$$eval[(\texttt{COND},((\texttt{CAR},\texttt{A}),\texttt{A}),((\texttt{CDR},\texttt{B}),\texttt{B})); a] = \\ = [eval[(\texttt{CAR},\texttt{A}); a] \rightarrow eval[\texttt{A}; a]; eval[(\texttt{CDR},\texttt{B}); a] \rightarrow eval[\texttt{B}; a]].$$

Općenito, ako e = (COND,(p_1,e_1),...,(p_n,e_n)) onda

$$eval[e; a] = eval[(\texttt{COND},(p_1,e_1),\ldots,(p_n,e_n));a] = \\ = [eval[p_1; a] \rightarrow \text{eval}[e_1; a]; \ldots; eval[p_n; a] \rightarrow eval[e_n; a]].$$

6. Car-izrazi. Primjerice,

$$eval[(\texttt{CAR},(\texttt{QUOTE},(\texttt{A},\texttt{B})));a] = car[eval[(\texttt{QUOTE},(\texttt{A},\texttt{B}));a]$$

Općenito, ako e = (CAR,e_0) onda

$$eval[e; a] = eval[(\texttt{CAR},e_0); a] = car[eval[e_0; a]].$$

7. Cdr-izrazi. Primjerice,

$$eval[\texttt{(CDR,(QUOTE,(A,B)))}; a] = cdr[eval[\texttt{(QUOTE,(A,B))}; a].$$

Općenito, ako $e = \texttt{(CDR,}e_0\texttt{)}$ onda

$$eval[e; a] = eval[\texttt{(CDR,}e_0\texttt{)}; a] = cdr[eval[e_0; a]].$$

8. Cons-izrazi. Primjerice,

$$eval[\texttt{(CONS,(QUOTE,(A,B)),(QUOTE,(C,D)))}; a] =$$
$$= cons[eval[\texttt{(QUOTE,(A,B))}; a]; eval[\texttt{(QUOTE,(C,D))}; a]].$$

Općenito, ako $e = \texttt{(CONS,}e_1\texttt{,}e_2\texttt{)}$ onda

$$eval[e;a] = eval[\texttt{(CONS,}e_1\texttt{,}e_2\texttt{)}; a] =$$
$$cons[eval[e_1; a];eval[e_2; a]].$$

9. S-izrazi čiji je *car* lambda-izraz. Primjerice,

$$eval[\texttt{((LAMBDA,(X),(APPEND,X,X)),(QUOTE,(Y)))};\texttt{()}] =$$
$$= eval[\texttt{(APPEND,X,X)}; \texttt{((X,(Y)))}].$$

Općenito, ako $e = \texttt{((LAMBDA,(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{),}e_0\texttt{),}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}$ onda

$$eval[e; a] = eval[\texttt{((LAMBDA,(}x_1\texttt{,}\ldots\texttt{,}x_n\texttt{),}e_0\texttt{),}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}; a] =$$
$$= eval[e_0; append[\texttt{((}x_1\texttt{,}eval[e_1; a]\texttt{),}\ldots\texttt{,(}x_n\texttt{,}eval[e_n; a]\texttt{))}; a]].$$

10. S-izrazi čiji je *car* label-izraz. Primjerice,

$$eval[\texttt{((LABEL,F,(LAMBDA,(X),(F,X))),Y)}; \texttt{()}]=$$

$$= eval[\texttt{((LAMBDA,(X),(F,X)),Y)};\texttt{((F,(LAMB-DA,(X),(F,X))))}].$$

Općenito, ako $e = \texttt{((LABEL,}f\texttt{,}l\texttt{),}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}$ onda

$$eval[e; a] = eval[\texttt{((LABEL,}f\texttt{,}l\texttt{),}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}; a] =$$
$$= eval[\texttt{(}l\texttt{,}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}; append[\texttt{((}f\texttt{,}l\texttt{))}; a]].$$

11. S-izrazi čiji je *car* simbol. Primjerice,

$$eval[\texttt{(F,A)};\texttt{((F,(LAMBDA,(X),X)))}] =$$

$$= eval[\texttt{((LAMBDA,(X),X),A)};\texttt{((F,(LAMBDA,(X),X)))}].$$

Općenito, ako $e = \texttt{(}s\texttt{,}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}$, gdje je s simbol, onda

$$eval[e; a] = eval[\texttt{(}s\texttt{,}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}; a] =$$
$$eval[\texttt{(}assoc[s; a]\texttt{,}e_1\texttt{,}\ldots\texttt{,}e_n\texttt{)}; a].$$

9.21. S-FUNKCIJA EVAL

Prethodni je opis dovoljan za matematički korektnu definiciju funkcije *eval*. No, McCarthy je definirao *eval* jednako kao i većinu drugih ne-elementarnih S-funkcija, meta-izrazom koji uvjetnim izrazom objedinjava sve različite slučajeve. Definicija funkcije *eval* u članku, pa i u svim internim dokumentima koji su prethodili sadrži očigledne, ali nebitne pogreške o kojima su pisali Jordan[161], Stoyan[162] i Graham[163]. Pogreške su ispravljene u definiciji

$$\begin{aligned}
&eval[e; a] = \\
&[atom[e] \rightarrow assoc[e; a]; \\
&\ atom[car[e]] \rightarrow [eq[car[e]; \texttt{QUOTE}] \rightarrow cadr[e]; \\
&\qquad\qquad eq[car[e]; \texttt{ATOM}] \rightarrow atom\ [eval[cadr[e]; a]]; \\
&\qquad\qquad eq[car[e]; \texttt{EQ}] \rightarrow eq[eval[cadr[e]; a]; \\
&\qquad\qquad\qquad\qquad\qquad eval[caddr[e]; a]]; \\
&\qquad\qquad eq[car[e]; \texttt{COND}] \rightarrow evcon[cdr[e]; a]; \\
&\qquad\qquad eq[car[e]; \texttt{CAR}] \rightarrow car[eval[cadr[e]; a]]; \\
&\qquad\qquad eq[car[e]; \texttt{CDR}] \rightarrow cdr[eval[cadr[e]; a]]; \\
&\qquad\qquad eq[car[e]; \texttt{CONS}] \rightarrow cons[eval[cadr[e]; a]; \\
&\qquad\qquad\qquad\qquad\qquad eval[caddr[e]; a]]; \\
&\qquad\qquad \texttt{T} \rightarrow eval[cons[assoc[car[e]; a]; cdr[e]^{164}];
\end{aligned}$$

161 Jordan, *A note on LISP universal S-functions*, 1973.

162 Stoyan, *The influence of the designer on the design — McCarty and Lisp*, 1991.

163 Graham, *Roots of Lisp*, 2002.

164 U originalnom članku, ali i u Lisp I. priručniku ovdje se nalazi, po mišljenju svih komentatora, logička pogreška: umjesto *cdr*[*e*] navedeno je *evlis*[*cdr*[*e*];*a*].

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad\qquad\qquad a]];\\
&eq[caar[e];\ \texttt{LABEL}] \rightarrow\ eval[cons[caddar[e];cdr[e]];\\
&\qquad\qquad\qquad\qquad\qquad\quad cons[list[cadar[e];car[e]];\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad a]]];\\
&eq[caar[e];\ \texttt{LAMBDA}] \rightarrow\\
&\quad eval[caddar[e];append[pair[cadar[e];evlis[cdr[e];\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad a]];\\
&\qquad\qquad\qquad\qquad\qquad\quad a]]].
\end{aligned}
$$

Koriste se "pokrate" *cadr*, *caddr*, *caddar*, ... kao i *list*. Pomoćne S-funkcije *evcon* koja služi za izračunavanje cond-izraza i *evlis* koja primjenjuje *eval* "unutar liste" definirane su s

$$
\begin{aligned}
evcon[c;\ a] = [&eval[caar[c];\ a] \rightarrow eval[cadar[c];\ a];\\
&\texttt{T} \rightarrow evcon[cdr[c];\ a]]
\end{aligned}
$$

$$
\begin{aligned}
evlis[m;\ a] = [&null[m] \rightarrow \texttt{NIL};\\
&\texttt{T} \rightarrow cons[eval[car[m];\ a];\\
&\qquad evlis[cdr[m];\ a]]].
\end{aligned}
$$

Funkcija *eval* primjenjuje "strategiju izračunavanja" danas poznatu pod imenom "call by value" za sve pozive funkcija. Argumenti funkcije se izračunavaju prije nego što se funkcija primijeni. Quote-izrazi i cond-izrazi se izračunavaju bez prethodnog izračunavanja argumenata.

9.22. PRIMJEDBE UZ DEFINICIJU FUNKCIJE EVAL

Lisp definiran unutar funkcije *eval* različit je od Lispa kojim je *eval* definiran. Nije riječ samo o sintaksi, simboličkim izrazima umjesto originalnih meta-izraza. "Unutrašnji" Lisp ima operator QUOTE, koji se ne upotrebljava pri definiciji S-funkcija. "Vanjski" Lisp prihvaća niz definicija funkcija u implicitnom ili eksplicitnom obliku, a onda omogućuje izračunavanje izraza koji koriste te funkcije. "Unutrašnji" Lisp traži da se sve prethodno definirane funkcije dodaju u drugi argument, u obliku

$$(f^*,(\text{LAMBDA},(x_1^*,\ldots,x_n^*),e^*))$$

gdje su *f* ime funkcije, x_1, …., x_n simboli, *e* izraz. Pri tome, ako je funkcija *f* rekurzivna, nije potrebno koristiti label--izraze. Primjerice, poziv funkcije

```
eval[(APPEND,X,Y);
     ((X,(A)),
     (Y,(B)),
     (APPEND,(LABEL,APPEND,(LAMBDA,(X,Y),…))),
     (NULL,(LAMBDA,(X),…)))      ],
```

se, pogleda li se kako je definirano izračunavanje label-izraza, izračunava malo složenije, ali ima istu vrijednost kao i

```
eval[(APPEND,X,Y);
     ((X,(A)),
      (Y,(B)),
      (APPEND,(LAMBDA,(X,Y),…))),
      (NULL,(LAMBDA,(X),…)))      ].
```

Postoje neke razlike između McCarthyjevog *evala* i onog u većini suvremenih implementacija jezika. Programer vjerojatno ne očekuje da

$$\begin{aligned} &eval[(\text{P},(\text{QUOTE},(\text{A},\text{B}))); ((\text{P},\text{Q}),(\text{Q},\text{CAR}))] = \\ &= eval[(\text{Q},(\text{QUOTE},(\text{A},\text{B}))); ((\text{P},\text{Q}),(\text{Q},\text{CAR}))] = \\ &= eval[(\text{CAR},(\text{QUOTE},(\text{A},\text{B}))); ((\text{P},\text{Q}),(\text{Q},\text{CAR}))] = \\ &= car[(\text{QUOTE},(\text{A},\text{B}))] = \\ &= \text{A}. \end{aligned}$$

U McCarthyjevoj implementaciji Lispa izračunavanje `(P,(QUOTE,(A,B)))` bi zacijelo završilo prijavom greške poput "`Q` nije operator." Ipak, ne može se tvrditi da je to greška u izvornom *evalu*. Ako se o M-izrazima razmišlja kao o posebnoj vrsti matematičkih izraza, onda iz $p = q$ i $q = car$ slijedi $p[x] = car[x]$.

McCarthyjev *eval* ne procesira korektno izraze u kojima je argument funkcije lambda-izraz ili label-izraz. Primjerice,

$$\begin{aligned} &eval[\texttt{(ATOM,(LAMBDA,(X),X))}; a] = \\ &= atom[eval[\texttt{(LAMBDA,(X),X)}; a]] = \\ &= eval[cons[assoc[\texttt{LAMBDA}; a]; \texttt{((X),X)}]; a]]. \end{aligned}$$

Prilikom izračunavanja $assoc[\texttt{LAMBDA}; a]$ došlo bi do greške. McCarthy je lambda- i label-izraze na ovom mjestu jednostavno zaboravio. Tom se problemu vratio tek nakon tri godine.[165]

Druga moguća, a po mnogima bitna pogreška u definiciji *eval* je ignoriranje mogućnosti da se parametri lambda-izraza koriste u listama asocijacija, između ostaloga odgovorno za "funarg problem".[166]

Često se navodi da *eval* definira, "formalnu"[167] ili "operacionu"[168] semantiku Lispa.

Mnogim čitaocima, susret s funkcijom *eval* predstavlja vrlo ugodno iznenađenje. Oduševljenje evalom izrazili su, primjerice, autor Smalltalka Alan Kay[169], autor Eiffela

165 Vidi poglavlje *Nova funkcija eval*.

166 Vidi poglavlje *Lisp 1.5*.

167 Gilmore, *An abstract computer with a Lisp-like machine language* …, 1963., str. 72.

168 Sebesta, *Concepts of Programming Languages*, 2012., str. 680.

169 "Yes, that was the big revelation to me when (…) I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwell's Equations of Software!" This is the whole world of programming in a few lines that I can put my hand over."
Kay, *A conversation with Alan Kay*, 2004., str. 26.

Bertrand Meyer[170] i Paul Graham.[171] Ipak, nije lako objasniti zašto *eval* ostavlja takav dojam. Vrlo ugledan Edsger W. Dijkstra, odricao je vrijednost tako definiranom interpreteru.[172]

9.23. UNIVERZALNA S-FUNKCIJA APPLY

Po uzoru na univerzalan Turingov stroj, McCarthy je definirao funkciju *apply* koja može zamijeniti svaku drugu S-funkciju, ako joj se kod te druge funkcije da kao argument. Primjerice,

$$cons[\texttt{A}; \texttt{(B,C)}] = apply[\texttt{CONS}; \texttt{(A,(B,C))}].$$

$$\lambda[[x; y]; cons[y; x]][\texttt{(A)}; \texttt{B}] =$$
$$apply[\ \texttt{(LAMBDA,(X,Y), (CONS,Y,X))}; \texttt{((A),B)}].$$

Općenito, neka je *f* ime S-funkcije koja "ima meta-izraz" *m* bez pokrata, iskaznih veznika i poziva ne-elementarnih funkcija. Tada za sve S-izraze e_1, ..., e_n vrijedi

170 "... an unbelievable feat, especially considering that the program takes hardly more than half a page — an interpreter for the language being defined, written in that very language! The more recent reader can only experience here the kind of visceral, poignant and inextinguishable jealously that overwhelms us the first time we realize that we will never be able to attend the première of Don Giovanni ..."
Meyer, *John McCarthy*, 2011.

171 "So by understanding eval, you're understanding what will probably be the main model of the computation well into the future."
Graham, *Roots of Lisp*, 2002., str. 11.

172 "In the early sixties we have suffered from a piece of computer science folklore, viz. that the crucial test whether on had designed "a good language" was, whether one could write its own interpreter in itself. I have never understood the reason for that criterion --it was suggested that such a language definition would solve the problem of the definition of (particularly) the semantics of the language--, the origin of the criterion was the example of LISP, which was "defined" by a ten-line interpreter written in LISP, and that, somehow, set the standard. … This unfortunate LISP-interpreter formulated in LISP has somehow given mechanistic (or "operational") language definitions an undeserved aura of respectability …"
Dijkstra, *Trip report, Edinburgh and Newcastle,* 1974., str. 0.

$$f[e_1; ...; e_n] = apply[m^*; (e_1, ..., e_n)].$$

S-funkcija *apply* može se definirati uz pomoć S-funkcije *eval*. Primjerice,

$$apply[\texttt{CONS}; \texttt{(A,(B,C))}] =$$
$$eval[\texttt{(CONS,(QUOTE,A), (QUOTE,(B,C)))}; \texttt{()}].$$

Općenito,

$$apply[f; args] = apply[f; (args_1, ..., args_n)] =$$
$$= eval[(f, (\texttt{QUOTE}, args_1), ..., (\texttt{QUOTE}, args_n)); \texttt{()}] =$$
$$= eval[cons[f; appq[args]]; \texttt{()}]$$

gdje

$$appq[l] = [null[l] \rightarrow \texttt{NIL};$$
$$\texttt{T} \rightarrow cons[list[\texttt{QUOTE}; car[l]];$$
$$appq[cdr[l]]]].$$

S-funkcija *apply* ne dozvoljava korištenje "okoline" onako kao što to dozvoljava *eval*. Ako je funkcija *f* definirana meta-izrazom *m* u kojem se koriste imena prethodno definiranih, neelementarnih S-funkcija, *apply* neće raditi. Zato je potrebno eliminirati pozive pomoćnih funkcija.

9.24. INTERPRETER ZA LISP

Programer Russell uočio je da implementacija *evala* u mašinskom jeziku daje jednostavan i upotrebljiv interpreter za Lisp. Ta je ideja iznenadila i samog McCarthyja koji nije vjerovao u njenu praktičnu provedivost.[173,174] Mc-

173 "... This EVAL was written and published on the paper and Steve Russell said, look, why don't I program this EVAL and you remember the interpreter, and I said to him, ho, ho, you're confusing theory with practice, this EVAL is intended for reading not for computing. But he went ahead and did it. That is, he compiled the EVAL in my paper in to 704 machine code fixing bugs and then advertised this as a LISP interpreter which it certainly was, so at that point LISP had essentially the form that it has today, the s-expression form ..."
Stoyan, *Early LISP history (1956-1959)*, 1984., str. 307.

Carthyjeva sumnja se Stoyanu činila nevjerojatnom.[175]

Interpreter za programski jezik napisan u jeziku samom obično se naziva "meta-cirkularni evaluator". Taj je naziv skovao, čini se, John C. Reynolds, 1972.[176] McCarthyjev *eval* nije prvi meta-cirkularni evaluator; prethodio mu je Böhmov[177], a još ranije univerzalni Turingov stroj.[178]

U vrijeme nastanka "čistog Lispa", McCarthy je S-funkciju *apply* vidio kao interpreter.[179, 180] Tek znatno kasnije je počeo S-funkciju *eval* nazivati interpreterom za Lisp.[181,182]

Russell je uspio implementirati *eval* što je programerima omogućilo pisanje programa u obliku S-izraza.[183]

9.25. S-funkcije i teorija izračunljivosti

McCarthy je iznio i tezu da su S-funkcije pogodna osnova ne samo za programski jezik, nego i sredstvo za razvoj teorije izračunljivosti.[184] Tri su razloga za to: jednostavno izražavanje rekurzivnih funkcija nad simboličkim izrazima kao simboličkih izraza, što čini "*umjetne*" konstrukcije poput Turingovih strojeva i *Gödelovih brojeva* nepotrebnima.[185] Drugo, jednostavno i efikasno izračunavanje zanimljivih S-funkcija uz pomoć računala. Konačno, upotreba uvjetnih izraza znatno pojednostavljuje rekurzivne definicije.

Osnovni rezultati teorije izračunljivih funkcija su, po McCarthyju,

174 Stoyan, *LISP history*, 1979., str. 45.

175 Stoyan, *Lisp 50 years ago*, 2008.

176 Reynolds, *Definitional interpreters* ..., 1972., str. 725.

177 Böhm, *Calculatrices digitales* ..., 1954.

178 Turing, *On computable numbers, with an application* ..., 1936.

179 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., str. 144.

180 McCarthy, *Recursive functions* ..., CACM, 1960., str. 193.

181 McCarthy, *History of LISP*, 1981., str. 173., 179.

182 McCarthy, *Lisp – notes on its past and future*, 1980., str. v.

183 McCarthy, *History of LISP*, 1980.

184 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., str. 145.

185 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., str. 124.

1. Turingova teza, argument da svaki efektivno izračunljivi proces predstavljiv Turingovim strojem.

2. Postojanje univerzalnog Turingovog stroja koji simulira rad svih drugih Turingovih strojeva.

3. Dokaz da ne postoji Turingov stroj koji računa da li se svaki Turingov stroj zaustavlja.

Univerzalna S-funkcija je dana u nastavku; to je S-funkcija *apply*. Umjesto tvrdnje da se svaki efektivno izračunljivi proces može predstaviti S-funkcijom, McCarthy dokazuje da se svaki Turingov stroj može simulirati S-funkcijom. Konačno, iznosi i nezavisni dokaz da ne postoji S-funkcija koja računa je li S-funkcija definirana za dani skup argumenata.

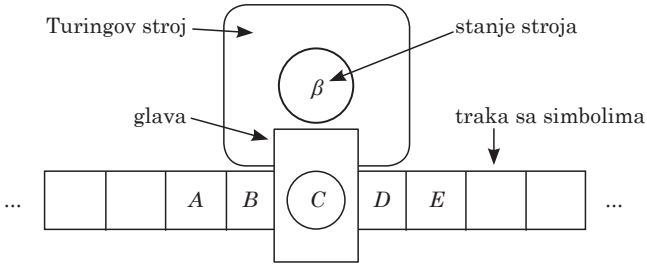
9.26. SIMULACIJA TURINGOVIH STROJEVA S-FUNKCIJOM

McCarthy definira S-funkciju turing[186, 187] koja "simulira" rad Turingovog stroja. Definicija je vrlo direktna, s nekoliko pomoćnih funkcija, a ovdje je izložena verzija iz RLE QPR 053, s manjim ispravkama i pojednostavljenjima.

Definicije Turingovih strojeva se u literaturi pojavljuju s manjim razlikama, pri čemu je McCarthyjeva definicija prilično tipična.

186 McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 9-12.

187 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., str. 145-147.



SLIKA 11. Skica Turingovog stroja. Stanje stroja je β. Vrijednost polja trake ispod glave za čitanje i pisanje je C.

Turingovi strojevi imaju dvije vrste memorije, tzv. *stanje stroja* i *traku*. Stroj se tijekom rada uvijek nalazi u jednom od konačnog broja stanja. Stanje stroja se mijenja tijekom rada u skladu s "instrukcijama" koje su također dio definicije Turingovog stroja.

Traka je jednodimenzionalna, beskonačna na obje strane i podijeljena na polja. Na svakom polju na traci može biti napisan jedan od konačno mnogo znakova. Iako je traka beskonačna, u svakom trenutku, samo na konačnom broju polja upisan je neki znak. Ostala polja su prazna. Dogovorno, prazno polje se može razumjeti kao da je na njemu napisan poseban znak, "blank". Turingov stroj ima "glavu" koja je smještena iznad jednog polja i može čitati i pisati znakove samo na polju ispod glave stroja, te se traka može pomicati lijevo ili desno po traci. Glava stroja se pomiče za jedno polje ulijevo ili udesno, u skladu s instrukcijama.

Rad Turingovog stroja određen je konačnim, nepromjenjivim nizom instrukcija oblika

"Ako
stanje stroja je x_1 i
znak ispod glave za čitanje je y_1
onda
promjeni stanje stroja u x_2,
zapiši znak y_2 na traku i
pomakni glavu u smjeru d."

Stroj završava rad ako ne postoji instrukcija koja opisuje što činiti za zadanu kombinaciju stanja stroja i znaka ispod glave stroja. Sadržaj trake nakon završetka rada je rezultat rada stroja.

S-funkcija koja simulira Turingove strojeve. Da bi pokazao da su sve funkcije izračunljive Turingovim strojevima također izračunljive S-funkcijama, McCarthy definira S-funkciju *turing*. Poziv funkcije

$$turing[machine; tape]$$

je definiran za S-izraz *machine* koji predstavlja Turingov stroj i S-izraz *tape* koji predstavlja traku Turingovog stroja. Vrijednost poziva je S-izraz koji predstavlja konfiguraciju trake u trenutku zaustavljanja, ako se Turingov stroj zaustavlja.

