```
*****************************************************************

          i                          L      I     SSS     PPP
m m m          ccc   r rr    ooo     L      I   S    S    P  P
mm m m    i   c         rr   o   o   L      I   S         P  P
m  m m    i   c         r    o   o   L      I    SSS      PPP
m  m m    i   c         r    o   o   L      I        S    P
m  m m    i     ccc     r     ooo    LLLL   I     SSS     P

*****************************************************************
```

1. UVOD

Tento manual nie je ucebnicou jazyka LISP. S programovacim jazykom LISP sa mozno oboznamit napr. vo Winston: LISP, Molnar-Navrat:Specialne programovacie jazyky - programovanie v jazyku LISP, Muller:Specialni programovaci jazyky a v dalsich.

MicroLISP je implementacia podmnoziny jazyka LISP na mikropocitace ZX Spectrum a PMD-85. Obsahuje vacsinu "dolezitych" funkcii LISPu 1.5. Svojou architekturou vychadza zo SpecLISP-u 1.3 od fy Serious Software, od ktoreho sa lisi napr. tym ze funguje, su v nom odstranene (dufame) vsetky fatalne chyby SpecLISPu, zmenenu funkciu vyhodnocovaca S-vyrazov, rychlejsie a efektivnejsie strojove rutiny - a hlavne velky pocet novych funkcii.

MicroLISP pozostava zo 4 suborov:

1- kratky zavadzaci program v BASIC-u
2- SCREEN- atributova cast obrazovky
3- vlastny kod o dlzke zhruba 7 kB
4- textovy blok, obsahujuci mena a adresy lispovskych funkcii

Po odstartovani sa textovy blok presunie do videoRAM, vytvori sa zretazena volna pamat od adresy zhruba #7e00 po #efff (posledne 4 kB su rezervovane pre stack), potom sa vyuzitim textov a adries vytvori oblist, a system oznami vypisom hlavicky a '->' pripravenost. Od tejto chvile sa bude opakovat vecny cyklus interpreta - vstup S-vyrazu, vyhodnotenie, vystup hodnoty (READ-EVAL - PRINT).

2. KOMUNIKACIA SO SYSTEMOM.

Obsluha klavesnice je realizovana vlastnym modulom, tiez praca s obrazovkou. Okrem generatora znakov, LOAD/SAVE rutin,BEEP-u je microLISP nezavisly na ROM Spectra. Pre zvysenie rychlosti pracuje so zakazanymi preruseniami.

Rozlozenie klaves odpoveda "klasickemu", s vynechanim kodov pre tokeny. Su k dispozicii len CAPS-SHIFT a SYMBOL-SHIFT, neexistuje G, E ani K -mod. Niektore specialne znaky pristupne na standardnej klavesnici cez extended-mode, su dostupne priamo cez SYMBOL-SHIFT. Takisto su odstranene riadiace kody farby. K dispozicii su nasledovne riadiace klavesy:

Pozn: MicroLISP bol vytvoreny pomocou sysytemu MRS ((c) UAK Bratislava), z ktoreho su prebrate rutiny INKEY a SCAN. Sam interpret vsak nie je standardny kniznicny modul MRS, tabulky externalov a entry-pointov boli odstranene kompakciou kniznice.

```
CS - SPACE      BREAK-mode  (pozri dalej)
CS - 0          DELETE znak pred kurzorom
CS - 5          CANCEL zrusenie obsahu buffera
CS - 1          EXIT system
```

Studeny start systemu je 28871. POZOR!!! - da sa pouzit len raz, bezprostredne po nahrati !!!. Teply start je 28544 - zmaze sa obrazovka, vypise hlavicka, stav systemu zodpoveda stavu pri exite.

2.1 PRIMITIVNE KONSTRUKCIE

System rozlisuje nasledovne primitivy: atom, cislo, oddelovac. Ich specifikacia je nasledovna:

```
< atom >      := < pis > < zvys_id >
< zvys_id >   := < pis_cis > < zvys_id > ! []
< pis >       := ascii znaky 3AH az 7FH
< pis_cis >   := ascii znaky 30H az 7FH
< cislo >     := + < zv_cis > ! - < zv_cis > ! < zv_cis >
< zv_cis >    := < cis > < zv_cis > ! < cis >
< cis >       := 0 ! 1 ! ... ! 9
```

Cisla su len cele so znamienkom v rozsahu < -32767, 32767 >

```
Oddelovace : '(',')'      - zoznamove zatvorky
             ',' ' '      - separator
             "'"          - "quotovaci" apostrof
             '.'          - konstruktor bodka-dvojice
             '%'          - uzatvorenie vsetkych '('
```

Ostatne znaky vedu k chybovemu hlaseniu.