Predstavljanje Turingovog stroja. Turingov stroj je predstavljen simboličkim izrazom

$$(initial\text{-}state, instruction_1, \ldots, instruction_n),$$

gdje je *initial-state* simbol koji predstavlja inicijalno stanje stroja, a $instruction_i$, i=1, …, n, $n \geq 0$, su petorke koje predstavljaju instrukcije Turingovog stroja.

Predstavljanje instrukcija Turingovog stroja. Petorka $instruction_i$ oblika

$$(current\text{-}state_i, current\text{-}symbol_i, new\text{-}symbol_i, direction_i, new\text{-}state_i).$$

predstavlja instrukciju

"Ako
stanje stroja je $current\text{-}state_i$ i
$current\text{-}symbol_i$ se nalazi ispod glave
onda
zapiši $new\text{-}symbol_i$ na traku,
pomakni glavu u smjeru $direction_i$ i
promjeni stanje stroja u $new\text{-}state_i$."

Primjerice, Turingov stroj koji se postavlja na najljevije neprazno polje na traci, i računa parnost brojeva 1 na tra-

ci ignorirajući nule, dok ne naiđe na prazno polje je predstavljen S-izrazom

```
(0,(0,0,B,R,0),(0,1,B,R,1),(0,B,0,R,2),
   (1,0,B,R,1),(1,1,B,R,0),(1,B,1,R,2)).
```

Predstavljanje trake Turingovog stroja. Traka Turingovog stroja predstavljena je simboličkim izrazom oblika

(*current-symbol*,*left-part*,*right-part*),

gdje *current-symbol* predstavlja znak upisan na traci ispod trenutnog položaja glave za čitanje, *left-part* je lista simbola koja predstavljaju dio trake lijevo od glave stroja, a *right-part* je lista simbola koja predstavlja dio trake desno od glave stroja. Ostala polja na traci su prazna.

SLIKA 12. Primjer trake Turingovog stroja

Primjerice, traka Turingovog stroja na slici 12 s istaknutim mjestom nad kojim se nalazi glava za čitanje i pisanje je predstavljena simboličkim izrazom `(0,(1,0,1,1),(1,1,0))`.

Pomoćna S-funkcija *find*. Poziv S- funkcije

find[*current-state*; *current-symbol*; *instructions*],

je definiran za simbole *current-state* koja predstavlja stanje stroja, *current-symbol* koji predstavlja znak koji se nalazi ispod glave Turingovog stroja, i liste *instructions* koja predstavlja skup instrukcija. Poziv S-funkcije ima vrijednost

(*new-symbol*,*direction*,*new-state*),

trojku koja opisuje što Turingov stroj treba učiniti sljedeće. Funkcija *find* definirana je meta-izrazom

find[*current-state*; *current-symbol*; *instructions*] =
[*null*[*instructions*] → NIL;
[*first*[*first*[*instructions*]] = *current-state* ∧
second[*first*[*instructions*]] = *current-symbol*]
→ *third*[*first*[*instructions*]];
T → *find*[*current-state*;
current-symbol;
rest[*instructions*]]].

gdje su funkcije *first*, *rest*, *second*, *third* ekvivalentne funkcijama *car*, *cdr*, *cadr*, *caddr*.

Predstavljanje konfiguracije Turingovog stroja. *Konfiguracija Turingovog stroja* sastoji se od promjenjivih elemenata Turingovog stroja za vrijeme rada: stanja stroja, trake i pozicije glave na traci. Predstavljena je S-izrazom

(*current-state*,*current-symbol*,*left-part*,*right-part*),

gdje *current-state* predstavlja trenutno stanje stroja, *current-symbol, left-part* i *right-part* kao u S-izrazu koji predstavlja traku Turingovog stroja.

Pomoćna S-funkcija *successor*. Poziv funkcije

successor[*machine*; *configuration*]

definiran je za svaki S-izraz *machine* oblika

(*initial-state*,$instruction_1$,…,$instruction_n$)

koji predstavlja Turingov stroj i svaki S-izraz *configuration* koji predstavlja konfiguraciju Turingovog stroja. Vrijednost poziva funkcije je sljedeća konfiguracija koja nastaje pri radu Turingovog stroja; NIL ako takve nema, tj. ako se stroj zaustavlja. Radi preglednijeg izlaganja definicije koriste se neke pokrate koje McCarthy nije originalno koristio; *todo* je pokrata za meta-izraz

find[*first*[*configuration*]; *second*[*configuration*];
rest[*machine*]]

koji ima vrijednost (*new-symbol*,*direction*,*new-state*), trojku koja opisuje što stroj treba učiniti sljedeće;

current-tape je pokrata za meta-izraz

rest[*configuration*]

koji ima vrijednost (*current-symbol*,*left-part*,*right-part*) koji predstavlja stanje trake. S-funkcija *successor* je definirana meta-izrazom

```
successor[machine; configuration] =
    [todo = NIL → NIL;
    T→cons[third[todo];
            [second[todo] = L →
               list[first[second[current-tape]];
                    rest[second[current-tape]];
                    cons[first[todo];third[current-tape]]];
             second[todo] = R →
              list[first[third[current-tape]];
                    cons[first[todo];second[current-tape]];
                    rest[third[current-tape]]]]]].
```

Uvrste li se vrijednosti za pokrate, dobiva se "prava definicija", znatno složenija od ove.

Pomoćna S-funkcija *tu*. Poziv funkcije

tu[*machine*; *configuration*]

(*current-state*,*current-symbol*,*left-part*,*right-part*),

definiran je za simbolički izraz *machine* koji predstavlja Turingov stroj i simbolički izraz *configuration* koji predstavlja konfiguraciju Turingovog stroja, ako se odgovarajući Turingov stroj uz odgovarajuću konfiguraciju zaustavlja. Vrijednost poziva predstavlja stanje Turingovog stroja nakon zaustavljanja. S-funkcija *tu* definirana je meta-izrazom

tu[*machine*; *configuration*] =
  [*successor*[*machine*; *configuration*] = `NIL` →
    rest[*configuration*];
  `T` → *tu*[*machine*; *successor*[*machine*; *configuration*]]].

S-funkcija *turing*. Poziv funkcije

turing[*machine*; *tape*]

izračunava završno stanje Turingova stroja predstavljenog s-izrazom *machine* primijenjenog na traci predstavljenoj simboličkim izrazom *tape*. S-funkcija *turing* definirana je meta-izrazom

turing[*machine*; *tape*] =
tu[*machine*; *cons*[*first*[*machine*]; *tape*]].

McCarthy nije samo pokazao kako se pojedinačni Turingovi strojevi mogu kodirati u pojedinačne S-funkcije koje ih simuliraju, nego je definirao i S-funkciju koja simulira sve Turingove strojeve.

9.27. PITANJA O S-FUNKCIJAMA NEODLUČIVA S-FUNKCIJOM

McCarthy dokazuje, slično kao i u drugim teorijama izračunljivosti, da se neka važna pitanja o S-funkcijama ne mogu odlučiti S-funkcijama. Njegov je dokaz u nastavku neznatno pojednostavljen i malo preciznije opisan nego u originalu.

Označimo s $h^{(S)}$ prijevod meta-izraza pridruženog funkciji *h*. Primjerice,

list1 = λ[[*x*]; *cons*[*x*; `NIL`]]
list1$^{(S)}$ = `(LAMBDA,(X),(CONS,X,(QUOTE,NIL)))`.

Za S-funkciju *h* možemo reći da je *samoprimjenjiva* ako je vrijednost $h[h^{(S)}]$ definirana. Primjerice, *car* nije samoprimjenjiva; λ[[*x*]; *car*[*x*]] jest. Mogli bi se zapitati može li se samoprimjenjivost opisati S-funkcijom. Preciznije, postoji li S-funkcija *selfappliable* sa svojstvom

$$selfappliable[h^{(S)}] = \begin{cases} \texttt{T} \text{ ako } h[h^{(S)}] \text{ je definirana,} \\ \texttt{F} \text{ ako } h[h^{(S)}] \text{ nije definirana.} \end{cases}$$

Pretpostavimo da postoji. Tada se može definirati i S-funkcija *contra*

$$contra[x]=[selfappliable[x] = \texttt{T} \rightarrow car[\texttt{NIL}]; \\ selfappliable[x] = \texttt{F} \rightarrow \texttt{ANYVALUE}].$$

Funkcija *contra* radi suprotno od onoga što tvrdi *selfappliable*. Je li funkcija *contra* samoprimjenjiva, tj. je li vrijednost *contra*[*contra*$^{(S)}$] definirana? Pretpostavimo da nije definirana. Iz definicije *contra* slijedi da

$$selfappliable[contra^{(S)}] = \texttt{T}.$$

Tada, po svojstvu *selfappliable*, vrijednost *contra*[*contra*$^{(S)}$] jest definirana, što je u kontradikciji s pretpostavkom.

Pretpostavimo da *contra*[*contra*$^{(S)}$] jest definirana. Tada, po definiciji *contra*, vrijedi da

$$selfappliable[contra^{(S)}] = \texttt{F}.$$

Tada, po svojstu *selfappliable*, vrijednost *contra*[*contra*$^{(S)}$] nije definirana.

Dakle, tvrdnja da je *contra* S-funkcija vodi do kontradikcije. No, definicija *contra* je korektna, ako je *selfappliable* S-funkcija. Dakle, *selfappliable* nije S-funkcija.

Na isti način se može dokazati da općenitija funkcija,

$$def[h^{(S)}; (args_1, \ldots, args_n)]$$

koja izračunava je li S-funkcija h definirana za proizvoljne argumente $args_1$, …, $args_n$ nije S-funkcija. Kad bi *def* bila S-funkcija, tada bi se mogla definirati i *selfappliable*

$$selfappliable[h^{(S)}] = def[h^{(S)}; h^{(S)}],$$

što je korektna definicija S-funkcije. Kako *selfappliable* nije S-funkcija, to niti *def* nije S-funkcija.

McCarthy je zabilježio i kratku uputu za prevođenje programa u drugim programskim jezicima u S-funkcije. Ideja je jednostavna i nije izazvala veće diskusije.

"Konfiguracija računala" (engl. *machine configuration*) je u svakom trenutku definirana vrijednošću svih varijabli čija se vrijednost u programu definira. Te varijable i njihove vrijednosti mogu se "kombinirati" u listu oblika

`((varijabla,vrijednost),(varijabla,vrijednost),...)`.[188]

"Programski blokovi", dijelovi programa koji imaju samo jedan "ulaz" i jedan "izlaz" transformiraju konfiguraciju. Primjerice, programski blok

```
A = B + 1
```

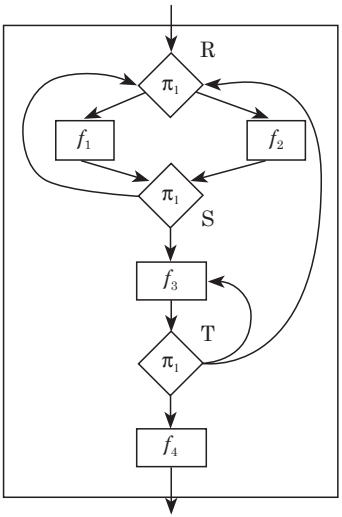
transformira konfiguraciju `((A,0),(B,1))` u `((A,2),(B,1))`. Neka su $x^{(1)}$, …, $x^{(n)}$ sve moguće "konfiguracije" računala, te neka su $y^{(1)}$, …, $y^{(n)}$ odgovarajuće konfiguracije nakon izvršavanja programskog bloka. Tada S-funkcija definirana s

$$f[x] = [x = x^{(1)} \rightarrow y^{(1)}, \ldots, x = x^{(n)} \rightarrow y^{(n)}].$$

izračunava istu konfiguraciju kao programski blok. Važno ograničenje je, pri tome, da skup mogućih konfiguracija mora biti konačan. U mnogim programima, broj konfiguracija je gotovo beskonačan i ograničen je samo memorijom računala.

Programski blok se može sastojati od dijelova koji su također programski blokovi, kao i izbora koji određuju koji će blok nastaviti izvršavanje. Takvi skupovi blokova i izbora mogu se prikazati tzv. dijagramom toka.

188 McCarthy, *Programs in LISP*, AIM-012, 1959., str. 2.



SLIKA 13. Program opisan blok dijagramom[189]

McCarthy navodi primjer prevođenja programa zadanog dijagramom toka na Slika 13. Neka su r, s i t S-funkcije koje simuliraju dio programa između točaka R, S i T i izlaza iz programa, redom. Neka su π_{11} i π_{12} mogući izbori toka programa na mjestu π_1 itd. Tada

$$r(x) = [\pi_{11}[x] \rightarrow s[f_1[x]]; \pi_{12} \rightarrow s[f_2[x]]],$$
$$s[x] = [\pi_{21}[x] \rightarrow r[x]; \pi_{21}[x] \rightarrow t[f_3[x]]],$$
$$t[x] = [\pi_{31}[x] \rightarrow f_4[x]; \pi_{32}[x] \rightarrow r[x]; \pi_{33}[x] \rightarrow t[f_3[x]]].$$

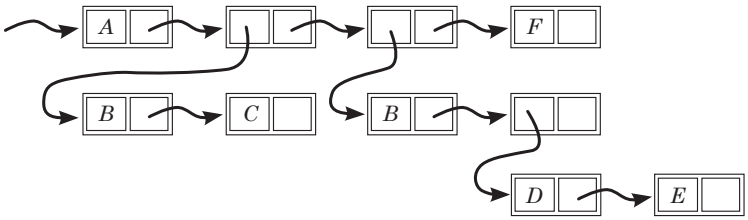
Nema bitne razlike niti ako programski blok ima više izlaza.

9.29. PREDSTAVLJANJE SIMBOLIČKIH IZRAZA U MEMORIJI RAČUNALA

Reprezentacija simboličkih izraza u memoriji računala je ista kao i u "imperativnom Lispu". Riječi su podijeljene na dva dijela, adresni i dekrementni. U svakom dijelu mogu se nalaziti adrese simbola ili drugih riječi. Skup svih tako

189 Prema McCarthy, *Recursive functions* ..., RLE QPR, 1959., str. 150.

povezanih adresa, od kojih je jedna istaknuta kao početna McCarthy naziva strukturom liste (engl. *list structure*). Pri tome, za mnoga razmatranja stvarne memorijske adrese nisu bitne; samo odnosi između adresa. Zato se strukture lista, kao i u prvom memou, mogu dobro grafički reprezentirati. U odnosu na Memo 1., grafičke reprezentacije su unaprijeđene: dodana je strelica koja pokazuje na početak liste.

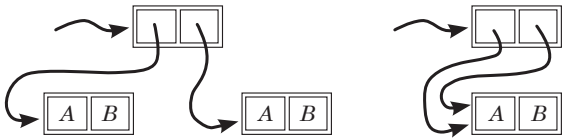


SLIKA 14. Grafička reprezentacija strukture liste koja odgovara simboličkom izrazu `(A,(B,C),(B,(D,E)),F)`.

Strukture lista mogu rasti i smanjivati se po potrebi, te omogućuju efikasno upravljanje memorijom računala.

9.30. RAZLIKE IZMEĐU SIMBOLIČKIH IZRAZA I STRUKTURA LISTA

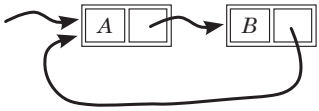
Strukture lista su općenitije od simboličkih izraza. Simbolički izrazi u kojima se isti podizraz pojavljuje dva ili više puta mogu se predstaviti kao strukture lista na više bitno različitih načina.



SLIKA 15. Dvije različite memorijske reprezentacije izraza `((A.B),(A.B))`.

S druge strane, strukture lista koje sadrže cikluse, *kružne*

strukture lista (engl. *circular list structures*) uopće nisu reprezentacije simboličkih izraza.



SLIKA 16. Primjer strukture lista koja sadrži ciklus.

Dizajner jezika je mogao definirati jezik tako da programer može procesirati sve strukture lista i prihvatiti moguće komplikacije koje proizlaze iz razlike strukture lista i simboličkih izraza; možda i iskoristiti prednosti koje proizlaze iz općenitije strukture. Alternativno, mogao je isključiti strukture lista koje ne reprezentiraju simboličke izraze, a ako simbolički izrazi imaju više reprezentacija, podržati samo jednu od tih. McCarthy je odabrao srednji put; dozvolio je različite reprezentacije istog simboličkog izraza, ali ne i strukture lista koje sadrže cikluse.[190]

9.31. GARBAGE COLLECTION

Upravljanje slobodnom memorijom uz pomoć "liste slobodnog prostora" uvedeno je u FLPL-u i prihvaćeno u "imperativnom Lispu". McCarthy je ubrzo uočio nezgrapnost eksplicitnog oslobađanje zauzete memorije i izrazio namjeru za automatiziranjem oslobađanja[191]. Rezultat McCarthyjevog intenzivnog rada[192] na problemu je program u strojnom jeziku, dio Lisp sistema interno nazivan "garba-

190 "The prohibition against circular list structures is essentially a prohibition against an expression being a subexpression of itself. Such an expression could not exist on paper in a world with our topology. Circular list structures would have some advantages in the machine, for example, for representing recursive functions, but difficulties in printing them, and in certain other operations, make it seem advisable not to use them for the present."
McCarthy, *Recursive functions ...*, CACM, 1960., str. 192.

191 Vidi poglavlja *FLPL*, *Imperativni Lisp* i *Elementi funkcionalnog programiranja.*

192 McCarthy, *Guy Steele interviews John McCarthy, father of Lisp*, 2009.

ge collection", imenom koje se McCarthyju nije činilo dovoljno ozbiljnim za znanstveni članak.[193]

"Garbage collector" se aktivira tek kad funkcija *cons* pokuša iskoristiti riječ iz liste slobodne memorije i pokaže se da je lista slobodne memorije prazna. Tada "garbage collector", polazeći od nekoliko osnovnih riječi uzastopnom primjenom funkcija *car* i *cdr* prolazi kroz cijelu dostupnu memoriju i označava sve riječi do kojih je došao postavljanjem vrijednosti S (sign) bita na 1. Naiđe li pri prolasku na riječ u kojoj je S bit već postavljen na 1, program pretpostavlja da je ta riječ, i sve riječi kojima se od te riječi može pristupiti procesirana.

U drugom prolasku kroz memoriju, ovaj put cijelu, "garbage collector" sve one riječi u kojima S bit ima vrijednost 0 organizira u listu slobodne memorije.

Konačno, program treći puta prolazi kroz memoriju, krenuvši samo od osnovnih riječi i postavlja vrijednost bita S na 0.

Opisana je tehnika danas poznata pod imenom "mark--sweep". McCarthy je razmišljao i o drugoj poznatoj, "reference counting" tehnici koja je i primijenjena u nekim implementacijama Lispa, ali ne pod McCarthyjevim vodstvom.[194]

Članovi projekta nisu žurili s implementacijom "garbage collectora" jer su prvi programi bili tek isprobavanje sistema i nisu morali biti stvarno efikasni.[195] Iako je u vrijeme razvoja "garbage collectora" memorija s kojom su programi raspolagali bila vrlo mala (lista slobodnog prostora je inicijalno imala oko 15 000 riječi), program je u praksi bio sporiji od McCarthjyeve procjene (nekoliko sekunda) i postao izvorom anegdota.[196]

193 McCarthy, *Recursive functions* ..., 1995., str. 27., bilješka 7.

194 McCarthy, *Guy Steele interviews John McCarthy, father of Lisp*, 2009.

195 McCarthy, *History of LISP*, 1981., str. 178.

196 McCarthy, *History of LISP*, 1981., str. 183.

10.

Linearni Lisp

U Memou 8.[197] i kasnijim dokumentima[198,199] McCarthy navodi da postoji više mogućih načina definiranja funkcija, "alternativnih formalizama" na simboličkim izrazima sličnih "prihvaćenom sistemu" i ukratko opisuje dvije "varijante Lispa": – "*linearni*" i "*binarni Lisp*". "Linearni Lisp" je opisan u svim internim i objavljenim verzijama članka.

10.1. L-IZRAZI

Simbolički izrazi, osnovna vrsta podataka u "čistom Lispu" namijenjeni su procesiranju matematičkih i logičkih izraza. Razlika između simboličkih izraza i općih matematičkih izraza nije mala. Primjerice, jednakost zvana "razlika kvadrata",

$$a^2 - b^2 = (a + b) \cdot (a - b)$$

u većini programskih jezika bi bila zapisana

```
a^2 - b^2 = (a + b)*(a - b).
```

Ekvivalentan simbolički izraz ima oblik

```
(= (- (^ A 2) (^ B 2)) (* (+ A B) (- A B))).
```

Mogli bi se zapitati zašto Lisp ne procesira proizvoljne nizove znakova. Osim što bi time bio riješen stalan prigovor na račun Lispa, nečitljivost S-izraza, jezik bi dobio na općenitosti. McCarthy je poglavlje u članku[200] posvetio diskusiji upravo takvog, "linearnog Lispa" koji procesira proizvoljne nizove znakova, stringove, ili kako ih još naziva, *L-izraze*, definirane s:

1. Svaki pojedinačni znak je *L-izraz*.

197 McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 16.
198 McCarthy, *Recursive functions* ..., RLE QPR 053, 1959., str. 148.
199 McCarthy, *Recursive functions* ..., CAMC, 1960., str. 194.
200 McCarthy, *Recursive functions* ..., CAMC, 1960., str. 194.

2. Bilo koji niz znakova, uključujući i prazan niz (oznaka: Λ) je *L-izraz*.

10.2. ELEMENTARNE FUNKCIJE

Definirano je šest elementarnih funkcijama L-izrazima; tri od njih su predikati.

1. Elementarna funkcija *first*. Primjerice,

$$first[\texttt{AB}] = \texttt{A}.$$

Općenito, *first*[*x*] je prvi znak u stringu *x*. Posebno, *first*[Λ] nije definiran.

2. Elementarna funkcija *rest*. Primjerice,

$$rest[\texttt{ABC}] = \texttt{BC}.$$

Općenito, *rest*[*x*] je niz znakova koji ostaje nakon brisanja prvog znaka. Posebno, *rest*[Λ] nije definiran.

3. Elementarna funkcija *combine*. Primjerice,

$$combine[\texttt{A}; \texttt{BC}] = \texttt{ABC}.$$

Općenito, *combine*[*x*; *y*] je niz znakova formiran spajanjem L-izraza *x* i *y*.

4. Elementarna funkcija *char*. Primjerice,

$$char[\texttt{A}] = \texttt{T},$$

$$char[\texttt{AA}] = \texttt{F}.$$

Općenito, ako je *x* niz od samo jednog znaka onda *char*[*x*] ima vrijednost T. Ako je *x* niz od više od jednog znaka, onda *char*[*x*] ima vrijednost F.

5. Elementarna funkcija *null*. Primjerice,

$$null[\Lambda] = \mathtt{T},$$

$$null[\mathtt{A}] = \mathtt{F}.$$

Općenito, vrijednost *null*[*x*] je T ako i samo ako je $x = \Lambda$. Inače, vrijednost *null*[*x*] je F.

6. Elementarna funkcija =. Primjerice,

vrijednost A = A je T,

vrijednost A = B je F.

Općenito, ako su *x* i *y* isti niz znakova onda $x = y$ ima vrijednost T. Ako su *x* i *y* različiti, onda $x = y$ ima vrijednost F.

10.3. IZDVAJANJE PODIZRAZA

"Linearni Lisp" je regularniji i općenitiji od "čistog Lispa". U "čistom Lispu" zagrade i zarezi imaju posebnu ulogu. U "linearnom Lispu" takvih, posebnih znakova nema. Svaki S-izraz je L-izraz, ali mnogi L-izrazi nisu S-izrazi.

S druge strane, izdvajanje podizraza u "linearnom Lispu" je složena operacija. Programi bi morali imati posebna pravila za izdvajanje izraza, ovisno o korištenim operatorima. McCarthy je vjerovao da bi većina programera u linearnom Lispu implementirala podršku za simboličke izraze i koristila "linearni Lisp" kao obični Lisp, da izbjegnu implementiranje posebnih funkcija za izdvajanje podizraza. Tada "linearni Lisp" ne bi imao prednosti pred "čistim Lispom", a zbog veće općenitosti, bio bi sporiji.

11.

Binarni Lisp

Drugu "varijantu LISPa", "binarni Lisp", McCarthy je opisao samo u Memou 8[201], ali ne i u ostalim dokumentima. Opis "binarnog Lispa" je vrlo kratak.

Asimetrična definicija funkcija *car* i *cdr*, po McCarthyju, može biti "izvor nelagode". Stoga "binarni Lisp" dozvoljava samo liste koje se sastoje od točno dva elementa. Na takvim se listama definiraju funkcije first, rest i combine. Primjerice,

$$first[(X,Y)] = X$$

$$rest[(X,Y)] = Y$$

$$combine[X; Y] = (X,Y).$$

Općenitije, za sve S-izraze e1 i e2 vrijedi

$$first[(e_1,e_2)] = e_1$$

$$rest[(e_1,e_2)] = e_2$$

$$combine[e_1; e_2] = (e_1,e_2).$$

Dovoljno je definirati dva predikata; jednakost simbola, *eq* i *atom*. U ovakvom Lispu, prazne liste ne bi bile potrebne. McCarthyjeva zamjerka sistemu je složenost reprezentacije funkcija.

Iako Memo 8. nosi isti naslov kao kasniji dokumenti, u tom memou opisani simbolički izrazi su bitno drugačiji: ne postoje "točkasti parovi".[202] S druge strane, funkcije *car*, *cdr* i *cons* u "čistom Lispu" su identične funkcijama "binarnog Lispa", ali su definirane na točkastim parovima. Stoga se čini da je "binarni Lisp" važna karika u razvoju "čistog Lispa".

201 McCarthy, *Recursive functions* ..., AIM-008, 1959., str. 17.

202 vidi raspravu u poglavlju *Čisti Lisp*.

12.

Slagleov jezik za manipulaciju simbola

Slagle je u radnji iz 1961. ukratko opisao i "symbol manipulation language" "sličan McCarthyjevom Lispu".[203]

Osnovno svojstvo Slagleovog jezika je odustajanje od koncepta "točkastih parova". Dok McCarthy uvodi simboličke izraze kao parove oblika $(e_1 . e_2)$, a $(e_1, \ldots, e_n)$ tek kao pokratu za $(e_1 . (e_2 . (\ldots (e_n . \texttt{NIL}) \ldots)))$, Slagle uvodi liste direktno. Za njega, $(e_1, \ldots, e_n)$ je pravi simbolički izraz, a ne pokrata, dok "točkasti parovi" uopće nisu simbolički izrazi. Funkcija *cons*[*x*; *y*] je definirana samo ako je *y* ne-atomarni S-izraz ili `NIL`.

Umjesto `T` i `F`, Slagle koristi `TRUE` i `FALSE`. Umjesto *car* i *cdr*, piše kratko *f* (od engl. *first*) i *r* (od engl. *rest*.) Umjesto *caar*, *cadr*, *cdar*, *cddr* itd. piše *ff*, *fr*, *rf*, *rr* itd.

Značajna je inovacija i operacija kompozicije funkcija, primjerice, $f \circ g$.

Slagle, za razliku od McCarthyja, eksplicira da se funkcije mogu definirati izrazima oblika

$$\text{funkcija} = \lambda[[x_1; \ldots; x_n]; \text{forma}]$$

$$\text{funkcija}[x_1; \ldots ; x_n] = \text{forma}.$$

Slagle je definirao niz zanimljivih funkcija, primjerice procedure za ubacivanje elemenata u uređene liste i za sortiranje lista, kao i predikate *forall* (za svaki) i *therex* (postoji), te funkciju *depth* koja izračunava "maksimalni nivo zagrada" u S-izrazu.

$$forall[s; p] = null[s] \vee [p \circ f[s] \;\&\; forall[r[s]; p]]$$

$$therex[s; p] = \sim forall[s; \lambda[[s_i]; \sim p[s_i]]]$$

$$depth[s] = [atom[s] \rightarrow 0.0;$$
$$\texttt{TRUE} \rightarrow max[[1.0 + depth \circ f[s]];$$
$$depth \circ r[s]]].$$

Slagleov Lisp nije nikada implementiran.

203 Slagle, *A heuristic program* ..., 1961., str. 16.

13.

Simbolički izrazi kao sintaksa jezika

Izvorno[204], sintaksa Lispa je sličila sintaksi Algola i drugih viših programskih jezika. U prvim mjesecima, ta sintaksa poprima uniformni oblik meta-izraza. Programeri su osobno prevodili programe u asembler, stječući iskustva potrebna za pisanje kompajlera. Neočekivana posljedica definicije i implementacije S-funkcije *eval* je pojava upotrebljivog interpretera za programe predstavljene u memoriji računala u obliku strukture lista. Maling i Silver su implementirali funkciju `READ` koje čita simboličke izraze i pretvara ih u strukturu liste što je omogućavalo programerima da počnu pisati programe u obliku simboličkih izraza.

Lisp zajednica nije lako prihvatila simboličke izraze kao sintaksu jezika. Već ih je, uostalom, McCarthy ocijenio "teškima za čitanje"[205], "donekle čitljivima"[206], a M-izraze "lakšima za čitanje".[207] Kasnije je predložio označavanje zagrada proizvoljnim brojem točaka i zareza, primjerice `(. . i . .)`, pri čemu bi zatvaranje takvih zagrada zatvorilo i sve ostale zagrade koje su ostale otvorene.[208] I neki drugi članovi AI projekta smatrali su Lisp nepraktičnim zbog velikog broja zagrada.[209, 210] Lisp programeri, uočio je McCarthy, ako problem zahtjeva znatniji rad sa simboličkim izrazima rado odabiru druge reprezentacije podataka i implementiraju vlastite, *ad hoc* prevodioce. Zato je smatrao da bi podrška za izraze u drugim oblicima trebala biti

204 vidi poglavlje *Imperativni Lisp*.

205 "This notation is rather formidable for a human to read …" McCarthy, *Recursive functions …,* AIM-008, 1959., str. 14.

206 "This notation is writable and somewhat readable." McCarthy, *Recursive functions …*, CAMC, 1960., str. 189.

207 McCarthy et al., *LISP I programmer's manual*, 1960., str. 53.

208 McCarthy, *New eval function*, AIM-034, 1962., str. 2.

209 "I thought it was a marvelous sort of language, except that it was somehow impractical. You can't ask people to do that; you need a translator. Such programs for LISP may exist now. I don't know. A program you could give simple statements to that would be automatically be translated into the multiply-parenthesized code." Fox, *An interview with Phyllis A. Fox*, 2005., str. 31.

210 "The task of writing out S-expressions to define programs is a tedious one, especially since the LISP notation seems to run counter to the natural way that people think of mathematical expressions." Abrahams, *Application of LISP to checking mathematical proofs*, 1964., str.159.

ugrađena u Lisp.[211] Maling i Silver su osim nikad dovršene[212] podrške za meta-izraze pokušali podržati i "algebarske izraze" poput `a*b+c*d`.[213] Razmišljalo se i o mogućnosti da S-funkcija *eval* prihvaća argumente u "uobičajenoj notaciji".[214] Programi u člancima i knjigama su još dugo pisani u obliku M-izraza. Manje je poznato da je Lisp 1.5 podržavao i izraze oblika

$$\texttt{(IF (}a\ \texttt{THEN}\ b\ \texttt{ELSE}\ c\texttt{))}^{215}$$

i sličan oblik `FOR` izraza. Nakon McCarthyjevog odlaska sa MIT-ja pojavljuju se i javne kritike[216] i pokušaji stvaranja alternativnih oblika sintakse.[217,218]

Ipak, zajednica je zadržala simboličke izraze kao sintaksu jezika i postepeno prestala koristiti M-izraze. U priručniku za Lisp 1.5, prog-izraz je uveden kao meta-izraz i ekvivalentan S-izraz, ali bez eksplicitnih pravila za prevođenje.[219] Prvi veći dokument u kojem nema spomena M-izraza je *Handbook of LISP functions* grupe studenata iz Baltimorea, iz kolovoza 1961.[220]

211 McCarthy, *LISP – notes on its past and future*, 1980., str. vi.

212 McCarthy, *History of LISP*, 1981., str. 179.

213 Maling, *The Maling-Silver read program*, AIM-013, 1959.

214 McCarthy et al., *LISP programmer's manual*, 1959., str. 3/1.

215 McCarthy et al., *LISP 1.5 programmers manual*, 1962., str. 98.

216 "… it has become evident that the general list processing languages (such as LISP) are not appropriate for processing complex mathematical data. This is due not only to the extreme complexity of the coding that necessarily ensues in the analysis of an intricate problem, but also to the fact that the input-output procedures most ideally suited for these languages are totally inadequate for the individual prlmarily interested in the analysis of a problem and not the underlying mathematical and input-output structure involved." Morris, *Models for mathematical systems,* 1966., str. 1520.

217 Henneman, *An auxiliary language for more natural expressions …*, 1964.

218 Abrahams et al., *The LISP 2 programming language and system*, 1966.

219 McCarthy et al., *LISP 1.5 programmer's manual*, 1962., str. 29.

220 Conrad et al., *A handbook of LISP functions*, 1961.

14.

Fortranolike naredbe, funkcija program, moć i multiparadigmatičnost Lispa

McCarthyjev Memo 11.[221] još uvijek je uvodio "čisti Lisp", a već Memo 12.[222] ponovo uvodi naredbe za pridruživanje, kako ih McCarthy naziva, "fortranolike" (engl. *Fortran-like*) naredbe. Primjerice, ako je "stanje računala" određeno varijablom x koja ima vrijednost `(A)` i varijablom y koja ima vrijednost `(B,B)`, onda će nakon izvršavanja fortranolikih naredbi

$$x = append[x; x]$$

$$y = cons[x; y]$$

varijabla x imati vrijednost `(A,A)`, a varijabla y će imati vrijednost `((A,A),B,B)`. Po McCarthyju, dva su razloga zbog kojih su fortranolike naredbe dobre. Prvo, mnogi programi i funkcije mogu biti zapisani konciznije, sa većom nezavisnošću dijelova nego li u "čistom Lispu". Drugo, mnogi programi i funkcije mogu biti efikasniji.

14.1. KODIRANJE STANJA STROJA I FORTRANOLIKIH NAREDBI.

Istovremeno, fortranolike naredbe nisu, po McCarthyju, više od pogodnosti. Stanja računala mogu se *kodirati* u liste parova, poput onih korištenih u definiciji S-funkcije *eval*. Primjerice, početno i završno stanje računala iz prethodnog primjera mogu se kodirati u S-izraze

`((X,(A)),(Y,(B,B)))` i

`((X,(A,A)),(Y,((A,A),B,B)))`.

Na kodiranim stanjima računala mogu se definirati S-funkcije ekvivalentne djelovanju fortranolikih naredbi na stanja računala.

Nadalje, nizovi fortranolikih naredbi se također mogu kodirati u liste parova. Primjerice,

221 McCarthy, *Recursive functions* ..., AIM-011, 1959.

222 McCarthy, *Programs in LISP*, AIM-012, 1959.

$$x = append[x; x]$$

$$y = cons[x; y]$$

se može kodirati u `((X,(APPEND,X,X)),(Y,(CONS,X,Y)))`.

14.2. S-FUNKCIJA PROGRAM

Moguće je definirati S-funkciju *program* koja primjenjuje kodirani niz fortranolikih naredbi na kodirana stanja računala. Primjerice,

program[`((X,(APPEND,X,X)),(Y,(CONS,X,Y)))`;
`((X,(A)),(Y,(B,B)))`] =
= `((X,(A,A)),(Y,((A,A),B,B)))`.

Općenito, ako niz fortranolikih naredbi s_1, s_2, …, s_n mijenja stanje računala a_1 u stanje računala a_2, onda

$$program[(s_1{*}, s_2{*}, \ldots, s_n{*}); a_1{*}] = a_2{*},$$

gdje su s_1*, s_2*, …, s_n*, a_1* i a_2* redom kodovi. S-funkciju *program* McCarthy je definirao meta-izrazom

program[*p*; *a*] = [*null*[*p*] → *a*;
T→ *program*[*cdr*[*p*];
change[*a*;
caar[*p*];
eval[*cadar*[*p*]; *a*]]]].

Pomoćna S-funkcija *change* definira ili mijenja vrijednost varijable u predstavljenom stanju stroja. Primjerice,

change[`()`; `Y`; `C`]=`((Y,C))`,

change[`((X,A),(Y,B))`; `Y`; `C`]=`((X,A),(Y,C))`.

Funkcija *change* je definirana meta-izrazom

change[*a*; *var*; *val*] =
[*null*[*a*] → *list*[*list*[*var*; *val*]];
caar[*a*] = *var* → *cons*[*list*[*var*; *val*];*cdr*[*a*]];
T → *cons*[*car*[*a*]; *change*[*cdr*[*a*];*var*; *val*]]]

U istom memou, McCarthy površno opisuje i neke druge zanimljive mogućnosti. Ekvivalent poznate GO TO naredbe može se implementirati posebnim tumačenjem simbola IL. Tako bi se *program2*[((IL,*e*)...); *a*] izračunavao kao *program2*[*eval*[*e*; *a*]; *a*].

14.3. SIMULTANO IZVRŠAVANJE FORTRANOLIKIH NAREDBI I UREĐAJ MEĐU SIMBOLIMA

Nadalje, McCarthy definira funkciju *program3* tako da se fortranolike naredbe izračunavaju simultano, primjerice,

program3[((X,Y),(Y,X)); ((X,A),(Y,B))] =
((X,B),(Y,A)).

U implementaciji posljednjeg programa McCarthy koristi relaciju uređaja (<) među simbolima i pretpostavlja da za svaki vektor stanja oblika

$((var_1,val_1),(var_2,val_2),\ldots,(var_n,val_n))$

vrijedi $var_1 < var_2 < \ldots < var_n$.

14.4. MULTIPARADIGMATIČNOST LISPA

Memo 12. otkriva više o McCarthyjevom razumijevanju Lispa nego što se čini na prvi pogled. Prva verzija Lispa je bila pragmatična i imperativna; ljepota "čistog Lispa" je usputni, čak i za članove AI grupe iznenađujući rezultat[223]

223 "And then, two years later came John's paper ... That changed the whole ball game, and it changed how people perceived LISP. Now all of a sudden, LISP was not merely a language you used to do things. It was now something you looked at: an object of beauty. It was something to be studied as an object in and of itself." Abrahams, *Transcript of discutant's remarks* in McCarthy, *History*

koji ih nije potaknuo da inzistiraju na "čistoći", a zbog čega je Lisp kasnije ponekad oštro kritiziran.[224] Štoviše, argument u prilog fortranolikih naredbi[225] se može primijeniti i na druge jezične konstrukte koje bi se moglo dodati Lispu. Kako je "čisti Lisp" Turing-potpun, to za svako zamislivo proširenje "čistog Lispa" postoji ekvivalentna funkcija "čistog Lispa" koja stanju računala — onakvom kao što ga je definirao McCarthy ili možda nešto drugačijem — pridružuje novo stanje računala, pa se onda i to proširenje može dodati u Lisp. Za bilo koji programski jezik PL, moguće je definirati S-funkciju *pl* koja se za programe i stanja stroja kodirane u izračunava vrijednosti programa u PL definirana na za svaki program u programskom jeziku PL, kodiran u S-izraz *p* i svako stanje stroja *x* izračunava novo stanje stroja, jednako kao da je PL primijenjen na P.

$$x' = pl(p,x).$$

Takvo dodavanje je, slijedi li se McCarthyjevo razmišljanje, samo pogodnost. Nasuprot Stoyanovom stavu[226], čini

of LISP, 1981., str. 193.

224 "It needs to be said very firmly that LISP, at least as represented by the dialects in common use, is not a functional language at all. LISP does have a functional subset, but that is a rather inconvenient programming language and there exists no significant body of programs written in it. Almost all serious programming in LISP makes heavy use of side effects and other referentially opaque language features. I think that the historical importance of LISP is that it was the first language to provide garbage-collected heap storage. This was a very important step forward. For the development of functional programming, however, I feel that the contribution of LISP has been a negative one. My suspicion is that the success of LISP set back the development of a properly functional style of programming by at least ten years."
Turner, *Functional programs as executable specifications*, 1984., str. 387.

225 "Since all computable functions can be expressed in LISP without the program feature, this feature can only be regarded as a convenience. However, it is a convenience at which we cannot afford to sneer."
McCarthy, *Programs in LISP*, AIM-012, 1959., str. 1.

226 "As things stand, he must prefer SCHEME to Common LISP — a clear, understandable small diamond, to a messy, incomprehensible clump."

se da je McCarthy u vrijeme pisanja Memoa 12 bio vrlo otvoren prema proširenjima jezika i tako bliži shvaćanju Lispa izraženom danas u "multiparadigmatskom" dijalektu Common Lisp nego u "elegantnijem" dijalektu Scheme. Taj stav je, čini se, McCarthy zadržao i kasnije, ograđujući se samo od proširenja koja su narušavala sintaksu jezika.[227]

14.5. MOĆ JEZIKA SA STANOVIŠTA PROGRAMERA

U Memou 12., McCarthy ističe, slično kao u *Prijedlogu kompajlera* da je "moć programskog jezika" ili "moć sustava" *sa stanovišta programera* najvažniji cilj u razvoju Lispa.[228] Taj je entuzijastički stav autora jezika, začuđujuće, rijetko citiran.

Primjerice, Lisp je moćniji od asemblera zato što programer u Lispu ne mora uvoditi pomoćne koncepte poput adresa na koje se sprema vrijednost varijabli ili *lista slobodnog prostora*. Slično, programski jezik u kojem bi se matrice `A` i `B` mogle množiti direktno, izrazom poput `A*B`, bio bi moćniji od jezika u kojem programer mora eksplicitno izračunati vrijednost svakog elementa rezultirajuće matrice. Općenito, programski sustav je *moćniji* od drugog programskog sustava ako funkcije koje mogu biti opisane

Stoyan, *The Influence of the Designer on the Design*…, 1991., str. 424.

227 "Steele: Let's move up to the list to question no. 3. What would you add to Lisp if you could goback in time? What would you delete and what would you change?
McCarthy: People have added some Englishy stuff and at least the syntax of that is not in the spirit of Lisp. I don't have any objection to the content and generally I don't have any objection to things being added to Lisp, because you can always not use them. I don't have any particular ambitions to add anything in particular to Lisp right now. I'd like to add some direct logic, but I don't know how to do that in a good way.
Steele, *Interview with John McCarthy*, 2009.

228 "In developing LISP our first goal is to describe a language which is as powerful as possible from the point ot view of the programmer. More precisely, we wish to be able to describe the transformation between the input of a program and the desired output as directly as possible."
McCarthy, *Programs in LISP,* AIM-012, 1959., str. 4.

direktno u jednom sustavu zahtjevaju opisivanje pomoćnih izračunavanja u drugom sustavu.[229] Do pomoćnih izračunavanja još uvijek dolazi, samo što je to zadaća interpretera ili kompajlera.

S vremenske distance, mogli bi se upitati kako bi izgledao jezik u kojem su pomoćna izračunavanja minimizirana. Jasno je da bi takav jezik morao imati što veću biblioteku standardnih funkcija, čemu, uostalom, svi dizajneri jezika teže. Posebno podržavao bi "deklarativno programiranje" u kojem se opisuje što, a ne kako izračunati. Doduše, puno kasnije, McCarthy se zalagao i za uključivanje podrške za logičke sudove[230] i povezivanje Lispa i Prologa.[231] S druge strane, jezik bi morao ostati "otvoren", tako da se često ponavljana šablonska izračunavanja — ma kako složena bila — mogu opisati jednostavnim sintaktičkim konstrukcijama. Još of Fortrana, za tu svrhu se koriste funkcije. No, same funkcije, bar onako kao što su definirane u "čistom Lispu" ne dozvoljavaju izbjegavanje svih šablonskih izračunavanja.[232] Ipak, najambiciozniji kasniji pokušaji razvoja Lispa u tom smjeru, primjerice, Carl Hewittov Planner ili Brian Smithov 3-Lisp nisu utjecali na najpopularnije dijalekte jezika.

McCarthy uočava i neke neefikasnosti Lispa. Podržavanjem S-funkcija Lisp uvijek izračunava nove vrijednosti, nikad ne mijenja stare vrijednosti. Taj je problem rješiv na nekoliko načina.[233] Druga neefikasnost je u potrebi za pretraživanjem liste asocijacija u potrazi za vrijednosti pridruženoj varijabli.

229 "We shall consider one programming system more powerful than another if functions which can be described directly in the one require the description of auxiliary computation in the other." McCarthy, *Programs in LISP*, AIM-012, 1959., str. 5.

230 McCarthy, *Beyond Lisp*, 2006., sl. 4.

231 McCarthy, *Guy Steele interviews John McCarthy*, father of Lisp, 2009.

232 Vidi poglavlje *Lisp 1.5*, *Specijalne forme*.

233 Vidi poglavlja *Lisp 1. 5* i *Memoizacija*.

15.

Woodward-Jenkinsova aritmetika

Ako bi programer u "čistom Lispu" želio implementirati aritmetiku, brojeve bi trebalo predstaviti kao simboličke izraze, primjerice, 857 kao `(8,5,7)` i onda implementirati algoritme poput onih kojima se ljudi služe pri računanju osnovnih računskih operacija.

D. P. Jenkins je 1960. ili 1961. implementirao Lisp na računalu TREAC u *Royal Radar Establishmentu* u Malvernu. Jenkins i Philip Woodward su 1961.[234] opisali program za zbrajanje brojeva. Znamenke su simboli koji se nalaze u listi *tab* koja ima vrijednost

```
(0,1,2,3,4,5,6,7,8,9).
```

Cijeli brojevi su predstavljeni kao liste podlista *tab*, zapisane obrnutim redom. Primjerice, broj 857 je predstavljen listom

```
((7,8,9),(5,6,7,8,9),(8,9)).
```

Funkcije potrebne za zbrajanje lista su

$$sum[x; y] = [null[y] \rightarrow x;\ \mathtt{T} \rightarrow inc[x; y; tab]]$$

$$inc[x; y; b] = [eq[caar[y]; car[b]] \rightarrow cons[car[x]; sum[cdr[x]; cdr[y]]];\ \mathtt{T} \rightarrow inc[step[x]; y; cdr[b]]]$$

$$step[x] = [null[x] \rightarrow cons[cdr[tab]; \mathtt{NIL}];\ null[cdar[x]] \rightarrow cons[tab; step[cdr[x]]];\ \mathtt{T} \rightarrow cons[cdar[x]; cdr[x]]].$$

McCarthy je kasnije pisao da je i izvorna grupa na MIT-ju pokušala predstaviti brojeve na sličan način, ali da je taj pristup odbačen zbog sporosti.[235] Autor ove knjige nije pronašao dokument koji bi te pokušaje opisao, a niti članovi izvorne grupe s kojima se konzultirao takvih se pokušaja

234 Woodwards & Jenkins, *Atoms and lists*, 1961., str. 52.

235 Numbers were originally implemented in LISP I as lists of atoms, and this proved too slow for all but the simplest computations. A reasonably efficient implementation of numbers as atoms in S-expressions was made in LISP 1.5
McCarthy, *History of Lisp*, 1981., p. 180.

nisu sjećali. U Lispu I., kao i u kasnijem Lispu 1.5, brojevi su već implementirani kao poseban tip atoma. Takva implementacija zasigurno narušava ljepotu jezika i dovodi njegovu upotrebljivost u sumnju, ali je, čini se, bila neizbježna. Čak i danas, u svim praktičnim implementacijama Lispa, brojevi su implementirani kao poseban tip podataka.