2.2 VSTUP A VYSTUP

Pri zadavani vstupu sa text zapisuje do buffera o dlzke 254 byte. Vstup sa ihned konvertuje do vnutorneho zoznamoveho tvaru a obsah buffera sa nahradi novym. ! MicroLISP nema k dispozicii ziaden 'zdrojovy tvar' programu ako suboru ASCII znakov, z coho vyplyvaju aj obmedzenia pri editovani.(*)

Po napovedi '->' ocakava LISP vstup znakov az po ENTER. Pri presiahnuti poctu 253 znakov sa vlozi ENTER automaticky - cize maximalna dlzka identifikatora je 253 znakov (ENTER sa chape ako oddelovac). Pri vypise vkladaneho textu prechadza automaticky na novy riadok, na poslednom znaku posledneho riadku nasleduje scroll o 1 riadok. Presun vypisu na novy riadok ani scroll sa

nechapu ako oddelovace. Vo vkladanom riadku sa mozno deletovanim vracat a opravovat chybne znaky az po zaciatok aktualneho buffera bez ohladu na riadky na obrazovke. Odoslany riadok nie je mozne editovat (ale je mozne editovat priamo vnutorny tvar programu ako S-vyraz - pozri funkciu EDI). Podrzanie kazdej klavesy ma za nasledok autorepeat, okrem klavesy 'T', ktora ma specialnu funkciu.

Po odoslani system konvertuje obsah buffera do vnutorneho tvaru a zaroven kontroluje pripustnost vytvaranej struktury. Ak sa uzivatel pokusal vlozit nespravnu strukturu (napr. S-vyraz typu (a b.c d), hlasi system 'READ LIST ERROR'. Po uspesnom konvertovani kontroluje system pocet neuzatvorenych lavych zatvoriek a ak je nenulovy, pozaduje vstup dalsieho textu do noveho buffera s vypisom '_' do noveho riadku, az kym sa vsetky vlozene lave zatvorky neuzavru. Nato sa vlozena struktura vyhodnoti a vytlaci sa jej hodnota.

Pri vstupe ako oddelovac atomickych symbolov sluzi medzera - (lub. pocet), ciarka (lub. pocet), ENTER alebo niektory z vyssie definovanych oddelovacov. Pri vystupe konvertuje LISP S-vyraz do postupnosti ASCII znakov. Atomicke symboly oddeluje medzerami a pri tlaceni nerespektuje hranice riadkov. Ak je to mozne, pouziva pri vystupe zoznamovu notaciu, DTPR-notaciu (DoTted-PaiR, bodka-dvojica) pouziva len v nutnych pripadoch. (Podrobnejsie o jednotlivych notaciach v Muller: Specialni programovaci jazyky). Pri dosiahnuti posledneho riadku sa automaticky scrolluju riadky 2 az 24. Vypis mozno zastavit stlacenim a podrzanim klavesy 't' - to je dovod, preco nema autorepeat.

System rozlisuje male a velke pismena. Vsetky standardne funkcie maju identifikatory LEN z malych pismen. To iste plati o systemovych atomoch ako su t,nil,subr atd. (Presvecte sa, ze system pozna funkciu car ale nie Car ani CAR,). Uzivatel moze v identifikatoroch pouzivat lubovolne male i velke pismena. Aj ked v tomto navode pre zvyraznenie uvedieme systemovu funkciu pisanu velkymi pismenami, pri komunikacii s microLISPom sa pouzivaju VYHRADNE standardne atomy pozostavajuce z malych pismen.

Podrobnejsie o vstupe a vystupe mozno najst v prilohe A. Zaverom este k editovacim obmedzeniam. Pretoze interpret LISPu ma dost velke naroky na pamat a drviva vacsina uzivatelov Spectra nema k dispozicii vonkajsie pamatove media s priamym pristupom (floppy ci hard disk), nie mozne drzat v pamati 'zdrojovy tvar' programu ako ASCII subor. Hlavnym poslanim mikro LISPu je prve zoznamenie sa s LISPom a ziskanie zakladnych programatorskych skusenosti a obmedzenie pracovnej oblasti by malo za nasledok zahltenie systemu uz pri pomerne trivialnych pokusoch s rekurzivnymi funkciami. Ciastocnou nahradou tohto nedostatku je editor S-vyrazov - funkcia EDI.