16.

Lisp 1.5

Unatoč ljepoti, "čisti Lisp" je bio neefikasan i praktično neupotrebljiv.[236] Iskustvo s nizom malih problema je uvjerilo McCarthyja da podrška za simboličke izraze nije dovoljna i da Lisp mora omogućiti i računanje s drugim vrstama podataka.[237] Stoga je daljnji razvoj jezika usmjeren prema praktičnim aspektima jezika. Taj period, nažalost, nije tako dobro dokumentiran[238] kao prethodni, i moguće je da su neke zanimljive ideje zauvijek izgubljene.

LISP programmer's manual iz ožujka 1959. sadrži popis funkcija i njihove definicije u Lispu ili asembleru IBM 904. U to vrijeme, sintaksa Lispa je još uvijek slična onoj u "imperativnom Lispu". [239]

Znatno opsežniji *Lisp preliminary programmer's manual* iz siječnja 1960.[240] te *Lisp I programmer's manual* iz travnja 1960.[241] donose mnogo promjena. Između ostaloga, sintaksa "imperativnog Lispa" je konačno odbačena.

Sve značajne promjene u Lispu I. su našle mjesto i u Lispu 1.5 koji se u internim dokumentima spominje već u proljeće 1961.[242] *LISP 1.5 programmer's manual* objavljen je u kolovozu 1962. *Priručnik* je, čini se, ostavio veći utje-

236 "LISP 1 was characterized by great elegance, but in practice it turned out to be a language in which it was impossible to write useful programs. This situation led to many additions to LISP 1, and the result of these additions has become known as LISP 1.5 (since it was believed to be halfway between LISP 1 and LISP 2)." Abrahams pod LISP 1. misli na "čisti LISP" – privatna komunikacija
Abrahams, *Symbol manipulation languages*, 1968., str. 69.

237 "Experience with the present LISP system on a number of small problems has shown the importance of being able to compute with other quantities than symbolic expressions, and a revised version of LISP is being considered which will allow computation of recursive functions of a wide variety of kinds of information."
McCarthy, *The LISP programming system*, RLE QPR 056, 1960., str. 159.

238 "… they never documented or wrote down anything, especially McCarthy. Nobody in that group ever wrote down anything. McCarthy was furious that they didn't document the code, but he wouldn't do it, either."
Fox, *An interview with Phyllis A. Fox*, 2005., str. 30.

239 McCarthy et al., *LISP programmer's manual*, 1959.

240 McCarthy et al., *LISP preliminary programmer's manual*, 1960.

241 McCarthy et al., *LISP I. programmer's manual*, 1960.

242 Levin, *Arithmetic in LISP 1.5*, AIM-024, 1961.

caj negoli članak iz 1960. Primjerice, Kay, Meyer i Dijkstra, diskutirajući o *evalu*, govore o verziji iz *Priručnika*.

Manual ostavlja dojam zbornika relativno nezavisnih članaka više nego li cjelovite knjige. Oštro ga je kritizirao Edsger W. Dijkstra.[243] Ponajbolja dopuna priručniku je članak Robert A. Saundersa.[244]

16.1. ČISTI LISP

Prvo poglavlje *Priručnika* donosi kratak rezime "čistog Lispa", s nekim izmjenama. Između ostaloga, umjesto termina "S-funkcija" koristi se jednostavno "funkcija".

Osim zareza, za razdvajanje elemenata liste mogu se koristiti i razmaci. Primjerice, umjesto `(CAR,X)` može se pisati `(CAR X)`.

Kao liste parova, koje se u Lisp 1.5 nazivaju i liste asocijacija (engl. *association lists*) ne koriste se više obične liste od dvaju elemenata nego "točkasti parovi". Primjerice, umjesto `((X (A B)) (Y C))` koristi se `((X . (A B)) (Y . C))`. Ta je mala promjena uvedena vjerojatno radi uštede memorije. Ipak, neke funkcije su redefinirane, a uvedene su i neke nove.

Funkcija *member*[*x*; *y*] je predikat koji je istinit ako i samo ako je simbolički izraz *x* element liste *y*. Primjerice,

$$member[\texttt{X}; \texttt{(A X)}] = \texttt{T},$$

$$member[\texttt{Y}; \texttt{(A X)}] = \texttt{F}.$$

Definicija funkcije je

$$\begin{aligned} member[x; y] = [&null[y] \rightarrow \texttt{F};\\ &equal[x; car[y]] \rightarrow \texttt{T};\\ &\texttt{T} \rightarrow member[x; cdr[y]]]. \end{aligned}$$

Funkcija *pairlis*[*x*; *y*; *a*] stvara odgovarajuću listu parova od lista *x* i *y* i dodaje je listi asocijacija *a*. Primjerice,

$$pairlis[\texttt{(A B C)}; \texttt{(U V W)}; \texttt{((D.X) (E.Y))}] =$$

243 Dijkstra, *Trip report, Edinburgh and Newcastle,* EWD 448., 1974.

244 Saunders, *LISP — on the programming system*, 1964.

= ((A.U) (B.V) (C.W) (D.X) (E.Y)).

Definicija funkcije je

$$pairlis[x; y; a] = [null[x] \rightarrow a;$$
$$\mathtt{T} \rightarrow cons[cons[car[x]; car[y]];$$
$$pairlis[cdr[x]; cdr[y]; a]]].$$

Funkcija *assoc*[*x*; *a*] traži par simbola *x* u listi asocijacija *a*. Definicija funkcije je

$$assoc[x; a] = [equal[caar[a]; x] \rightarrow car[a];$$
$$\mathtt{T} \rightarrow assoc[x; cdr[a]]].$$

Funkcija *sublis*[*a*; *y*] supstituira vrijednosti umjesto varijabli u simbolički izraz *y*. Primjerice,

sublis[((X.S) (Y.(T T)); (X WROTE Y)] =
(S WROTE (T T)).

Definicija funkcije je

$$sublis[a; y] = [atom[y] \rightarrow sub2[a; y];$$
$$\mathtt{T} \rightarrow cons[sublis[a; car[y]]$$
$$sublis[a; cdr[y]]]].$$

Pomoćna funkcija *sub2*[*a*; *z*] rješava poseban slučaj u kojem je *z* atom. Definicija funkcije je

$$sub2[a; z] = [null[a] \rightarrow z;$$
$$eq[caar[a]; z] \rightarrow cdar[a];$$
$$\mathtt{T} \rightarrow sub2[cdr[a]; z]].$$

Zbog novog formata asocijacijskih lista, definicija funkcije *sub2* jednostavnija je nego li ranije.

Definicije funkcija *eval* i *apply* razlikuju se od onih u dokumentima iz 1959-60. Funkcija *apply* ima dodatni, treći argument, listu vrijednosti koje se supstituiraju na mjestu slobodnih varijabli te joj je povjeren dio funkcija koje je prethodno imala funkcija *eval*.

$$
\begin{aligned}
&apply[fn; x; a] = \\
&\quad [atom[fn] \rightarrow [eq[n; \texttt{CAR}] \rightarrow caar[x]; \\
&\qquad\qquad eq[fn; \texttt{CDR}] \rightarrow cdar[x]; \\
&\qquad\qquad eq[fn; \texttt{CONS}] \rightarrow cons[car[x]; cadr[x]]; \\
&\qquad\qquad eq[fn; \texttt{ATOM}] \rightarrow atom[car[x]]; \\
&\qquad\qquad eq[fn; \texttt{EQ}] \rightarrow eq[car[x]; cadr[x]]; \\
&\qquad\qquad \texttt{T} \rightarrow apply[eval[fn; a]; x; a]]; \\
&\quad eq[car[fn]; \texttt{LAMBDA}] \rightarrow eval[caddr[fn]; \\
&\qquad\qquad\qquad\qquad pairlis[cadr[fn]; x; a]]; \\
&\qquad\qquad eq[car[fn]; \texttt{LABEL}] \rightarrow apply[caddr[fn]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad x; \\
&\qquad\qquad\qquad\qquad\qquad\qquad cons[cons[cadr[fn]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad caddr[fn]]; \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad a]]]
\end{aligned}
$$

$$
\begin{aligned}
&eval[e; a] = [atom[e] \rightarrow cdr[assoc[e; a]]; \\
&\qquad\qquad atom[car[e]] \rightarrow [eq[car[e]; \texttt{QUOTE}] \rightarrow card[e]; \\
&\qquad\qquad\qquad\qquad\qquad eq[car[e]; \texttt{COND}] \rightarrow evcon[cdr[e]; a]; \\
&\qquad\qquad\qquad\qquad\qquad \texttt{T} \rightarrow apply[eval[fn; a]; x; a]]; \\
&\qquad\qquad \texttt{T} \rightarrow apply[car[e]; evlis[cdr[e]; a]]
\end{aligned}
$$

Pomoćne funkcije *evcon* i *evlis* su definirane kao u članku *Recursive functions* … S-funkcija koja se u originalnom predstavljanju "čistog Lispa" nazivala *apply* ovdje se naziva *evalquote* i definirana je izrazom

$$evalquote[fn; x] = apply[fn; x; \texttt{NIL}].$$

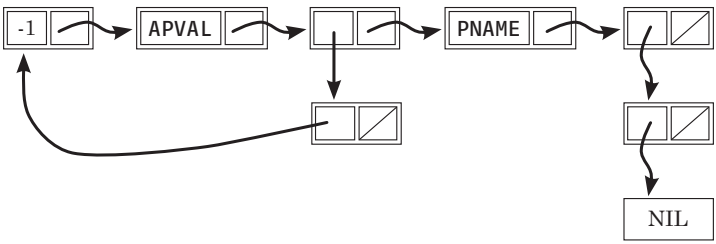
"Čisti Lisp" je ostao matematička ideja, dok je stvarni interpreter radio nešto drugačije. Uvedene su mnoge promjene koje su Lisp 1.5 učinile praktičnijim. Definirano je mnogo više funkcija, ne samo pet osnovnih. Stvarni *eval* i *apply* su bili mnogo složeniji. Osim pet elementarnih funkcija, definirano je i mnogo drugih funkcija.

Liste svojstava su interna reprezentacija simbola. Liste svojstava su slične ostalim listama, ali se o njihovom održavanju brine Lisp sistem, iako ih i programer može kontrolirati. Pogrešno bi bilo misliti da su liste svojstava vrijednosti simbola. Naprotiv, Lisp sistem samo koristi listu svojstava da bi izračunao vrijednost simbola, ako to programer traži.

Primjerice, interna reprezentacija simboličkog izraza `(A B)` je grafički prikazana na slici 17.

SLIKA 17. Grafički prikaz reprezentacije simboličkog izraza `(A B)`

Na mjestima na kojima su u grafičkom prikazu upisani `A` i `B` nalaze se adrese lista svojstava simbola `A` i `B` redom. U prekriženom, dekrementnom dijelu drugog elementa liste nalazi se adresa liste svojstava simbola `NIL`.



SLIKA 18. Grafički prikaz liste svojstava simbola `NIL`

Na primjeru liste svojstava simbola `NIL` može se vidjeti i opći oblik liste svojstava. U adresnom dijelu prvog elementa liste svojstava uvijek se nalazi binarno zapisani broj −1. U adresnom dijelu sljedećih elemenata liste svojstava nalaze se adrese indikatora, ovdje `APVAL` i `PNAME`. Indikatori su simboli koji označavaju vrstu svojstva koja neposredno slijede. Primjerice, indikator `APVAL` označava da je svojstvo koje slijedi vrijednost simbola. Simbol `NIL` je poseban slučaj: vrijednost `NIL` je sam `NIL`.

Indikator PNAME označava da je svojstvo koje slijedi ime simbola, spremljeno u jednoj ili više memorijskih riječi u tada korištenom BCD kodu.

Ako programer nekim izrazom definira funkciju onda će se u listi simbola nalaziti indikator EXPR za kojim će slijediti adresa na kojoj se nalazi reprezentacija lambda-izraza kojim je funkcija definirana. Lisp 1.5 dozvoljava i da funkcije budu napisane u mašinskom jeziku. Tada se mašinski kod te funkcije nalazi u listi svojstava simbola, iza indikatora FSUBR.

Funkcija *deflist*[((*u*$_1$ *v*$_1$) … (*u*$_n$ *v*$_n$)); *i*] ubacuje indikator *i* i njemu pridruženo svojstvo *v*$_1$ u listu svojstava simbola *u*$_1$, …, indikator *i* i njemu pridruženo svojstvo *v*$_n$ u listu svojstava simbola *u*$_n$. Primjerice, izrazom

deflist[((IDENTITY (LAMBDA (X) X)));EXPR]

definira se funkcija identiteta. Ako neki simbol *u*$_j$ već ima indikator *i* i pridruženo svojstvo, to će svojstvo biti zamijenjeno sa *v*$_j$.

Funkcija *get*[*s*; *i*] vraća svojstvo simbola *s* pridruženo indikatoru *i*. Primjerice, nakon izračunavanja gornjeg izraza

get[IDENTITY; EXPR] = (LAMBDA (X) X).

Funkcija *attrib*[*x*; *e*] konkatenira dvije liste u memoriji računala mijenjajući zadnji element prve liste tako da pokazuje na prvi element druge liste. Posebno, ako je *x* simbol, onda će *e* biti pridodan listi svojstava tog simbola. Primjerice, poziv funkcije

attrib[IDENTITY; (EXPR (LAMBDA (X) X))]

ubacuje indikator EXPR i pridruženi simbolički izraz (LAMBDA (X) X) u listu svojstava simbola IDENTITY.

Funkcija *remprop*[*x*; *i*] uklanja indikator *i* i pridruženo svojstvo iz liste svojstava simbola *x*.

Programer može umetnuti bilo koji indikator i pridruženo svojstvo u listu svojstava simbola ako to želi. No, samo neki od indikatora su korišteni od strane sistema.

Korištenje liste svojstava omogućuje da se rezultati dobiveni tijekom izračunavanja simboličkih izraza spreme i ponovo koriste kasnije, pri izračunavanju drugih simboličkih izraza. Funkcije koje koriste tu mogućnost — a koja u matematici obično ne postoji — nazivaju se pseudo-funkcijama. S-funkcije *deflist i attrib* su očigledno pseudo-funkcije.

Drugi razlog za korištenje pseudofunkcija je vezan uz strukture lista koje su uvedene samo kao interna reprezentacija simboličkih izraza, ali koje i same izazivaju probleme koji traže rješavanje. U Lispu 1.5 definiran je niz funkcija inspiriranih potrebom za efikasnijim korištenjem struktura lista. Najvažniji primjer su funkcije *rplaca* i *rplacd*. Funkcija *rplaca*[*x*;*y*] mijenja prvi element simboličkog izraza *x* s *y*. Primjerice,

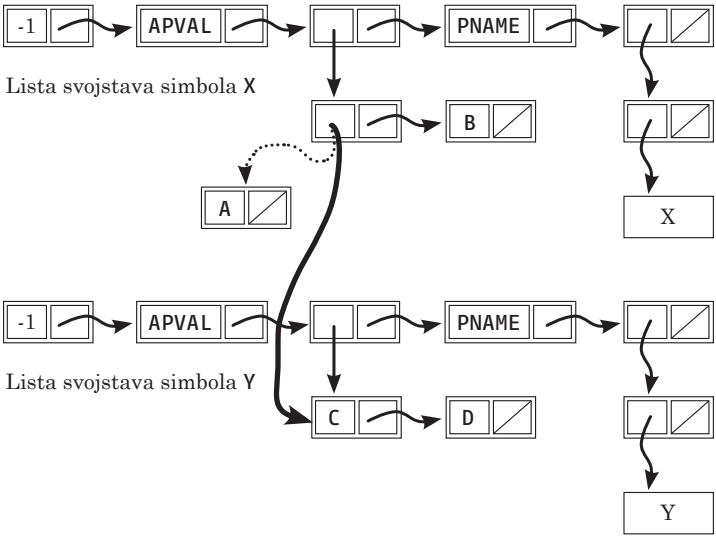
rplaca[`((A) B)`; `(C D)`]=`((C D) B)`.

Moglo bi se misliti da

$$rplaca[x; y] = cons[y; cdr[x]].$$

To i jest istina dok se misli samo u terminima simboličkih izraza. Ali, dok izračunavanje izraza *cons*[*y*; *cdr*[*x*]] uzima novi element liste s liste slobodnog prostora i postavlja adresni i dekrementni dio elementa liste na adrese *y* i *cdr*[*x*], izračunavanje *rplaca*[*x*; *y*] upisuje adresu *y* na mjesto adresnog dijela prvog elementa liste *x*.

Primjerice, neka je `((A) B)` vrijednost simbola `X` i `(C D)` vrijednost simbola `Y`. Poziv funkcije *rplaca*[*x*; *y*] ne samo da će vratiti `((C D) B)`, nego će i vrijednost `X` biti promijenjena iz `((A) B)` u `((C D) B)`. Slika 19. prikazuje što će se točno dogoditi.



SLIKA 19. Liste svojstava simbola X i Y
prije i nakon poziva funkcije *rplaca*[*x*; *y*]

Isprekidana i tanka strelica označava vrijednost adresnog dijela elementa liste prije izračunavanja *rplaca*[*x*; *y*] a debela i puna strelica označava vrijednost nakon izračunavanja *rplaca*[*x*; *y*]. Nakon izvršavanja *rplaca*[*x*; *y*] vrijednost simbola X će biti promijenjena; nakon izračunavanja *cons*[*y*; *cdr*[*x*]] neće.

Funkcija *rplacd*[*x*; *y*] na isti način mijenja dekrementni dio prvog elementa liste *x*.

Funkcije *rplaca* i *rplacd* imaju analogno djelovanje i na strukturama liste koje nisu vrijednosti simbola.

Druge važne pseudofunkcije su *cset*[*x*; *val*] i *csetq*[*x*; *val*]. Te dvije funkcije su Lisp 1.5 ekvivalent naredbe za pridruživanje, ili, kako je McCarthy prije pisao, "fortranolike naredbe" $x = val$. Obje pseudofunkcije postavljaju vrijednost simbola *x* na *val* upisujući *val* u listu svojstava simbola *x*, odmah iza indikatora APVAL. Slovo *c* ispred *set* i *setq* dolazi od engleske riječi 'constant'; simboli čija je vrijednost spremljena u listi svojstava nazivali su se u Lisp 1.5 priručniku konstantama.

Razlika između pseudofunkcija *cset* i *csetq* ne može se vidjeti iz meta-izraza, ali se može vidjeti u prijevodu meta-izraza u simboličke izraze.

meta-izraz	**prijevod meta-izraza u simbolički izraz**
cset[X; (A B C D)]	(CSET (QUOTE X) (QUOTE (A B C D))).
csetq[X; (A B C D)]	(CSETQ X (QUOTE (A B C D))).

U prijevodu *csetq* nije potreban jedan QUOTE, pa se zato *csetq* koristio češće od *cset*.

U funkciji *eval*, funkcija CSETQ mora imati drugačiji tretman od ostalih funkcija jer se prvi argument funkcije ne smije izračunati. Zato se CSETQ i ne naziva funkcijom ili pseudofunkcijom nego "specijalnom formom", kao i COND i QUOTE.

Općenitije, *specijalne forme* su izrazi oblika (e_1 e_2 … e_n) koji se izračunavaju bez izračunavanja simboličkih izraza e_2, …, e_n prije nego što se kontrola izračunavanja prenese na simbolički izraz e_1. U *LISP 1.5 Manualu* specijalnom formom se naziva i sam izraz e_1. Danas se e_1 u izrazu (e_1 e_2 … e_n) koji se izračunava obično naziva *operatorom*. Posebno, ako je (e_1 e_2 … e_n) specijalna forma, e_1 se naziva *specijalnim operatorom*.[245]

Drugo važno svojstvo specijalnih formi je da, za razliku od funkcija, mogu (ali ne moraju) dozvoljavati varijabilni broj argumenata. Primjerice, specijalna forma COND može imati proizvoljan broj argumenata.

16.5. FEXPROVI

U "čistom Lispu", specijalne forme imaju posebna pravila za izračunavanje u funkciji *eval* i programer ne može definirati vlastite specijalne operatore. U Lispu 1.5 specijalni operatori se mogu definirati lambda-izrazima kao i funkcije, a interpreter ih prepoznaje po indikatoru FEXPR[246] u listi

245 Pitman, *Special forms in Lisp*, 1980.

246 Termin *fexpr* vjerojatno je skraćeni oblik termina "*functional*

svojstava i izračunava ih drugačije. S vremenom se takve specijalne forme uvriježilo nazivati *fexprovima* da bi ih se razlikovalo od drugih vrsta specijalnih formi koje će se pojaviti tijekom razvoja Lispa. U Lispu 1.5, `CSETQ` je, primjerice, *fexpr*.

Fexprovi su vrlo moćni. Primjerice, neka je fexpr `Q2` definiran s

deflist[((Q2 (LAMBDA (V1 V2) (CAR V1))); FEXPR],

`(Q2 X)` je jednostavan simbolički izraz koji koristi fexpr i `((X A))` je lista asocijacija. Tada vrijedi

eval[(Q2 X); ((X Y))] =

= *eval*[((LAMBDA (V1 V2) (CAR V1)) (QUOTE (X))
(QUOTE ((X Y))));
((X Y))],

pa iz definicije *eval* slijedi

= *eval*[(CAR V1); ((V1 (X)) (V2 ((X Y))) (X Y))] =

= *car*[*eval*[V1; ((V1 (X)) (V2 ((X Y))) (X Y))]] =

= *car*[(X)] =

= X.

Dakle, fexpr `Q2` može se koristiti umjesto specijalnog operatora `QUOTE`. Kao što se vidi iz prethodne jednakosti, fexprovi imaju pristup listi asocijacija s kojom je pozvan *eval* koji ih primjenjuje.

Općenitije, fexprovi se definiraju izrazima oblika

deflist[((*fx* (LAMBDA (v_1 v_2) *e*)));FEXPR]

gdje je simbol *fx* ime specijalne forme, v_1 i v_2 su simboli, *e* je simbolički izraz koji vjerojatno sadrži v_1 i v_2. Neka je *a* lista asocijacija. Tada je vrijednost

expression". Potonji je termin korišten za meta-izraze u McCarthy, *RFSE*, MIT AIM-008, 1959., str. 4. U meta-izrazima, kao i u izrazima čiji je prvi element fexpr nije potreban simbol `QUOTE`.

$$eval[(fx\ e_1\ e_2\ \ldots\ e_n); a]$$

jednaka vrijednosti

$$eval[((\texttt{LAMBDA}\ (v_1\ v_2)\ e)\ (\texttt{QUOTE}\ (e_1\ e_2\ \ldots\ e_n))\ (\texttt{QUOTE}\ a)); a].$$

Fexprovi su se, kao i funkcije, mogli definirati i u mašinskom kodu. Tada bi u listi svojstava definiciji prethodio indikator `FSUBR`. Gotovo svi specijalni operatori u Lisp 1.5 sistemu, uključujući `COND`, `QUOTE`, `AND`, `OR` i `LIST` imaju indikator `FSUBR`.

U "čistom Lispu" i Lispu 1.5 funkcije ne moraju imati ime; dovoljno je koristiti lambda- ili label-izraze, primjerice `(LAMBDA (X) X)`. Takve se funkcije danas nazivaju "anonimnim funkcijama". Informacija da neki lambda-izraz treba koristiti kao fexpr može se nalaziti samo u listi svojstava simbola pa u Lispu 1.5 nisu mogući anonimni fexprovi.

Fexprovi omogućuju programeru da definira elemente jezika koji se čine temeljnijima od funkcija. Iako se ta spoznaja čini iznimno važnom i izazovnom[247], fexprovi su vrlo površno opisani, bez ijednog primjera. I u drugim dokumentima iz tog vremena fexprovi jedva da se i spominju. McCarthy je razmatrao fexprove u memou iz 1962.[248]

16.6. PROGRAMI U LISP-U

Program u Lispu 1.5 je niz parova simboličkih izraza. Primjerice, program koji definira S-funkciju

247 "… pure language was supposed to be based on functions, but its most important components — such as lambda expressions, quotes, and conds — were not functions at all, and instead were called special forms. (…) In the practical language things were better. There were not just EXPRs (which evaluated their arguments), but FEXPRs (which did not). My next questions was, why on Earth call it a functional language? Why not just base everything on FEXPRs and force evaluation on the receiving side when needed? I could never get a good answer (…)"
Kay, *Early history of Smalltalk*, 1996., str. 534.

248 Vidi poglavlje *Nova funkcija eval*.

append[*x*; *y*] = [*null*[*x*] → *y*;
T → *cons*[*car*[*x*]; *append*[*cdr*[*x*]; *y*]]].

i nakon toga izračunava *append*[(A B); (B C)] je

```
DEFINE (((APPEND
          (LAMBDA (X Y)
            (COND ((NULL X) Y)
                  (T (CONS (CAR X)
                           (APPEND (CDR X)
                                   Y)))))))))
APPEND ((A B) (B C)) .
```

Interpreter redom poziva funkciju *evalquote* s argumentima koji su parovi simboličkih izraza u programu. U ovom primjeru, interpreter izračunava, redom

evalquote[DEFINE; (((APPEND …)))],

evalquote[APPEND; ((A B) (B C))].

Funkcija *evalquote* prema potrebi poziva funkciju *apply*, funkcija *apply* prema potrebi poziva funkciju *eval* itd.

Lisp 1.5 interpreter "pamti" neke vrijednosti simbola i nakon što završi izračunavanje poziva *evalquote*. U gornjem primjeru, interpreter je u prvom pozivu u listu svojstava simbola APPEND upisao vrijednost (LAMBDA (X Y) …). U drugom pozivu, *evalquote* izračunava

apply[APPEND; ((A B) (B C)); NIL].

Funcija *apply* onda izračunava

eval[APPEND; NIL]

pri čemu interpreter prvo traži vrijednost APPEND u pridruženoj listi svojstava, a ako je tamo ne nađe, onda u listi asocijacija *a*. Dakle, jednom definirana funkcija APPEND može se koristiti u cijelom programu. Tako nešto nije bilo moguće u "čistom Lispu" u kojem se APPEND može koristiti samo ako se definicija nalazi u listi asocijacija, primjerice

eval[`(APPEND (QUOTE (A B)) (QUOTE (B C));`
`((APPEND (LAMBDA (X Y) (COND ... ))))`].

16.7. FUNKCIONALNI ARGUMENTI

Funkcionali, funkcije koje prihvaćaju druge funkcije za argumente su već razmatrane u poglavljima *Elementi funkcionalnog programiranja* i *Čisti Lisp*. U jesen 1959. Slagle je naišao na neočekivane probleme kasnije nazivane "*funarg problemima*".[249] Ti su problemi jedva naznačeni u *LISP 1.5 Manualu*[250] ali Saunders[251] navodi primjer sličan Slagleovom[252,253] koji pokazuje zašto naivno korištenje funkcija kao argumenata funkcije ne daje očekivane rezultate. Taj je primjer ovdje izložen u nešto pojednostavljenom obliku.

Neka je funkcija `TESTR` definirana lambda-izrazom

```
(LAMBDA (L FN)
  (COND ((NULL L) NIL)
        (T (TESTR (CAR L)
                  (QUOTE (LAMBDA ()
                            (CONS (CDR L)
                                  (FN)))))))).
```

Nakon poziva

249 Stoyan, *The influence of the designer on the design*, 1991., str. 420.

250 "We also need a special rule to translate functional arguments into S-expression. If *fn* is a function used as an argument. then it is translated into `(FUNCTION` *fn*`*)`. Example

```
(CHANGE (LAMBDA (A)
                (MAPLIST A
                         (FUNCTION (LAMBDA (J)
                                           (CONS (CAR J)
                                             (QUOTE X)))))))
```

An examination of *evalquote* shows that `QUOTE` will work instead of `FUNCTION` provided that there are no free variables present." McCarthy et al., *LISP 1.5 programmers manual*, 1962., str. 21.

251 Saunders, *LISP — on the programming system*, 1964.

252 Stoyan, *LISP history*, 1979., str. 47.

253 McCarthy, *History of LISP*, 1981., str. 180.

```
(TESTR (QUOTE ((A) (B) (C)))
       (QUOTE (LAMBDA (X) X)))
```

parametri L i FN bi imali sljedeće vrijednosti:

```
L:  ((A) (B) (C))
FN: (LAMBDA (X) X).
```

Kako nije zadovoljen uvjet (NULL L), izvršavala bi se druga grana *cond-izraza*

```
(TESTR (CAR L)
       (QUOTE (LAMBDA ()
                      (CONS (CDR L)
                            (FN)))))
```

Nakon ponovnog poziva funkcije TESTR, vrijednosti parametara bi bile

```
L:  (A)
FN: (LAMBDA()(CONS (CDR L) (FN))).
```

U izrazu koji je vrijednost parametra FN pojavljuju se simboli L i FN, ali vrijednost tih simbola je sada različita od vrijednosti koju su imali u trenutku u kojem je pozvana funkcija G.

McCarthy nije pokazao interes za ovaj problem pa su rješenje razvili Russell, Edwards i Patrick Fischer.[254] Da bi Lisp interpreter korektno izračunavao ovakve izraze,

254 "And I never understood the complaint very well, namely, I said : 'oh, there must be a bug somewhere, fix it!' And Steve Russell and Dan Edwards said, there is a fundamental difficulty and I said, there can't be, fix it, and eventually they fixed it and there was something they called the funarg device. He tried to explain it to me but I was distracted or something and I didn't pay much attention so I didn't really learn what the funarg thing did until really quite a few years later. But then it turned out that these same kinds of difficulties appeared in ALGOL and at the time, the LISP solution to them, the one that Steve Russell and Dan Edwards had simply cooked up in order to fix this bug, was a more comprehensive solution to the problem than any which was at that time in the ALGOL compiler."
McCarthy, *Talk about LISP history*, 1974., citiran prema Stoyan, *History of Lisp*, 1979., str. 47.

nužno je da s izrazima budu sačuvane i originalne vrijednosti varijabli. U Lispu 1.5 za to služi operator FUNCTION posredstvom kojeg se funkcije mogu koristiti kao argumenti drugih funkcija. Tako, TESTR treba definirati kao

```
(LAMBDA (L FN)
  (COND ((NULL L) NIL)
        (T (TESTR (CAR L)
                  (FUNCTION (LAMBDA ()
                                    (CONS (CDR L)
                                          (FN)))))))).
```

Tada će nakon poziva

```
          (TESTR (QUOTE ((A) (B) (C)))
                 (QUOTE (LAMBDA (X) X)))
```

vrijednosti parametara biti

```
               L:   ((A) (B) (C))
               FN:  (LAMBDA (X) X)
```

Kako nije zadovoljen uvjet (NULL L) izvršavala bi se druga grana cond-izraza.

```
     (TESTR (CAR L)
            (FUNCTION (LAMBDA ()
                              (CONS (CDR L)
                                    (FN)))))
```

Nakon ponovnog poziva funkcije TESTR vrijednosti parametara bi bile

```
L: (A),
FN: (FUNARG (LAMBDA()(CONS (CAR L) (FN)))
           ((L . ((A) (B) (C)))
           (FN . (LAMBDA (X) X)))).
```

LISP sistem tako raspolaže svim podacima potrebnima za izračunavanje.

Općenitije,

$$eval[\texttt{(FUNCTION}\ fn\texttt{)}; q] = \texttt{(FUNARG}\ fn\ q\texttt{)}$$

$$apply[\texttt{(FUNARG}\ fn\ q\texttt{)}; x; p] = apply[fn; x; q].$$

Budućnost će pokazati da Lisp zajednica nije bila zadovoljna ovim rješenjem. I sam McCarthy, nakon početne nezainteresiranosti, izrazio je nezadovoljstvo rješenjem.[255]

16.8. SPECIJALNI OPERATOR PROG

Funkcija *program*[256] opisana u jednom od prethodnih poglavlja uvedena je u Lisp 1.5 kao specijalni operator *prog* i u dosta razrađenom obliku. Primjerice, funkcija *length* koja izračunava duljinu liste, definirana je s

$$\begin{aligned} length[l] = prog[\ & [u; v];\\ & v := 0;\\ & u := l;\\ & \texttt{A}\ [null[u] \rightarrow return[v]]\\ & u := crd[u];\\ & v := v + 1;\\ & go[\texttt{A}]]. \end{aligned}$$

Prijevod te definicije u simboličke izraze je

```
DEFINE (((LENGTH (LAMBDA (L)
                    (PROG   (U V)
                            (SETQ V 0)
                            (SETQ U L)
                          A (COND ((NULL U)
                                   (RETURN V)))
                            (SETQ U (CDR U))
                            (SETQ V (ADD1 V))
                            (GO A) ))))).
```

255 "Thus the system eval will accept functional arguments only when they are preceded by FUNCTION, which is both a theoretical and practical nuisance."
McCarthy, *New eval function*, AIM-034, 1962., str. 8.

256 Vidi poglavlje *Fortranolike naredbe i funkcija program*.

Prvi argument *prog*-izraza je lista "programskih varijabli" koje se tretiraju jednako kao parametri u lambda-izrazima. Ako nema programskih varijabli onda treba pisati `()` ili `NIL`.

Ostali argumenti prog-izraza su naredbe i atomski simboli. Prog-izraz se izračunava izvršavanjem *naredbi* po redu po kojem su zapisane u prog-izrazu. Termini "naredba" i "izvršavanje" nasuprot "izrazu" i "izračunavanju" ukazuju na bitnu razliku: dobivena vrijednost se ignorira.

Unutar prog-izraza mogu se "postavljati varijable". Varijable se postavljaju M-izrazima poput $a := b$.

Naredba *return*[*e*] se izvršava tako da se izračuna *e* nakon čega prog-izraz završava izračunavanje s vrijednošću koja je dobivena izračunavanjem e. Ako prog-izraz ostane bez naredbi koje treba izvršiti, završava izračunavanje s vrijednošću `NIL`.

Naredba *go*[*s*] uzrokuje nastavak izvršavanja prog-izraza od naredbe koja se nalazi odmah iza simbola *s*.

Prevođenje prog-izraza u S-izraze je nešto složenije i uopće nije opisano u literaturi. Iz primjera je vidljivo da se M-izrazi $a := b$ prevode u `(SETQ` *a** *b**`)`, pseudo-funkcije `SET` i `SETQ` se ponašaju kao `CSET` i `CSETQ`, osim što se primjenjuju na programske varijable. Naredba *go*[*s*] se prevodi u `(GO` *s*`)` a naredba *return*[*e*] u `(RETURN` *e**`)`. Unutar prog-izraza simboli koji služe kao imena naredbi se prevode direktno, bez `QUOTE`.

16.9. GENSYM I OBLIST

Jedna od najzanimljivijih funkcija u Lispu 1.5 je *gensym*[ ] koja svaki put kad je pozvana generira novi simbol, prethodno nekorišten u programu. Ti simboli su u Lispu 1.5 imali imena `G00001`, `G00002`, `G00003` …

Manual ne opisuje kako se funkcija *gensym* može koristiti. Iz jedva nekoliko dokumentiranih primjera upotrebe *gensym* mogu se uočiti dva, donekle različita razloga za upotrebu.

Generiranje novih, prethodno nekorištenih simbola je često potrebna operacija pri procesiranju logičkih formula. Primjerice, prva dva aksioma iskaznog računa, zapisani u obliku simboličkih izraza su:

1. `(→ A A)`

2. `(→ B (→ A B))`

Uvrsti li se aksiom 1. umjesto `B` u aksiom 2. dobit će se

3. `(→ (→ A A) (→ A (→ A A)))`

Iz 1. i 3. po pravilu *modus ponens* slijedi

4. `(→ A (→ A A))`

Taj je teorem slabiji od teorema `(→ A (→ G00001 G00001))` koji bi se dobio da se svi simboli korišteni u 1. teoremu zamjene s novim, prethodno nekorištenim simbolom. Izraz

sublis[*cons*[*cons*[`A`; *gensym*[ ]]; `NIL`]; `(→ A A)`]

se izračunava u `(→ G00001 G00001)`, ako je `G00001` prvi nekorišteni simbol. Na sličan je način funkcija *gensym* korištena u Abrahamsovom programu PROOFCHECKER.[257,258]

Drugi razlog zbog kojeg se funkcija *gensym* koristila je procesiranje Lisp koda. Primjerice[259], rekurzivna funkcija *search*[*l*; *p*; *h*; *u*] u listi *l* traži *y* za koji je *p*[*y*] = `T`; ako je pronađen, vrijednost funkcije je *h*[*y*]; ako nije, vrijednost funkcije je *u*. Funkcija je definirana izrazom

$$\begin{aligned} search[l; p; h; u] = [&null[l] \rightarrow u;\\ &p[car[l]] \rightarrow h[car[l]];\\ &\mathtt{T} \rightarrow search[cdr[l]; p; h; u]]. \end{aligned}$$

Tako definirana funkcija je spora jer se prilikom svakog poziva koriste argumenti *p*, *f* i *u* koji se ne mijenjaju. Moguće rješenje je zamjena funkcije *search* funkcijom *compsearch*[*l*; *p*; *h*; *u*] koja za svaku četvorku argumenata definira i poziva novu, funkciji *search* sličnu, ali efikasniju funkciju *searchf*. Definicija funkcije *searchf* ima oblik

257 Abrahams, *Application of LISP to checking mathematical proofs*, 1964., str.141.

258 Abrahams, *The proofchecker*, AIM-021, 1961., str. 9.

259 McCarthy et al., *LISP I manual*, 1960., str. 139.

$$searchf[g] = [null[g] \rightarrow u;\ pf \rightarrow hf;\ \mathtt{T} \rightarrow searchf[cdr[g]]],$$

pri čemu je *g* neki generirani simbol, primjerice `G00001`, a *pf* i *hf* su M-izrazi koji su po djelovanju jednaki *p*[*car*[*g*]] i *h*[*car*[*g*]], ali nisu primjene funkcija *p* i *h*, nego su konstruirani iz definicije funkcija *p* i *h*. Primjerice, ako je

$$h[y] = [y = \mathtt{NIL} \rightarrow \mathtt{NIL};\ \mathtt{T} \rightarrow \lambda[[r]; \mathrm{cons}[r; y]][cons[y; y]]]$$

onda na mjestu *hf* treba biti

$$[car[g] = \mathtt{NIL} \rightarrow \mathtt{NIL};\ \mathtt{T} \rightarrow \lambda[[r]; cons[r; car[g]]][cons[car[g]; car[g]]]].$$

Generirani simbol *g* je potreban zato što bi bilo koji drugi simbol mogao već biti korišten u tijelu funkcije *h*, kao što je u prethodnom primjeru korišten simbol *r*. Slično vrijedi i za *pf*.

Sistem održava listu *oblist* koja sadrži sve simbole koje je programer koristio. Generirani simboli se ne upisuju u *oblist*, po čemu se razlikuju od simbola koje je koristio programer, čak i ako programer namjerno koristi simbole s istim imenom.

16.10. SIMBOLI T, *T* I NIL

Neke promjene u Lispu 1.5 su sasvim praktično motivirane. Simbolički izraz `(QUOTE T)` korišten je tako često u programima, posebno u zadnjem uvjetu cond-izraza da se pribjeglo triku. Umjesto simbolom `T`, istina je predstavljena simbolom `*T*` a simbol `T` ima vrijednost `*T*`. Stoga, primjerice, izraz "čistog Lispa"

```
(COND,((EQ,Z,Y),X),
      ((QUOTE,T),Z))
```

nije korektan u Lispu 1.5. Umjesto toga, treba pisati

```
(COND ((EQ Z Y) X)
      (T Z)).
```

Vrijednost simbola `*T*` je `*T*`.

Analogno, neistina je predstavljena simbolom `NIL`. Vrijednost simbola `F` je `NIL`. Vrijednost simbola `NIL` je `NIL`.

Predikati se definiraju kao funkcije koje za vrijednosti imaju najviše dvije vrijednosti: `*T*` i `NIL`.

U funkcijama Lispa 1.5 koje kao argument očekuju `*T*` ili `NIL` svaka druga vrijednost se procesira jednako kao `*T*`.

U uvjetima uvjetnih izraza bilo koja vrijednost različita od `NIL` može se koristiti umjesto `*T*`.

Ove su odluke, osim što su skratile programe, istovremeno učinile jezik složenijim i manje lijepim. Autori priručnika navedene definicije su opisali frazom "Humpty-Dumpty semantics".[260]

Na sličan, pragmatičan, ali ne baš elegantan način su redefinirane i neke funkcije "čistog Lispa". Primjerice, funkcija *eq* u Lispu 1.5 je definirana i na ne-atomarnim izrazima i vrijednost te funkcije je istinita ako argumenti funkcije imaju istu reprezentaciju u memoriji računala.

16.11. ARITMETIKA

McCarthy je od samog početka želio da Lisp bude pogodan i za numerička izračunavanja.[261] Nažalost, implementacija aritmetike u stilu Woodwardsa i Jenkinsa nije bila dovoljno efikasna za praktične primjene pa Lisp 1.5 podržava aritmetiku slično Fortranu i drugim programskim jezicima. Primjerice, izraz 123 + 456 je u Lispu 1.5 predstavljen meta-izrazom

plus[123; 456]

260 Fraza se koristi u značenju "pokvaren i nepopravljiv".
McCarthy et al., *LISP 1.5 programmer's manual*, 1962., str. 22.

261 McCarthy, *Notes on improving Lisp*, 1986., str. LP-I.2.3.

koji se izračunava u vrijednost `579`, pri čemu su `123`, `456` i `579` simboli. Ti su simboli drugačiji od ostalih, pa se nazivaju "pseudo-atomskim simbolima".[262] Vrijednosti simbola `123`, `456` i `579` su redom sami simboli `123`, `456` i `579` i ne mogu im se pridruživati druge vrijednosti. Zato se meta--izrazi koji sadrže brojeve prevode u simboličke izraze bez simbola `QUOTE`. Primjerice, meta-izraz *plus*[`123`; `456`] prevodi se u simbolički izraz `(PLUS 123 456)`. Isti brojevi zapisani na drugačiji način, primjerice, `123` i `123.00` smatraju se istim simbolom.

Lisp 1.5 podržavao je brojeve u fiksnom i pomičnom zarezu. Definirano je dvadesetak osnovnih funkcija na brojevima.

16.12. POLJA

Polja se definiraju pseudo-funkcijom *array*. Primjerice, nakon definicije

array[`((ALPHA (7 10) LIST) (BETA (3 4 5) LIST))`]

alpha i *beta* su funkcije koje se ponašaju na specifičan način. Vrijednost alpha na kordinatama 5 i 6 se postavlja na vrijednost 8 izrazom

alpha[`SET`; `8`; `5`; `6`]

nakon čega izraz *alpha*[`5`; `6`] ima vrijednost `8`. Simbol `LIST` je obavezan, a iz priručnika je jasno samo da su autori razmišljali i o drugačijim poljima, u kojima bi se koristio neki drugi simbol.

16.13. LOGIKA

S obzirom na namjenu Lispa i priličnu zrelost jezika u vrijeme tiskanja *Priručnika*, podrška za logičke operacije, posebno za procesiranje logičkih izraza je začuđujuće slaba.

262 McCarthy et al., *Lisp 1.5 programmer's manual*, 1962., str. 14.

Ne postoji poseban logički tip. `NIL` označava neistinu a sve ostale vrijednosti označavaju istinu. Broj podržanih iskaznih operatora je malen: `OR`, `AND` i `NOT`, a postoje i verzije koje se primjenjuju na bitovima: `LOGAND`, `LOGOR`, uključujući i `LOGXOR` ("ekskluzivni ili"). Drugi se iskazni operatori mogu, doduše, lako definirati; primjerice,

$$implies[x; y] = \sim x \vee y,$$

ali isti argument vrijedi i za mnoge aritmetičke funkcije koje ipak jesu podržane u jeziku.

Jedine dvije funkcije koje mogu biti korisne pri procesiranju logičkih izraza (a ne samo logičkih vrijednosti) su već opisane `SUBST` i `GENSYM`.

McCarthy je puno kasnije savjetovao da se Lisp nadopuni podrškom za procesiranje logičkih izraza[263] kakva je postojala samo u eksperimentalnim dijalektima jezika.

263 McCarthy, *Beyond Lisp*, 2005., sl. 4.

17.

Matematička teorija izračunavanja

Tijekom 1961. i 1962. McCarthy je održao nekoliko izlaganja o matematičkoj teoriji izračunavanja na znanstvenim skupovima. Sredinom 1961. na *Western Joint Computer Conference* u Los Angelesu McCarthy je održao izlaganje *A basis for a mathematical theory of computation – preliminary report.* Iste godine, izlaganje s istom temom održao je na seminaru u IBM-ovoj školi u Blaricumu, Nizozemska. Izlaganje je objavljeno tek 1963. u knjizi *Computer Programming and Formal Systems*, ali nalazi se i u puno ranijem AIM-031 iz siječnja 1962. Čini se da se u članku iz 1963. ime LISP prvi puta u objavljenom dokumentu pojavljuje u obliku Lisp.[264] Drugo izlaganje, *Towards a mathematical science of computation* je održano na drugom kongresu IFIP-a u Munchenu, Njemačka i objavljeno u zborniku kongresa,[265] dostupno i u prijepisu iz 1996., objavljenom na McCarthjevim Web stranicama. Ta dva izlaganja se velikim dijelom preklapaju i dopunjavaju.

McCarthy predlaže razvoj "matematičke teorije izračunavanja". Jedan element teorije izračunavanja je razvoj univerzalnog programskog jezika. McCarthy tvrdi da formalizam koji opisuje, vrlo sličan poopćenom "čistom Lispu" nije pogodan kandidat za univerzalni programski jezik ali razlog koji navodi nije posebno jak.[266]

Matematička teorija izračunavanja bi uvelike koristila praksi. Programski jezici mogli bi se razvijati sistematičnije, a dokazivanje korektnosti programa moglo bi gotovo potpuno zamijeniti "debuggiranje".[267] Među ciljevima ma-

264 McCarthy, *A basis for a mathematical theory of computation*, 1963., str. 52

265 McCarthy, *Towards a mathematical science of computation*, 1962.

266 "We believe that this goal has been written off prematurely by a number of people. Our opinion of the present situation is that ALGOL is on the right track but mainly lacks the ability to describe different kinds of data, that COBOL is a step up a blind alley on account of its orientation towards English which is not well suited to the formal description of procedures, and that UNCOL is an exercise in group wishful thinking. The formalism for describing computations in this paper is not presented as a candidate for a universal programming language because it lacks a number of features, mainly syntactic, which are necessary for convenient use." McCarthy, *A basis for a mathematical theory of computation*, 1963., str. 34.

267 McCarthy, *Towards a mathematical science of computation*, 1996., str. 6.

tematičke teorije izračunavanja, nalazi se i predstavljanje algoritama simboličkim izrazima tako da se značajne promjene u ponašanju algoritama mogu predstaviti malim promjenama u simboličkim izrazima. Navodi čak i hrabru ideju da je svijest reprezentacija algoritma dostupna i pogodna procesiranju od strane samog algoritma.[268]

17.1. FUNKCIJE DEFINIRANE OSNOVNIM FUNKCIJAMA

U "čistom Lispu" je definirano pet osnovnih funkcija nad simboličkim izrazima, a iz njih se onda po određenim pravilima moglo definirati ostale funkcije. Ovdje, McCarthy dozvoljava bilo koji skup "osnovnih funkcija" F definiranih na proizvoljnom skupu, i onda definira funkcije koje su izračunljive na osnovi tih funkcija. Skup svih izračunljivih funkcija na osnovu F se označava C{F}. Sredstva kojima se mogu definirati funkcije ista su kao ona korištena u "čistom Lispu".

Primjerice, koristeći samo dvije osnovne funkcije na ne-negativnim brojevima, $succ(n) = n'$, primjerice, $succ(8) = 9$, i $eq(n_1, n_2) = \mathrm{T}$ ako i samo ako su n_1 i n_2 jednaki, moguće je definirati funkciju $pred(n)$ inverznu funkciji $succ$, zbrajanje, množenje, relaciju $\leq$ i mnoge druge funkcije. Sve su te funkcije prilično prirodno definirane, primjerice,

$$m \leq n = (m = 0) \vee (\sim (n = 0) \wedge (pred(m) \leq pred(n))).$$

17.2. FUNKCIONALI

Posebno, pravila za definiranje funkcija dozvoljavaju korištenje funkcija kao argumenata funkcija. Primjerice, funkcije *map* i *apply* su funkcionali. McCarthy uvodi hijerarhiju, podjelu funkcionala na "redove". *Funkcionali*

268 "Programs that are supposed to learn from experience change their behavior by changing the contents of the registers that represent the modifiable aspects of their behavior. From a certain point of view, having a convenient representation of one's behavior available for modification is what is meant by consciousness." McCarthy, *A basis for a mathematical theory of computation*, 1963., str. 34.

prvog reda su funkcije koje imaju argumente iz osnovne domene (u slučaju Lispa, simboličkih izraza). *Funkcionali drugog reda* imaju argumente iz osnovne domene ili funkcionale prvog reda itd.

Neki funkcionali se ne nalaze unutar ove hijerarhije. Primjerice, funkcije se mogu definirati tako da i same sebe prihvaćaju kao argumente. Nažalost, McCarthy je ostao na primjedbi i nije pokušao opisati sofisticiraniju klasifikaciju funkcionala.

17.3. UKLONJIVOST LABEL-IZRAZA

Funkcije koje same sebe prihvaćaju kao argumente su iskorištene za dokaz da label-izrazi nisu neophodni za definiranje rekurzivnih funkcija. Dokaz se temelji na zamjeni funkcije $f(x)$ apstraktnijom funkcijom $p(x, \varphi)$ koja se može definirati nerekurzivno, primijenjenoj na sebe samu, $p(x, p)$. McCarthy dokazuje da uz odgovarajuće definiranu funkciju p, za svaki x za koji je $f(x)$ definirana vrijedi

$$f(x) = p(x, p).$$

Pri izračunavanju $p(x, p)$ funkcija p poziva sebe samu, iako to pozivanje nije bilo potrebno pri definiranju p. Posebno, $f = \lambda((x), p(x, p))$. Ostaje pokazati kako definirati funkciju p. Primjerice, neka je

$$\begin{aligned} fact(n) = (&n = 0 \rightarrow 1,\\ &\mathrm{T} \rightarrow n \cdot fact(n - 1)). \end{aligned}$$

Funkcija *fact* može se definirati label-izrazom

$$\begin{aligned} fact = label(&f, \lambda((n), (n = 0 \rightarrow 1,\\ &\mathrm{T} \rightarrow n \cdot fact(n - 1)))). \end{aligned}$$

Funkcija p se definira kao funkcija koja izračunava faktorijele direktno izračunavajući samo osnovni slučaj, $n = 0$, a za ostale, zahtjevnije slučajeve, pozivajući neku drugu funkciju.

$$p(n, \varphi) = (n = 0 \to 1,$$
$$\mathrm{T} \to n \cdot \varphi(n - 1)).$$

Funkcija p nije rekurzivna funkcija. Posebno, ako je drugi argument funkcije p funkcija *fact* onda i p izračunava faktorijele, tj. $p(n, fact) = fact(n)$.

Primjena funkcije p na bilo koju drugu funkciju daje korektan rezultat za $n = 0$. Primjerice, $p(0, \sin) = fact(0)$.

Kako je p funkcija dvije varijable, da bi primjena funkcije na sebe samu imala smisla, valja malo promijeniti njenu definiciju.

$$p(n, \varphi) = (n = 0 \to 1, \quad \mathrm{T} \to n \cdot \varphi(n - 1, \varphi)). \tag{*}$$

Funkcija p još uvijek nije rekurzivna. Posebno, vrijedi

$$p(n, p) = (n = 0 \to 1, \quad \mathrm{T} \to n \cdot p(n - 1, p)). \tag{**}$$

Slijedi da

$$p(n, p) = fact(n).$$

Moglo bi se pomisliti da je $p(n, p)$, prema izrazu (**) ipak rekurzivna funkcija. No, jednakost (**) nije definicija funkcije p nego tek rezultat uvrštenja p umjesto φ, a defincija funkcije (*) nije rekurzivna, pa je funkciju p moguće definirati i lambda-izrazom

$$p = \lambda((n, \varphi), (n = 0 \to 1, \quad \mathrm{T} \to n \cdot \varphi(n - 1, \varphi))).$$

Funkcija p je ipak funkcija dvije varijable, i nije identična funkciji *fact*. No, funkcija

$$\lambda((n), p(n, p))$$

također nije rekurzivno definirana, može se definirati lambda-izrazom

$$fact_2 = \lambda((m), p(m, p))$$

i za svaki $n = 0, 1, \ldots$ vrijedi $fact_2(n) = fact(n)$.

Isti postupak McCarthy primjenjuje u općem slučaju. Neka je f rekurzivna funkcija definirana izrazom

$$f(x) = E(x, f)$$

gdje je E izraz u kojem se pojavljuju x i f. Funkcija f se može definirati label-izrazom što želimo izbjeći. Neka je p funkcija definirana izrazom

$$p(x,) = E(x, \lambda((y), \varphi(y,))).$$

Funkcija p nije rekurzivna pa

$$p = \lambda((x, \varphi), E(x, \lambda((y), \varphi(y, \varphi))).$$

Nadalje, vrijedi jednakost

$$p(x, p) = E(x, \lambda((y), p(y, p))).$$

Funkcija f_2 definira se izrazom

$$f_2(x) = p(x, p).$$

Funkcija f_2 nije rekurzivna pa se može definirati lambda izrazom

$$f_2 = \lambda((x), p(x, p)).$$

Tada za svaki x vrijedi

$$f_2(x) = p(x, p) = E(x, \lambda((y), p(y, p)) = E(x, f_2).$$

Funkcija $f_2(x)$, definirana isključivo lambda-izrazima je identična $f(x)$ za svaki $x = 0, 1, 2, \ldots$ Sama definicija je, raspiše li se, iznimno složena:

$$f = f_2 = \lambda((x), p(x, p)) =$$
$$= \lambda((x), \lambda((z, \varphi), E(z, \lambda((y), \varphi(y, \varphi)))$$
$$(x, \lambda((v, \psi), E(v, \lambda((w), \psi(w, \psi)))))$$

McCarthyjev izvod vrijedi, uz nebitne izmjene, i za funkcije više varijabli.

17.4. UVJETNI IZRAZI NISU FUNKCIJE

Moglo bi se misliti da je za dani prirodan broj moguće definirati funkciju $cond_n$ od $2n$ varijabli tako da

$$(p_1 \rightarrow e_1, ..., p_n \rightarrow e_n) = cond_n(p_1, ..., p_n, e_1, ..., e_n).$$

Po McCarthyju, to nije moguće jer "normalno" svi argumenti funkcije moraju biti definirani. Za uvjetne izraze, međutim, bitno je da neki od p_i i e_i nisu definirani. Primjerice, u definiciji faktorijela

$$n! = (n = 0 \rightarrow 1, \mathrm{T} \rightarrow n \cdot (n - 1)!)$$

podizraz $n \cdot (n - 1)!$ nije definiran za $n = 0$.

17.5. NEIZRAČUNLJIVE FUNKCIJE

Osim lambda- i label-izraza, McCarthy razvija nove izraze kojima se mogu definirati funkcije. Neka je e izraz koji može imati vrijednosti T i F, i koji sadrži varijablu x. Definiramo novu formu

$$\forall((x), e = \begin{cases} T & \text{ako } e \text{ ima vrijednost } T \text{ za sve } x \\ F & \text{ako } e \text{ ima vrijednost } F \text{ za bar jedan } x \\ \text{nedefiniran} & \text{inače} \end{cases}$$

Uz pomoć te forme mogu se definirati funkcije koje nisu S-funkcije, tj. ne mogu se definirati u Lispu zato što je $\forall((x), e) = F$ definirana čak i u slučaju da se za neki x izraz e izračunava do u beskonačnost.

17.6. VIŠEZNAČNE FUNKCIJE

Višeznačne funkcije imaju za svaki argument zadan skup dozvoljenih vrijednosti, a pri izračunavanju se odabire jedna od tih vrijednosti. Osnovna višeznačna funkcija je $amb(x,y)$ koja pri izračunavanju odabire jednu od vrijednosti x, y. Korištenjem funkcije amb mogu se definirati druge funkcije, primjerice

$$less(n) = amb(n - 1, less(n - 1)),$$

$$ult(n) \;=\; (n = 0 \rightarrow 0, T \rightarrow ult(less(n)))$$

Tada vrijedi

$$\forall((n), ult(n) = 0) \;=\; T.$$

17.7. REKURZIVNA DEFINICIJA SKUPA SIMBOLIČKIH IZRAZA

Neki rezultati opisani u izlaganjima su zanimljivi prvenstveno Lisp zajednici. U originalnom Lispu, simbolički izrazi su nizovi znakova; npr. `((A.B).C)`. Liste, poput `(A,B,C)` su bile pokrate, u ovom slučaju `(A.(B.(C.NIL))`. Ovdje, McCarthy definira simboličke izraze kao uređene n-torke nad nekim osnovnim skupom.

Neka je A skup simbola {`A`, `B`, `C`, `AA` ... }. Neka je $(a \cdot b)$ oznaka za uređen par. Tada je, primjerice, $((\texttt{A} \cdot \texttt{B}) \cdot \texttt{C})$ element skupa $(A \times A) \times A$. Skup svih simboličkih izraza nad A se označava $sexp(A)$. Tada vrijedi rekurzivna definicija skupa simboličkih izraza

$$sexp(A) = \{\Lambda\} \cup A \cup (sexp(A) \times sexp(A)).$$

Skup svih listi nad A je skup svih simboličkih izraza oblika

$$(a_1, ..., a_n) = (a_1 \cdot (a_2 \cdot ... (a_n \cdot \Lambda)))$$

gdje su a_1, ..., a_n. simbolički izrazi. Taj skup je rješenje jednadžbe

$$seq(A) = \{\Lambda\} \cup A \times seq(A).$$

Na tim skupovima se mogu definirati uobičajene Lisp funkcije poput *car*, *cdr*, *cons*, *atom*, *eq*.

17.8. REKURZIVNA INDUKCIJA

U uobičajenoj matematičkoj indukciji, želi li se dokazati da jednakost u kojoj se pojavljuje varijabla n vrijedi za svaki prirodan broj $n \geq 0$ dovoljno je dokazati da

1. jednakost vrijedi za $n = 1$;

2. ako jednakost vrijedi za $n - 1$ onda vrijedi i za n.

McCarthy uvodi analognu metodu dokazivanja jednakosti funkcija nad simboličkim izrazima koju naziva *rekurzivnom indukcijom*.

Neka su $g(x_1, \ldots, x_n)$ i $h(x_1, \ldots, x_n)$ funkcije nad simboličkim izrazima i treba dokazati da za sve $x_1, \ldots, x_n$ vrijedi

$$g(x_1, \ldots, x_n) = h(x_1, \ldots, x_n).$$

Tada je dovoljno dokazati da postoji i takav da

1. ako $null[x_i]$ onda $g(x_1, \ldots, x_i, \ldots, x_n) = h(x_1, \ldots, x_i, \ldots, x_n)$;

2. ako $g(x_1, \ldots, cdr[x_i], \ldots, x_n) = h(x_1, \ldots, cdr[x_i], \ldots, x_n)$ onda

$$g(x_1, \ldots, x_i, \ldots, x_n) = h(x_1, \ldots, x_i, \ldots, x_n).$$

Primjerice, neka je $x * y$ konkatenacija lista x i y. Primjerice,

```
(A B) * (B (C D)) = (A B B (C D)).
```

Funkcija $x * y$ je definirana s

$$x * y = [null[x] \rightarrow y; \mathrm{T} \rightarrow cons[car[x]; cdr[x] * y].$$

McCarthy dokazuje da je konkatenacija asocijativna operacija, tj. za sve simboličke izraze x, y i z vrijedi

$$[x * y] * z = x * [y * z].$$

Za lijevu stranu jednakosti vrijedi

$$\begin{aligned}[x * y] * z &= [null[x] \rightarrow y; \mathrm{T} \rightarrow cons[car[x]; cdr[x] * y]] * z \\ &= [null[x] \rightarrow y * z; \mathrm{T} \rightarrow cons[car[x]; cdr[x] * y] * z] \\ &= [null[x] \rightarrow y * z; \mathrm{T} \rightarrow cons[car[x]; [cdr[x] * y] * z]].\end{aligned}$$

Za desnu stranu jednakosti vrijedi

$$\begin{aligned}x * [y * z] = [&null[x] \rightarrow y * z ; \\ &\mathrm{T} \rightarrow cons[car[x]; cdr[x] * [y * z]]].\end{aligned}$$

Iz posljednje dvije jednakosti slijedi

1. ako $null[x]$ onda $[x * y] * z = x * [y * z]$;

2. ako $[cdr[x] * y] * z = cdr[x] * [y * z]$ onda $[x * y] * z = x * [y * z]$

što je i trebalo dokazati.

McCarthyjev student Lewis M. Norton je početkom 1962. dokazao dvadesetak lema i teorema o jednakostima izraza koji sadrže primjenu funkcije *subst*.[269]

17.9. APSTRAKTNA SINTAKSA PROGRAMSKIH JEZIKA

U obimnoj literaturi vlada konsensus da je McCarthy uveo ideju apstraktne sintakse programskih jezika.[270] Apstraktna sintaksa je definirana funkcijama koje (1) prepoznaju vrstu *terma*, simboličkih izraza legalnih u programskom jeziku, (2) analiziraju terme te (3) sintetiziraju terme.

269 Norton, *Some indentities concerning the function subst[x; y; z]*, AIM-037, 1962.

270 Bjorner, *Software engineering 2*, 2006., str. 87.

Primjerice, neka je zadan programski jezik u kojem postoje samo aritmetički izrazi, jedine operacije su zbrajanje i množenje dva argumenta. Apstraktna sintaksa tog programskog jezika opisana je sljedećim nizom funkcija.

1. Predikati koji prepoznaju vrstu terma: *isconstant*(*t*), *isvariable*(*t*), *issum*(*t*) i *isproduct*(*t*). Predikat *equalvariables*(*t*, *u*) koji provjerava jesu li dva simbola pojavljivanja iste varijable.

2. Funkcije koje izdvajaju članove zbroja i umnoška: *augment*(*t*) i *addend*(*t*), *multiplier*(*t*) i *multiplicand*(*t*).

3. Funkcije koje sintetiziraju izraze: *makesum*(*t*, *u*), *makeproduct*(*t*, *u*).

Ako se implementacija programskog jezika pri procesiranju terma služi samo navedenim funkcijama za procesiranje terma, onda na tu implementaciju nema utjecaja jesu li termi oblika $a + b$, $+ab$, `(PLUS A B)` ili čak Gödelovi brojevi.

Apstraktna sintaksa ima dvije prednosti pred tada popularnim *Backusovim normalnim formama*. Prvo, Backusove normalne forme su sintetičke; ne daju pravilo za rastavljanje programa na dijelove. McCarthyjeva apstraktna sintaksa je sintetička, ali i analitička, omogućuje rastavljanje programa na dijelove. Drugo, apstraktna sintaksa je nezavisna od konkretne sintakse programskog jezika.

McCarthy nije pokušao povezati Lisp s apstraktnom sintaksom, ali način na koji je Lisp definiran – funkcijama primijenjenima na simboličke izraze – čini ga *gotovo* jezikom apstraktne sintakse. McCarthyjeve definicije *evala*, u "čistom Lispu" i Lispu 1.5, ipak, sadrže složene M-izraze koji služe za izdvajanje dijelova S-izraza, što znači da nisu pisane imajući u vidu apstraktnu sintaksu.[271] Kasnije je McCarthy tvrdio da bi svaki jezik, uključujući i Lisp trebao sadržavati i funkcije koje podržavaju njegovu vlastitu apstraktnu sintaksu, te da bi jezici trebali imati nekoliko konkretnih sintaksi, za različite svrhe.[272]

271 "Lisp is close to its abstract syntax, but needs it anyway." McCarthy, *Beyond Lisp*, 2006., sl. 2.

272 McCarthy, *Guy Steele interviews John McCarthy, father of Lisp*, 2009.

Ideja apstraktne sintakse dopušta generalizacije o kojima McCarthy nije govorio: definiranje jezika složenijim strukturama podataka od samih simboličkih izraza, primjerice, strukturama lista.

17.10. SEMANTIKA

Da bi se definiralo semantiku opisanog aritmetičkog programskog jezika, tj. *značenje terma jezika*, po McCarthyju treba definirati

1. funkciju *valueconstant*(*t*) čija je vrijednost broj označen konstantom *t*,

2. funkciju *valuevariable*(*t, x*) čija je vrijednost jednaka broju pridruženom simbolu *t* u listi asocijacija *x*,

3. funkciju *makeconstant*(*n*) čija je term koji označava broj *n*.

Funkcija *valuevariable* je identična poznatoj funkciji *assoc*. Konačno, moguće je definirati *value*(*t*, *x*), značenje terma *t* za dani vektor stanja stroja *x*:

value(*t*, *x*) =
  (*isvariable(t)* → *valuevariable(t, x),*
   isconstant(t) → *valueconstant(t),*
   issum(t) → *value(addend(t), x)* + *value (augend(t), x),*
   isproduct(t) → *value(multiplier(t), x)* ·
                    value(multiplicand(*t*), *x*))

Funkcija *value*(*t*, *x*) u aritmetičkom jeziku je analogna funkciji *eval*(*e*, *a*) u Lispu.

Općenitije, *značenje programa* je određeno rezultatom primjene programa na vektor stanja stroja. Cijeli se programski jezik može definirati kao funkcija, primjerice

$$x' = algol(p, x)$$

gdje je *p* bilo koji program a *x*' i *x* su vektori stanja stroja, asocijacijske liste varijabli i vrijednosti.

McCarthyjeva razmišljanja su poopćenje koncepata poznatih iz Lispa. Stavovi stručne javnosti o značaju McCarthyjevih izlaganja su podijeljeni. Primjerice, dok ih Dines Bjorner smatra "velikim klasicima računalne znanosti"[273], Martin Davis je ostao "neuvjeren u teorijsku korisnost" bilo koje od McCarthyjevih ideja iznesenih u prvom izlaganju.[274]

273 Bjorner, *Software engineering 2*, 2006., str. 87.

274 Davis, *Review of John McCarthy, A basis for a mathematical theory ...,* 1968.

18.

Gilmoreov lispoliki jezik

U istoj knjizi kao i McCarthyjev članak *A basis for a mathematical theory of computation*, objavljen je i članak Paula C. Gilmorea *An abstract computer with Lisp-like machine language without label operator,* vjerojatno prvi "nezavisni" pokušaj unapređenja Lispa i jedini takav pokušaj u razdoblju koje je predmet ove knjige.

Gilmore razmatra općenitije definiran Lisp, određen klasom primitivnih funkcija F na nekom skupu U, "univerzumu", koji ne mora nužno biti skup svih simboličkih izraza. Nužno je, ipak, da elementi univerzuma, primitivne funkcije i operator *cond* mogu biti predstavljeni kao stringovi. S $C(F)$ označena je klasa svih funkcija koje se mogu definirati kao u McCarthyjevu članku *Rekurzivne funkcije nad simboličkim izrazima.*

18.1. UVJETNI IZRAZI

McCarthyjev Lisp ima, po Gilmoreu, dva nedostatka. Jedan je u definiciji uvjetnog izraza, $(p_1 \to e_1, \ldots, p_n \to e_n)$, gdje su $p_1, \ldots, p_n$ iskazni izrazi koji imaju "logičke vrijednosti", `T` ili `F`. Zbog toga, simboli `T` i `F` moraju biti elementi univerzuma U i bar neke od primitivnih funkcija moraju ih imati kao vrijednosti. Gilmore definira uvjetne izraze oblika

$$cond(x, y, z, w) = \begin{cases} z \text{ ako su } x \text{ i } y \text{ jednaki,} \\ w \text{ inače} \end{cases}$$

za koje `T` i `F` ne moraju biti elementi univerzuma. McCarthyjev uvjetni izraz se može definirati kompozicijom Gilmoreovih uvjetnih izraza. Primjerice, za $n = 2$ vrijedi

$$(p_1 \to e_1, p_2 \to e_2) = cond(p_1, \mathtt{T}, e_1, cond(p_2, \mathtt{T}, e_2, e)),$$

gdje je e bilo koji izraz.

18.2. QUOTE I LABEL

Drugi, veći problem je u operatorima *quote* i *label* koji ne pripadaju "prirodno" (engl. *naturally*) u teoriju izračunlji-

vih funkcija. McCarthyjevo uklanjanje label-izraza ne zadovoljava Gilmorea jer traži dodatnu pretpostavku da funkcije mogu prihvatiti same sebe kao argumente, a alternativne, nerekurzivne definicije su znatno složenije od polaznih.

Potreba za operatorima *quote* i *label* je, po Gilmoreu, posljedica neprepoznavanja da se simboli u Lispu koriste na bitno različite načine, kao:

1. elementi univerzuma U koji mogu biti argumenti funkcija ili operatora;

2. imena elemenata univerzuma U; primjerice, istina i laž mogu biti elementi U, a njihova imena su `T` i `F`;

3. imena funkcija ili operatora, primitivnih ili definiranih.

Gilmore uspijeva izbjeći operatore *quote* i *label* uvođenjem "*inicijalne naredbe učitavanja*" (engl. *initial load operation*), primjerice

$$!: (lambda,\ x,\ cond(x,\ 1,\ 1,\ x \cdot (x-1)!).$$

Smatrao je da se time "drastično" ne narušava jednostavnu strukturu Lispa. Inicijalna naredba je vrlo slična operatorima *cset* ili *csetq* iz Lispa 1.5. Kako Gilmore ne uspoređuje te dvije naredbe, vjerojatno u vrijeme pisanja članka nije bio upoznat s Lispom 1.5 o kome su prvi dokumenti publicirani tek 1962.

18.3. APSTRAKTNI STROJ

Semantika *lispolikog jezika* opisana je zamišljenim apstraktnim strojem s beskonačno memorijskih mjesta označenim nizovima znakova (bez ":"). U svakom memorijskom mjestu može se nalaziti niz znakova (opet, bez ":") proizvoljne duljine. Inicijalna naredba učitavanja $s_1{:}s_2$ izvršava se upisivanjem stringa s_2 u memorijsku lokaciju s_1. Kako s_1 i s_2 nisu smjeli sadržavati znak ":" to se naredbe učitavanja nisu mogle ugnježđavati. Kako sada svaki iz-

raz može biti spremljen na neku adresu, a po potrebi korišten, to u lispolikom jeziku nema potrebe za operatorima *label* i *quote*.

Gilmore je apstraktan stroj smatrao korisnim alatom koji pojednostavljuje implementaciju jezika, detaljno opisanu u članku, a istovremeno ukazuje na moguće probleme pri implementaciji u stvarnim računalima. Da bi pojednostavnio izlaganje, Gilmore pretpostavlja i da ne dolazi do "sukoba vezanih varijabli" (engl. *clash of bound variables*), u izrazima poput

(*lambda*, *x*, (*lambda*, *x*, *x*))

u kojem nije jasno je li posljednje pojavljivanje *x* vezano prvim ili drugim lambda. Taj se problem može izbjeći, primjerice, tako da izračunavanju lambda izraza prethodi preimenovanje varijabli, ili ako se ostavi programerima da brinu da do sukoba varijabli ne dolazi.

Prednost "lispolikog jezika" je u mogućnosti da se ne samo simboli, nego i simbolički izrazi mogu koristiti kao varijable, iako nije lako pronaći primjer u kojem bi takvo poopćenje bilo korisno. Nova sintaksa, ako i nije, kako Gilmore piše, "drastično različita", ne čini se nužnom.

Gilmoreov članak gotovo da nije imao odjeka u Lisp zajednici. No, citirao ga je Landin, 1965, u značajnom članku *A generalization of jumps and labels* kao rani primjer uvođenja imperativnih elemenata u "čisto funkcionalni" ili "referencijalno transparentni" jezik. Kako je Landin uveo nekoliko apstraktnih mašina, to je vjerojatno bio pod određenim Gilmoreovim utjecajem. Overheu[275] je razvio alternativnu apstraktnu mašinu koja zadržava prednosti Gilmoreove mašine – ukidanje *quote*, *label* i općenitiji *cond-izrazi*.

275 Overheu, *An abstract machine for symbolic computation*, 1966.

19.

Memoizacija

Otkriće memoizacije se obično pripisuje Richardu Bellmanu[276] a ideju je imenovao i popularizirao Donald Michie.[277] McCarthyjev memo *On efficient ways of evaluating certain recursive functions*[278] iz 1962. je vrlo rani primjer upotrebe memoizacije.

Naivna rekurzivna implementacija nekih algoritama je neefikasna jer se neke funkcije nepotrebno izračunavaju puno puta za iste vrijednosti argumenata. Najjednostavniji je primjer funkcija za izračunavanje Fibonaccijevih brojeva

$$\begin{aligned} \mathit{fibonacci}[n] = [&n \leq 1 \rightarrow 1; \\ &\mathsf{T} \rightarrow \mathit{fibonacci}[n-1] + \mathit{fibonacci}[n-2]]. \end{aligned}$$

Izračunavanje vrijednosti funkcije *fibonacci* za velike n je sporo, jer funkcija mnogo puta poziva samu sebe s istim argumentom i uvijek ponovo izračunava vrijednost funkcije. Primjerice, *fibonacci*$[n-2]$ se izračunava dva puta, *fibonacci*$[n-4]$ četiri puta, *fibonacci*$[n-6]$ osam puta itd.

Sličan problem je izračunavanje broja particija prirodnog broja. Primjerice, particije broja 5 su 5, 4+1, 3+1+1, 3+2, 2+2+1, 2+1+1+1 i 1+1+1+1+1.

McCarthy definira funkciju $q[m; n]$ koja izračunava broj particija broja m, takvih da niti jedan od pribrojnika nije veći od n. Funkcija je definirana izrazom

$$\begin{aligned} q[m; n] = [&m = 1 \vee n = 1 \rightarrow 1; \\ &m \leq n \rightarrow q[m; m-1] + 1; \\ &\mathsf{T} \rightarrow q[m-n; n] + q[m; n-1]]. \end{aligned}$$

Izračunavanje ove funkcije je neefikasno jer se funkcija q poziva puno puta s istim argumentima. Da bi to izbjegao, McCarthy definira funkciju tako da se rezultati izračunavanja spremaju u listu oblika $((m, n, q[m; n]), \ldots)$. Ako neka od definiranih funkcija očekuje tu listu kao argument, odgovarajući parametar je nazvan *known*.

276 Bellman, *Dynamic programming*, 1957.

277 Michie, *Memo functions and machine learning*, 1968.

278 McCarthy, *On efficient ways of evaluating certain recursive functions*, AIM-032, 1962.

Pomoćna funkcija *present*[*m*; *n*; *known*] je predikat čija je vrijednost T ako se element liste oblika (*m*,*n*,*qmn*) nalazi u listi known.

$$
\begin{aligned}
&present[m; n; known] = \\
&\quad \sim null[known] \wedge \\
&\quad [[eq[caar[known]; m] \wedge eq[cadar[known]; n]] \vee \\
&\quad\ present[m; n; cdr[known]]]
\end{aligned}
$$

Pomoćna funkcija *val*[*m*; *n*; *known*] je definirana samo za one vrijednosti m, n i known za koje je *present*[*m*; *n*; *known*] = T i tada ima vrijednost *qmn*.

$$
\begin{aligned}
&val[m; n; known] = \\
&\quad [eq[caar[known]; m] \wedge eq[cadar[known]; n] \rightarrow \\
&\qquad caddar[known]; \\
&\quad \mathtt{T} \rightarrow val[m; n; cdr[known]]]
\end{aligned}
$$

Pomoćna funkcija *prob*[*m*; *n*; *known*] ima za vrijednost listu *known*, ako treba proširenu, tako da uključuje (*m*,*n*,*qmn*).

$$
\begin{aligned}
&prob[m; n; known] = \\
&\quad [present[m; n; known] \rightarrow known \\
&\quad \mathtt{T} \rightarrow \lambda[[v]; cons[list[m; n; v]; known]] \\
&\qquad [[m = 1 \vee n = 1 \vee m = 0 \rightarrow 1; \\
&\qquad\ m \leq n \rightarrow val[m; n - 1; prob[m; n - 1; known]] + 1; \\
&\qquad\ \mathtt{T} \rightarrow \lambda[[p]; val[m - n; n; p] + val[m; n - 1; p]] \\
&\qquad\qquad [prob[m; n - 1; prob \\
&\quad [m - n; n; known]]]]]]
\end{aligned}
$$

Tehnika kojom se izbjegava višestruko pozivanje funkcije s istim argumentima je u posljednja dva retka. Da je McCarthy napisao

$$
\begin{aligned}
&\mathtt{T} \rightarrow val[m - n; n; prob[m - n; n; known] + \\
&\qquad val[m; n - 1; prob[m; n - 1; known]
\end{aligned}
$$

neki pozivi funkcije *prob*, uključujući *prob*[*m*; *n*; NIL], pozivali bi funkciju *prob* dva puta i bili bi jednako tako neefikasni kao pozivi funkcije *q*[*m*; *n*].

Općenito, želi li se izbjeći dvostruko izračunavanje izraza *e* u

$$g[\ldots, e, \ldots] + h[\ldots, e, \ldots]$$

može se koristiti

$$\lambda[[p]; g[\ldots, p, \ldots] + h[\ldots, p, \ldots]][e].$$

Liste asocijacija dobivene izračunavanjem

$$prob[m - n; n; known] \text{ i}$$

$$prob[m; n - 1; known]$$

nisu jednake. Ipak, u ovom kontekstu se obje mogu zamijeniti s

$$prob[m; n - 1; prob[m - n; n; known]].$$

Pri izračunavanju ovog izraza funkcija *prob* se poziva dva puta, ali se vrijednost dobivena izračunavanjem unutrašnjeg poziva funkcije koristi pri izračunavanju vanjskog poziva funkcije. Konačno,

$$q[m; n] = val[m; n; prob[m; n; \texttt{NIL}]].$$

Još uvijek, *q* se izračunava relativno sporo jer funkcije *present* i *val* pretražuju listu *known* linearno. McCarthy primjećuje da bi bilo efikasnije koristiti hash tablice.

McCarthy je zadao studentima da napišu funkciju koja transformira bilo koju S-funkciju u ekvivalentnu funkciju koja se ne izračunava više puta za isti argument i da rekuzivnom indukcijom dokažu korektnost funkcije.

McCarthyjev primjer je 1968. objavio D. W. Barron.[279]

279 Barron, *Recursive techniques in programming*, 1968., str. 18-19.

20.

Nova funkcija eval

McCarthyjev memo MIT AIM-034, *New eval function* napisan 1962.[280], najkasnije 6. ožujka[281], zakašnjeli je odgovor na nezadovoljavajuće procesiranje funkcija kao argumenata, kako u "teorijskom" tako i u "sistemskom" Lispu.

McCarthy uvodi nekoliko promjena u notaciji. Kao i u člancima o teoriji izračunavanja, u meta-izrazima koristi okrugle zagrade umjesto uglatih. Označava zagrade proizvoljnim brojem točaka ili zareza. Primjerice, `. . )` zatvara zagradu `( . .` i sve zagrade koje su otvorene između ta dva znaka. U uvjetnim izrazima piše *t* umjesto `T` pa se uvjetni izrazi prevode u oblik iz Lispa 1.5, tj. `(COND … (T …))` umjesto `(COND … ((QUOTE T) …))` kao u "čistom Lispu".

20.1. PROŠIRENI LISP

Za razliku od "čistog Lispa", "sistemski" ili "prošireni Lisp" sprema vrijednost simbola u listu svojstava. Izrazi "proširenog Lispa" ili *E-izrazi* se izgrađuju od atomskih simbola i *pokazivača* (orig. *full word pointer*), adresa u memoriji. Definicija E-izraza je:

1. Atomski simbol je E-izraz.

2. Pokazivač je E-izraz.

3. Ako su e_1 i e_2 E-izrazi onda je i $(e_1 . e_2)$ E-izraz.

Elementarne funkcije *car* i *cdr* nisu definirane za pokazivače.

Predikat *atom*[*x*] je istinit samo za simbole, a neistinit za ostale E-izraze.

Predikat *eq*[*x*; *y*] je istinit ako su *x* i *y* iste adrese u memoriji.

Predikat *fullword*[*x*] je istinit ako i samo ako *x* je pokazivač.

280 McCarthy, *A basis for a mathematical theory of computation*, AIM-031, 1962., str. 1.

281 Norton, *Some identities concerning function subst[x;y;z]*, AIM-037, 1962., str. 1.

Funkcija *value*[*x*] je definirana ako i samo ako je *x* simbol. Ako *x* ima vrijednost *v* zapisanu u listi svojstava, onda

$$value[x] = (\texttt{VALUE} \ . \ v).$$

20.2. NOVI EVAL

McCarthy je pokušao napisati funkciju *eval* koja bi omogućila korištenje funkcija kao argumenata i korektno se izračunavala bez obzira jesu li vrijednosti simbola zadani u listi asocijacija ili u listama svojstava.

Ako je *e* simbol, onda *eval* traži vrijednost *e* prvo u listi svojstava, a ako ga tamo ne nađe onda u asocijacijskoj listi *a*.

Ako *e* ima oblik $(fn \ arg_1 \ \ldots \ arg_n)$ onda se prvo definira *fnval* kao rezultat izračunavanja *fn*.

Ako je *fn* funkcija, onda je *fnval* adresa funkcije u mašinskom jeziku (a što se testira uvjetom *fullword*[*fnval*]), lambda- ili label-izraz. Tada *eval* primjenjuje (upotrebom *app1*) *fnval* na izračunate vrijednosti argumenata.

Ako je *fn* fexpr, onda je *fnval* oblika (*fx* . FEXPR) gdje je *fx* lambda- ili label-izraz. Tada *eval* primjenjuje *fx* na listu $(args \ a)$ gdje args = $(arg_1 \ \ldots \ arg_n)$.

Za primjenu funkcije na listu koristi se funkcija *app1*[*fnval*; *args*; *a*], slična funkciji *apply* iz *Lisp 1.5 programmer's manuala*. Za razliku od *apply*, funkcija *app1* ne izračunava vrijednost simbola na mjestu operatora automatski.

$$app1[eval[fn; a]; args; a] = apply[fn; args; a].$$

Evo kako je izgledao predloženi *eval*, koristeći uobičajene uglate zagrade i nešto kraća imena varijabli.

```
eval[e; a] =
  [atom[e] → search[e;
                   λ[[j]; [eq[car[j]; VALUE]]];
                   cadr;
                   λ[[ ]; assoc[e; a]]];
   t → prog[[fnval];
            fnval = eval[car[e]; a]
            return[[fullword[fnval] ∨
```

```
eq[car[fnval]; LAMBDA] ∨
eq[car[fnval]; LABEL] →
    app1[fnval;
            maplist[cdr[e];
                        λ[[j]; eval[car[j]; a]]]];
            a];
t→ app1[car[fnval];
            list[cdr[e]; a];
            a]]]]]
```

Nova funkcija *eval* nema posebna pravila za izračunavanje primjena osnovnih funkcija *car*, *cdr*, *cons*, *atom*, *eq* niti operatora `COND` ili `QUOTE` nego traži njihovu vrijednost jednako kao i za sve druge funkcije ili fexprove.

20.3. AUTONIMI LAMBDA I LABEL

Problem koji je preostao je izračunavanje samih lambda- i label-izraza, primjerice *eval*[`(LAMBDA (X) X)`; *a*].

McCarthyju se nije svidjela upotreba `QUOTE` ili `FUNCTION`. Umjesto toga, predlaže da lambda- i label-izrazi budu *autonimi*, izrazi koji se izračunavaju sami u sebe. Taj stav je konzistentan: lambda-izrazi i jesu uvedeni u "imperativni Lisp" da bi se spriječilo izračunavanje. McCarthy to postiže definirajući `LAMBDA` i `LABEL` kao fexprove. Primjerice, vrijednost `LAMBDA` se može definirati u listi svojstava simbola kao

```
((LAMBDA (E A) (CONS (QUOTE LAMBDA) E)) . FEXPR).
```

McCarthy je još uvijek ignorirao sada već dvije i pol godine star funarg problem. Implementatori Lispa 1.5 nisu prihvatili McCarthyjeve prijedloge.[282]

282 McCarthy et al., *LISP 1.5 Programmer's manual*, 1962., str. 70.

21.

Prve primjene Lispa

U ranom periodu, razvoj Lispa je apsorbirao toliko mnogo vremena i energije članova AI projekta da su u šali govorili da je Lisp jezik za pisanje Lispa.[283] Ubrzo, Lisp se ipak počeo i koristiti. Prvi programi su uvelike koristili mogućnost predstavljanja matematičkih i logičkih formula kao S-izraza.

Najraniji opisani Lisp program – ne računamo li primjere sastavljene radi opisivanja mogućnosti Lispa u ranijim memoima – je Rochesterov program za deriviranje izraza s jednom varijablom.[284] Program traži derivacije S-izraza koji se sastoje od varijable `x`, konstanti te operacija `times`, `plus`, `minus`. Prvo se umjesto `x` u cijelom izrazu supstituira `(plus,x,(Δ,x))`. Tako dobiveni izrazi se transformiraju po nizu od desetak pravila koje je Rochester izveo, uključujući simboličko dijeljenje s `(Δ,x)`, te simboličko traženje limesa. Tijekom cijelog procesiranja se izrazi pojednostavljuju, primjerice `(times,0,x)` se pojednostavljuje u `0`. Program je pisan u vrlo ranoj verziji Lispa. Sličan program je ubrzo razvio i Maling[285] čija funkcija *diff* je slična McCarthyjevoj.[286]

Tim Hart je tijekom 1961. implementirao funkciju *simplify*, koja je pojednostavljivala 45 slučajeva algebarskih izraza.[287]

Rochester, Goldberg, Rubenstein, Edwards i Markstein su proučavali svojstva strujnih krugova i mreža i napisali niz programa. Lisp su smatrali dobrim izborom zbog svrhe za koju je dizajniran: izražavanje matematičkih i logičkih algoritama i procesiranje simboličkih izraza.[288] McCarthy i drugi su do travnja 1959. uočili[289] da u tom projektu postoji potreba za organiziranjem podataka u matrice i efikasno invertiranje matrica.

283 Osobna komunikacija s Abrahamsom, 2014.

284 Rochester, AIM-005, 1958.

285 Maling, *The LISP differentiation demonstration program,* AIM-010, 1959.

286 McCarthy, *Recursive functions...*, *CACM*, 1960.

287 Hart, *Simplify*, AIM-027, 1961.

288 Rochester i dr., *Machine manipulation of algebraic expressions*, RLE QPR 055, 1959., str. 132.

289 McCarthy i dr., *Artificial intelligence*, RLE QPR 053, 1959., str. 123.

MIT-jev program za šah, iako su u njegovom razvoju sudjelovali McCarthy i Abrahams, pisan je u FORTRAN-u.[290]

McCarthy je opisao[291,292,293] program koji Wangovim algoritmom[294] provjerava je li "sekvent" u iskaznom računu tautologija. Formule su definirane na sljedeći način: Simboli *P*, *Q*, *R*, *M*, *N* itd. su formule. Ako su φ i ψ formule onda su $\sim\varphi$, $\varphi \,\&\, \psi$, $\varphi \vee \psi$, $\supset \psi$, $\varphi \equiv$ formule. "Sekventi" su izrazi oblika

$$\varphi_1, \ldots, \varphi_n \rightarrow \psi_1, \ldots, \psi_m$$

i istiniti su ako za svaku vrijednost iskaznih varijabli vrijedi: ako su sve formule $\varphi_1, \ldots, \varphi_n$ istinite, tada je bar jedna od formula $\psi_1, \ldots, \psi_m$ istinita. Iskazne formule se prevode u S-izraze kao inače. Sekvent $\varphi_1, \ldots, \varphi_n \rightarrow \psi_1, \ldots, \psi_m$ se prevodi u simbolički izraz

$$\texttt{(ARROW,(}\varphi_1\texttt{*,...,}\varphi_n\texttt{*),(}\psi_1\texttt{*,...,}\psi_m\texttt{*))}.$$

Tijekom 1960. ili 1961. S. R. Petrick napisao je program za pojednostavljivanje iskaznih logičkih izraza u "čistom Lispu".[295]

Tijekom 1960. Anthony Valiant Phillips napisao je program koji odgovara na pitanja na engleskom jeziku na temelju zadanog teksta, nadajući se da bi program mogao razumjeti tekst na nivou šestogodišnjeg djeteta. Primjerice, na temelju teksta

```
((AT SCHOOL JOHNNY MEETS THE TEACHER)
(THE TEACHER READS BOOKS IN THE CLASSROOM))
```

program je na pitanje

290 Norberg, *Paul W. Abrahams interview,* 2006., str. 10.

291 McCarthy, *The Wang algorithm for propositional calculus …*, *AIM-014*, 1959.

292 McCarthy et al., *LISP I. programmer's manual*, 1960., str. 25-36.

293 McCarthy et al., *LISP 1.5 programmer's manual*, 1962., str. 44-55.

294 Wang, *Toward mechanical mathematics*, 1960.

295 Petrick, *Use of list processing language in programming simplification procedures*, 1961.

```
(WHERE DOES THE TEACHER READ BOOKS)
```

odgovarao s

```
(((IN THE CLASSROOM)
(THE TEACHER READS BOOKS IN THE CLASSROOM))).
```

Iako program nije dosegao zadani nivo, Phillips je cilj smatrao dostižnim, a bio je i zadovoljan Lispom i smatrao ga bitnim za uspjeh projekta. [296]

Slagle je tijekom studija na MIT-ju razvio program SAINT za rješavanje neodređenih integrala. Tema je odabrana, između ostalog, jer uključuje "manipulaciju simboličkim izrazima" koja će vjerojatno biti temelj budućeg rješavanja problema.[297] Kvalitet programa je sam ocijenio s "*approximately a level of a good college freshman*."[298]

Student Louis Hodes je napisao jednostavan program za "pattern recognition" pod vodstvom Minskog.[299]

McCarthy je koristio Lisp za rješavanje jednostavnih problema teorije grafova.[300]

Paul W. Abrahams je razvijao program PROOFCHECKER[301, 302] kojim je provjerio dokaze u II. poglavlju Russell i Whiteheadove knjige *Principia Mathematica*. Svi teoremi iz tog poglavlja pripadaju iskaznom računu. Prevođenje, čak i iznimno formalnog jezika u oblik koji bi računalo razumjelo je predstavljalo problem. Formule iz Principia Mathematica su prevođene u S-izraze, primjerice

$$p \vee q \rightarrow q \vee p$$

se prevodi u

296 Phillips, *A question-answering routine*, AIM-016, 1960.

297 Slagle, *A heuristic program* ..., 1961., str. 10.

298 Slagle, *A heuristic program* ..., 1961., str. 7.

299 Hodes, *Some results from pattern recognition program using LISP*, AIM-018, 1960.

300 McCarthy, *Puzzle solving program in LISP*, AIM-020, 1960.

301 Abrahams, *The proofchecker*, AIM-021, 1961.

302 Abrahams, *Application of LISP to checking mathematical proofs*, 1964.

```
(IMPLIES (OR P Q) (OR Q P)).
```

Već pri prevođenju otkrivene su neke greške u Principia Mathematica. Posebno se zahtjevna pokazala operacija supstitucije, primjerice $p \rightarrow q$ za $\sim p \vee q$ i obratno. Provjere koraka u dokazu su se često svodile na konstrukciju simboličkog izraza i primjenu funkcije *eval*. PROOFCHECKER je kreirao i održavao listu prethodno provjerenih teorema, a na početku izvršavanja programa ta je lista sadržavala samo aksiome. Svaki je teorem bio lista od tri elementa: ime teorema, lista "zamjenjivih" varijabli i sam iskaz teorema. Primjerice, teorem

$$p \wedge q \rightarrow q$$

nazvan `CONJ` je bio zapisan u listi teorema u obliku

```
(CONJ (P Q) (IMPLIES (AND P Q) Q)).
```

Koraci u dokazu su pohranjeni u obliku liste od tri elementa: broj koraka, tekst dokazane formule i lista brojeva formula koji su korišteni u dokazu. Program je često trebao nove varijable koje nisu upotrebljene u prethodnim formulama i za tu svrhu je korištena Lisp funkcija *gensym*.

704 electronic data-processing machine – manual of operation. (1955). New York: International Business Machines Corporation.

Abrahams, P. (1961). *Character-handling facilities in the Lisp system, AIM-022.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Abrahams, P. (1961). *The proofchecker, AIM-021.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Abrahams, P. (1964). Application of LISP to checking mathematical proofs. U E. C. Berkeley, & D. G. Bobrow (Ur.), *The programming language LISP: its operation and applications* (str. 137-159). Cambridge, Massachusetts, USA: The M.I.T. Press.

Abrahams, P. W. (1968). Symbol manipulation languages. *Advances in computers, 9*, 51-111.

Abrahams, P. W., Barnett, J. A., Book, E., Firth, D., Kameny, S. L., Weissman, C., i dr. (1966). The LISP 2 Programming language and the system. *AFIPS '66 (Fall) Proceedings of the November 7-10 1966, fall joint computer conference* (str. 661-676). New York: ACM.

Andresen, S. L. (listopad 2002). John McCarthy: Father of Lisp. *IEEE Intelligent Systems, 17*(5), 84-85.

Backus, J. W., Beeber, R. J., Best, S., Goldberg, H., Herrick, H. L., Hughes, R. A., i dr. (1956). *The FORTRAN automatic coding system for the IBM 704 EDPM.* New York: International Business Machines Corporation.

Barendregt, H., & Barendsen, E. (2000). *Introduction to Lambda calculus.* Preuzeto 5. lipanj 2014 iz The Programming Logic Group in Göteborg: http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf

Bellman, R. (1957). *Dynamic Programming.* Princeton: Princeton University Press.

Berkeley, E. C., & Bobrow, D. G. (Ur.). (1964). *The programming language LISP: its operations and applications* (1st izd.). USA: Information International Inc.

Bjørner, D. (2006). *Software engineering 2 – specification of systems and languages.* Berlin, Germany: Springer Verlag.

Bobrow, D. A. (1963). *METEOR: A LISP interpreter for string transformations, AIM-051.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Böhm, C. (1954). Calculatrices digitales. Du déchiffrage des formules logico-mathématiques par la machine même dans la conception du programme. *Annali di Matematica Pura ed Applicata, 37*(4), 1-51.

Brayton, R. (1961). *LISP error stops as of may 10, 1961, AIM-025.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Brayton, R. (1961). *Trace-printing for compiled programs, AIM-023.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Burger, A. (n.d.). *PicoLisp reference.* Preuzeto 23. veljača 2014 iz http://software-lab.de/doc/ref.html

Burger, A. (n.d.). *The PicoLisp Reference.* Preuzeto 5. lipanj 2014 iz Software Lab. Alexander Burger: http://software-lab.de/doc/ref.html

Church, A. (1941). *The calculi of lambda conversion.* Princeton, New Jersey, USA: Princeton University Press.

Conrad, T., Conrad, P., Hartline, P., Schlessinger, J., Yates, B., Evans, R., i dr. (1961). *Handbook of LISP functions.* Technical report, RIAS, Baltimore.

Davis, M. (ožujak 1968). Review of John McCarthy, A basis for a mathematical theory of computation, preliminary report. *The Journal of Symbolic Logic, 33*(1), 117.

Davis, M. (ožujak 1968). Review of John McCarthy, Recursive functions of symbolic expressions and their computation by machine. *The Journal of Symbolic Logic, 33*(1), 117.

Dezani-Ciancaglini, M., & Hindley, J. R. (2008). Lambda Calculus. U B. Wah (Ur.), *Wiley encyclopedia of computer science and engineering.* John Wiley & Sons, Inc.

Dijkstra, E. W. (1974). *Trip report, Edinburgh and Newcastle.*

Edwards, D. (1963). *Secondary storage in Lisp, AIM-063.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Edwards, D. J. (1960). *Lisp II garbage collector, AIM-019.* RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

Edwards, D. J., & Hart, T. P. (1961). *The alpha-beta heuristics, AIM-030.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Evans, T. G. (1962). *A heuristic program to solve geometric analogy problems, AIM-046.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Faase, F. J. (2006). *The origin of CAR and CDR in LISP*. Preuzeto 3. lipanj 2014 iz I write, therefore I am: http://www.iwriteiam.nl/HaCAR_CDR.html

Fox, P. A. (7-8. lipanj 2005). An interview with Phyllis A. Fox. (T. Haigh, Ispitivač) Philadelphia, PA.

Gelernter, H., Hansen, J. R., & Gerberich, C. L. (1960). A FORTRAN-compiled list-processing language. *Journal of the ACM (JACM), 7*(2), 87-101.

Giles, H. A. (Ur.). (1889). *Chuang Tzu − Mystic, moralist and social reformer.* (H. A. Giles, Prev.) London: Bernard Quaritch.

Gilmore, P. C. (1963). An abstract computer with a Lisp-like machine language without a label operator. U P. Braffort, & D. Hirschberg (Ur.), *Computer programming and formal systems* (str. 71-86). Amsterdam: North Holland.

Graham, P. (2002). *Roots of Lisp.* Preuzeto 5. lipanj 2014 iz Paul Graham: http://lib.store.yahoo.net/lib/paulgraham/jmc.ps

Hart, T. (1961). *Simplify, AIM-027.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Hart, T., & Levin, M. (1962). *The new compiler, AIM-039.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Henneman, W. (1964). An auxiliary language for more natural expressions − the A-language. U E. C. Berkeley, & D. G. Bobrow (Ur.), *The programming language LISP: its operation and applications* (str. 239-248). Cambridge, Massachusetts, USA: The M.I.T. Press.

Hodes, L. (1960). *Programs with common sense, AIM-018.* RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

Jordan, C. R. (1973). A note on LISP universal S-functions. *The Computer Journal, 16*(2), 124-125.

Kay, A. (prosinac/siječanj 2004-2005). A conversation with Alan Kay. *Queue*, 20-30.

Kay, A. C. (1996). The early history of Smalltalk. U *History of programming languages II* (str. 511-598). New York, NY, USA: ACM.

Landin, P. J. (ožujak 1966). The next 700 programming languages. *Communications of the ACM, 9*(3), 157-166.

Levin, M. (1961). *Arithmetic in LISP 1.5, AIM-024.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Levin, M. (1961). *Errorset, AIM-026.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Levin, M. (1961). *LISP 1.5 Programmer's manual, AIM-028.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Levin, M. (1963). *Primitive recursion, AIM-055.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Levy, S. (2010). *Hackers* (1st izd.). Gravenstein Highway North, Sebastopol, CA, USA: O'Reilly Media, Inc.

Lord, T. (25. listopad 2011). *John McCarthy has passed*. Preuzeto 31. svibanj 2014 iz Lambda The Ultimate: http://lambda-the-ultimate.org/node/4387

Mac Lane, S. (1988). Group extensions for 45 years. *The mathematical intelligencer, 10*(2).

Maling, K. (1959). *The LISP differentiation demonstration program, AIM-010.* M.I.T., Artificial Intelligence Project.

Maling, K. (1959). *The Maling-Silver read program, MIT AIM-013.*

McCarthy, J. (1956). The Inversion of function defined by Turing machines. U C. E. Shannon, & J. McCarthy (Ur.), *Automata studies* (Svez. 267). Princeton, New Jersey: Princeton University Press.

McCarthy, J. (1957). *A proposal for a compiler, CC-56.* Massachusetts Institute of Technology, Computation Center.

McCarthy, J. (1958). *A revised definition of maplist, AIM-002.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1958). *An algebraic language for the manipulation of symbolic expressions, AIM-001.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1958). *Notes on the compiler, AIM-007.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1958). *Revisions of the language, AIM-003.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1958). *Revisions of the language, AIM-004.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (10. lipanj 1958). Some proposals for the Volume 2 (V2) language. *letter*. Cambridge, Massachusetts, USA.

McCarthy, J. (1959). LISP: a programming system for symbolic manipulations. *Proceedings ACM '59 Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery*. New York: ACM.

McCarthy, J. (August 1959). On conditional expressions and recursive functions. *Communications of the ACM, 2*(8), 2-3.

McCarthy, J. (1959). *Programs in LISP, AIM-012.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1959). Programs with common sense. *Symposium on Mechanization of Thought Processes, Teddington, England, 1958. I*, str. 75-92. London: Her Majesy's Stationary Office.

McCarthy, J. (1959). Recursive functions of symbolic expressions and their computation by machine. *Research Laboratory of Electronics, Quarterly progress report, 053*, 124-152. Cambridge, MA, Massachusetts, USA: MIT.

McCarthy, J. (1959). *Recursive functions of symbolic expressions and their computation by machine, AIM-008.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1959). *Recursive functions of symbolic expressions and their computation by machine, AIM-011.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1959). The LISP programming system. *Research Laboratory of Electronics, Quarterly progress report, 053*, 122. Cambridge, MA, Massachusetts, USA: MIT.

McCarthy, J. (1960). *Programs with common sense, AIM-017.* RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

McCarthy, J. (1960). *Puzzle solving program in LISP, AIM-020.* RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine. Part I. *Communications of the ACM, 3*(4), 184-95.

McCarthy, J. (1960). The LISP programming system. *Research Laboratory of Electronics, Quarterly progress report, 056*, 158-159. Cambridge, MA, Massachusetts, USA: MIT.

McCarthy, J. (1961). A basis for a mathematical theory of computation – preliminary report. U P. Braffort, & D. Hirschberg (Ur.), *Papers presented at the May 9-11, 1961, western joint IRE-AIEE-ACM computer conference* (str. 225-238). New York: ACM.

McCarthy, J. (1962). *A basis for a mathematical theory of computation, AIM-031.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1962). *New eval function, AIM-034.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1962). *On efficient ways of evaluating certain recursive functions, AIM-032.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

McCarthy, J. (1962). Towards a mathematical science of computation. U C. M. Popplewell (Ur.), *Proceedings of IFIP Congress 62, Munich, Germany, August 27 − September 1, 1962* (str. 21-28). Amsterdam: North-Holland.

McCarthy, J. (1963). A basis for a mathematical theory of computation. U P. Braffort, & D. Hirschberg (Ur.), *Computer programming and formal systems* (str. 33-70). Amsterdam: North Holland.

McCarthy, J. (1963). *Situations, actions and causal laws.* Stanford University, Stanford Artificial Intelligence Project. Springfield: Stanford University.

McCarthy, J. (1980). LISP − notes on its past and future. *LFP '80 Proceedings of the 1980 ACM conference on LISP and functional programming* (str. .5 − viii). New York: ACM.

McCarthy, J. (1981). History of Lisp. U R. L. Wexelblat (Ur.), *ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978* (str. 173-198). New York: Academic Press, Inc.

McCarthy, J. (lipanj-srpanj 1987). Notes on improving Lisp. *ACM SIGPLAN Lisp pointers, 1*(2), 3-4.

McCarthy, J. (1988). *The logic and philosophy of artificial intelligence.* Preuzeto 31. svibanj 2014 iz Kyoto Prize: http://emuseum.kyotoprize.org/sites/default/files/4kA_e.pdf

McCarthy, J. (2. ožujak 1989). An interview with John McCarthy. (W. Aspray, Ispitivač) Palo Alto, CA, USA: University of Minnesota.

McCarthy, J. (1995). *Recursive functions of symbolic expressions and their computation by machine.* Preuzeto 6. lipanj 2014 iz John McCarthy's home site: http://www-formal.stanford.edu/jmc/recursive.pdf

McCarthy, J. (1996). *Towards a mathematical science of computation.* (C. M. Popplewell, Ur.) Preuzeto 7. lipanj 2014 iz John McCarthy's Home Page: http://www-formal.stanford.edu/jmc/towards.pdf

McCarthy, J. (2002). John McCarthy: father of AI. 84-85. IEEE.

McCarthy, J. (19. lipanj 2002). *Some Lisp history and some programming languages ideas.* Preuzeto 7. lipanj 2014 iz John McCarthy's Home Page: http://www-formal.stanford.edu/jmc/slides/lisp/lisp-sli.dvi

McCarthy, J. (22. lipanj 2005). *Beyond Lisp*. Preuzeto 7. lipanj 2014 iz John McCarthy's Home Page: http://www-formal.stanford.edu/jmc/slides/lisp/beyond-sli.pdf

McCarthy, J. (studeni 2006). *Dartmouth and beyond*. Preuzeto 1. lipanj 2014 iz John McCarthy's Home page: http://www-formal.stanford.edu/jmc/slides/dartmouth/dartmouth-sli/

McCarthy, J. (12. studeni 2007). *What is artificial intelligence?* Preuzeto 31. svibanj 2014 iz John McCarthy's Home Page: http://www-formal.stanford.edu/jmc/whatisai.pdf

McCarthy, J. (2008). The philosophy of AI and AI of philosophy. U P. Adriaans, & J. van Benthem (Ur.), *Handbook of the philosophy of science – volume 8: philosophy of information* (1st izd., Svez. 8). North Holland.

McCarthy, J. (1. May 2009). *Guy Steele interviews John McCarthy*. Preuzeto 4. kolovoz 2013 iz InfoQ: http://www.infoq.com/interviews/Steele-Interviews-John-McCarthy

McCarthy, J. (n.d.). *Examples of proofs by recursion, AIM-015*. RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

McCarthy, J. (n.d.). *Progress and its sustainability*. Preuzeto 5. ožujak 2014 iz John McCarthy's Home Page: http://www-formal.stanford.edu/pub/jmc/progress/

McCarthy, J. (n.d.). *The Wang algorithm for the propositional calculus programmed in LISP, AIM-014*. RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

McCarthy, J., & Minski, M. (1959). Artificial intelligence. *Research Laboratory of Electronics, Quarterly progress report, 052*, 129. Cambridge, MA, Massachusetts, USA: MIT.

McCarthy, J., & Minski, M. (1959). Artificial intelligence. *Research Laboratory of Electronics, Quarterly progress report, 053*, 122-124. Cambridge, MA, Massachusetts, USA: MIT.

McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P., & Levin, M. I. (1962). *Lisp 1.5 Programmer's manual*. Cambridge, Massachusetts, USA: The M.I.T. Press.

McCarthy, J., Brayton, R., Edwards, D., Fox, P., Hodes, L., Luckham, D., i dr. (1960). *Lisp I programmer's manual*. Cambridge, Massachusetts, USA: manuscript.

McCarthy, J., Brayton, R., Edwards, D., Fox, P., Hodes, L., Luckham, D., i dr. (1960). *Lisp preliminary programmer's manual – draft*. Cambridge, Massachusetts, USA: manuscript.

McCarthy, J., Minsky, M., Rochester, N., & Shannon, C. E. (31. kolovoz 1955). A proposal for the Dartmouth summer research project on artificial intelligence.

McCarthy, J., Rochester, N., Maling, K., & Russell, S. (1959). *LISP programmer's manual*. MIT, Artificial Intelligence Project, Cambridge.

McCarthy, S. (25. ožujak 2012). *What your dentist doesn't want you to know*. Preuzeto 31. svibanj 2014 iz Celebration of John McCarthy: http://cs.stanford.edu/jmc

McJones, P. (n.d.). *History of LISP*. Preuzeto 6. lipanj 2014 iz Software Preservation Group: http://www.softwarepreservation.org/projects/LISP/

Meyer, B. (7. studeni 2011). *John McCarthy*. Preuzeto 5. lipanj 2014 iz Bertrand Meyer's technology+ blog: http://bertrandmeyer.com/2011/11/07/john-mccarthy/

Michie, D. (1968). Memo functions and machine learing. *Nature, 218*, 19-22.

Minski, M. (1963). Artificial intelligence. *Research Laboratory of Electronics, Quarterly progress report, 068*, 159-161. Cambridge, MA, Massachusetts, USA: MIT.

Minsky, M. (1983). Introduction to the COMTEX microfiche edition of the early MIT AI memos. *AI Magazine, 4*(1), 19-22.

Moore, C., & Martens, S. (2011). *Nature of computation*. New York: Oxford University Press.

Morris, A. H. (1966). Models for mathematical systems. *SYMSAC '66 Proceedings of the first ACM symposium on Symbolic and algebraic manipulation* (str. 1501-1524). New York: ACM.

N., N. (1962). *LAP (LISP assembly program), AIM-035*. Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Newell, A., & Shaw, J. C. (1957). Programming the Logic Theory Machine. *IRE-AIEE-ACM '57 (Western) Papers presented at the February 26-28, 1957, western joint computer conference: Techniques for reliability* (str. 230-240). New York: ACM.

Newell, A., & Simon, H. A. (1956). *The logic theory machine – a complex information processing system*. Santa Monica: Rand Corporation.

Nilsson, N. (2012). *John McCarthy 1927-2011*. Washington: National academy of science.

Norberg, A., & Abrahams, P. W. (15-17. listopad 2006). Paul W. Abrahams interview: October, 15, 16 and 17, 2007; Deerfield, MA. *Proceeding ACM Oral History interviews*. New York: ACM.

Norton, L. M. (1962). *Some identities concerning the function subst[x; y; z], AIM-037*. Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Overheu. (lipanj 1966). An abstract machine for symbolic computation. *Journal of the ACM, 13*(3), 444-468.

Perlis, A. J. (1978). The American side of the development of Algol. *ACM SIGPLAN Notices – Special issue: History of programming languages conference. 13*, str. 3-14. New York: ACM.

Petrick, S. R. (1961). Use of a list processing language in programming simplification procedures. *1st and 2nd Annual Symposium on Switching Circuit Theory and Logical Design*, (str. 18-24). Detroit, MI.

Phillips, A. V. (1960). *Examples of proofs by recursion, AIM-016.* RLE and MIT Computation Center, Artificial Intelligence Project. M.I.T.

Pitman, K. M. (1980). Special forms in LISP. *LFP '80 Proceedings of the 1980 ACM conference on LISP and functional programming* (str. 179-187). New York: ACM.

Raphael, B. (1961). *Introduction to calculus of knowledge, AIM-029.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Raphael, B. (1963). *Computer representation of semantic information, AIM-049.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Reynolds, J. C. (1972). Definitional interpreters for higher-order programming languages. *ACM '72 Proceedings of the ACM annual conference. 2*, str. 717-740. New York: ACM.

Robnett, R. A. (1963). *Suggested conventions for LISP time-sharing system, AIM-050.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Rochester, N. (1958). *AIM-005.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Rochester, N., Goldberg, S. H., & Edwards, D. J. (1959). Machine manipulation of algebraic expressions. *Research Laboratory of Electronics, Quarterly progress report, 055*, 132-134. Cambridge, MA, Massachusetts, USA: MIT.

Russell, S. (1958). *Writing and debuging programs, AIM-006.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Russell, S. (1959). *Explanation of BIG "P" as or march 20, AIM-009.* Internal memoranda, M.I.T., Artificial Intelligence Project, Cambridge, MA.

Russell, S. (25. ožujak 2012). *Adventures and pioneering with John.* Preuzeto 31. svibanj 2014 iz Celebration of John McCarthy: http://cs.stanford.edu/jmc

Saunders, R. (1964). LISP – on the programming system. U E. C. Berkeley, & D. G. Bobrow (Ur.), *The programming language LISP: its operation and applications* (str. 50-72). Cambridge, Massachusetts, USA: The M.I.T. Press.

Sebesta, R. W. (2012). *Concepts of programming languages* (10th izd.). Boston: Pearson Education Inc.

Shasha, D., & Lazere, C. (1995). *Out of their minds: the lives and discoveries of 15 great computer scientists.* New York: Copernicus/ An Imprint of Springer-Verlag.

Slagle, J. R. (1959). Formal integration on a digital computer. *ACM '59 Preprints of papers presented at the 14th national meeting of the Association for Computing Machinery* (str. 36-1 – 36-2). New York: ACM.

Slagle, J. R. (1961). *A heuristic program that solves symbolic integration problems in freshman calculus, Symbolic Automatic Integrator (SAINT).* Cambridge, MA, USA: Massachusetts Institute of Technology.

Stoyan, H. (prosinac 1979). Lisp history. (P. Greussay, & J. Laubsch, Ur.) *LISP bulletin*(3), 42-53.

Stoyan, H. (1980). *LISP, Anwendungsgebiete, Grundbegriffe, Geschichte.* Berlin: Akademie-Verlag.

Stoyan, H. (1984). Early history of LISP (1956-1959).

Stoyan, H. (1984). Early LISP History (1956-1959). *1984 ACM Symposium on LISP and functional programming* (str. 229-310). ACM.

Stoyan, H. (1988). *Programmiermethoden der Künstlichen Inteligenz* (1st izd.). Berlin Heidelberg, Germany: Springer.

Stoyan, H. (1992). List processing. U A. Kent, & J. G. Williams (Ur.), *Encyclopedia of computer science and technology* (Svez. 25, str. 143-158). New York: Marcel Dekker Inc.

Stoyan, H. (2008). Lisp 50 years ago. *Proceeding LISP50 Celebrating the 50th Anniversary of Lisp.* New York: ACM.

Stoyan, H. (n.d.). The influence of the designer on the design – J. McCarthy and LISP. U *Artificial intelligence and mathematical theory of computation* (str. 409-426). San Diego, CA, 1991: Academic Press Professional, Inc.

Talcott, C. (1988). Rum – an intensional theory of function and control abstractions. U M. Boscarol, L. Carlucci Aiello, & G. Levi (Ur.), *Foundation of logic and functional programming, workshop, Trento, Italy, December 15-19, 1986* (str. 3-44). Berlin: Springer-Verlag.

Turing, A. M. (1936). On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, 42*(2).

Turing, A. M. (1950). Computing machinery and intelligence. *Mind, 59*(236), 433-460.

Turing, A. M. (1986). Lecture to the London Mathematical Society on 20 February 1947. U A. M. Turing, B. E. Carpenter, & R. W. Doran (Ur.), *A. M. Turings ACE report of 1946 and other papers.* Cambridge, Massachusets, USA: Massachusetts Institute of Technology.

Turner, D. (1984). Functional programs as executable specifications. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences, 312*(1522), 363-388.

von Neumann, J. (1951). The general and logical theory of automata. U L. A. Jeffress (Ur.), *Cerebral mechanisms in behavior; the Hixon Symposium. 5*, str. 1-41. Oxford, England: Wiley.

Wang, H. (siječanj 1960). Toward mechanical mathematics. *IBM Journal of Research and Development, 4*(1), 1-22.

Weizenbaum, J. (April 1967). Rev. of The LISP 2 programming language and system, by P. W. Abrahams and C. Weissman. *IEEE Transactions on Electronic Computers, EC-16*(2), str. 236-238.

Woodward, P. M., & Jenkins, D. P. (1961). Atoms and lists. *The Computer Journal, 4*(1), 47-53.

AUTOR: Kazimir Majorinc
NASLOV: Moćan koliko je god moguće
– glavne ideje Lispa u McCarthyjevom periodu

IZDAVAČ: Multimedijalni institut
Preradovićeva 18
HR-10000 Zagreb
TELEFON: +385 [0]1 48 56 400
FAX: +385 [0]1 48 55 729
E-MAIL: mi2@mi2.hr
URL: http://www.mi2.hr

BIBLIOTEKA: Basic - biblioteka za progresivnu kulturu
i nove tehnologije
UREDNICI: Tomislav Medak, Petar Milat, Marcell Mars

RECENZENT: Alan Pavičić Aka
LEKTORI: Ivana Pejić, Ante Jerić

OBLIKOVANJE: Ruta
PISMA: Centuy Schoolbook, Fedra Mono, Lexicon
PAPIR: Munken Lynx 100 gr.
TISAK: Tiskara Zelina d.d.
NAKLADA: 300

Zagreb, srpanj 2015.

Tiskanje ove publikacije omogućeno je temeljem potpore Ministarstva socijalne politike i mladih.

Ministarstvo socijalne politike i mladih

ISBN 978-953-7372-28-6