3. POPIS FUNKCII

Na uvod treba povedat, ze mikroLISP pracuje vylucne v EVAL-mode (nema EVALQUOTE mod) a teda po zadani atomu na vstup sa snazi vratit jeho hodnotu, po zadani S-vyrazu tvaru (X Y Z) sa pouzije X pre referencovanie funkcie a Y a Z sa povazuju za argumenty. Z hladiska systemu neexistuju rozdiely medzi funkciami typu SUBR a FSUBR - interne su vsetky strojove funkcie typu FSUBR - pri vyhodnocovani sa odovzda funkcii zoznam nevyhodnotenych argumentov a sama funkcia ich podla potreby vyhodnocuje. Samotne spracovanie argumentov sposobuje, ze z hladiska uzivatela sa funkcie spravaju korektne. Podrobnejsie v prilohe.

Pri pouzivani systemu je dolezite poznat organizaciu dvoch lisppvskych zasobnikov pouzivanych systemom - OBLISTu a NAME-STACKu.

OBLIST - (OBject LIST) je zoznam vsetkych atomov znamych systemu. Jeho organizacia vyplyva zo struktury atomu: Smernik na atom je smernik na CCB (car-cdr bunka, zakladny prvok ZVP, zretazenej volnej pamati), v hlavicke ktorej je priznak atomu. Standardne funkcie spristupnovania casti S-vyrazu (car,cdr) sa pri narazeni na takuto atomicku bunku zastavia a vratia chybove hlasenie, pretoze car- aj cdr- cast tejto CCB maju specialnu semantiku. Car cast atomickej CCB je smernik na PLIST atomu - (Property LIST, zoznam vlastnosti) a cdr-cast je smernik na dalsi atom v obliste. Kazdy atom existuje jedinecne, zretazene su pomocou cdr-casti. Oblist je zasobnik, rozsiruje sa dolava a zlava sa aj prehladava a vypisuje. Manipulacia s OBLISTom je zalezitostou systemu, uzivatel ma moznost zmeny. (Pozri popis systemovych funkcii a prilohy).

NAME-STACK je lispovsky zasobnik obsahujuci DTPR (bodka-dvojice) atom.hodnota - teda ((a1.h1)(a2.h2) .....). Rozsiruje a skracuje sa z lavej strany a odraza vzdy aktualny stav vazieb hodnot atomov. Manipulacia s NS je VYHRADNOU zalezitostou systemu. Podrobnejsie pozri prilohu A.

Pri popise funkcii budu platit nasledovne konvencie:

Riadok zadany uzivatelom zacina napovednym znakom '->', ktory sa skutocne vypise, ked LISP caka na vstup a konci znakom <cr>. Riadky predstavujuce odpoved systemu budu zacinat ':', ktory sa ale pri praci zo systemom nevypisuje, sluzi len v tomto texte pre rozlisenie vstupu a vystupu. Pri slovnom opise cinnosti funkcie budeme argumenty funkcie oznacovat pismenami X, Y, Z .. a hodnoty argumentov, ak ich funkcia vyhodnocuje ako XX, YY ... Ku kazdej funkcii sa budeme snazit uviest niekolko prikladov, ktore maju sluzit na ilustraciu cinnosti funkcie a nie ako príklady programovania v LISPe. Ucelenejsie problemy a priklady programov su v prilohe C.

CONS - funkcia 2 argumentov. Vytvori novu CCB, ktorej car-cast je smernik na hodnotu prveho a cdr cast smernik na hodnotu druheho argumentu. Teda (cons X Y) vrati (XX.YY). Inac povedane XX sa vsunie do zoznamu YY ako prvy prvok.

```
->(cons 'a 'b)<cr>
:(a.b)
->(cons 11 12)<cr>
:(11.12)
->(cons '(a b) '(c d))<cr>
:( (a b) c d)
```

CAR - funkcia jedneho argumentu, ktory sa musi vyhodnotit na S-vyraz. (car X) vrati prvy prvok zoznamu XX alebo car cast DTPR XX. Ak XX je atom alebo prazdny zoznam, nasleduje chybove hlasenie 'CANT TAKE CAR OF ',XX.

```
->(car '(a b))<cr>
:a
->(car '((a b) c))<cr>
:(a b)
->(car 11)<cr>
:CAN'T TAKE CAR OF 11
->(car 'a)<cr>
:CAN'T TAKE CAR OF a
```

CDR - funkcia jedneho argumentu, ktory sa musi vyhodnotit na S-vyraz. (cdr X) vrati zvysok zoznamu XX bez prveho prvku, prip. cdr cast DTPR XX. Ak XX nie je S-vyraz, chybove hlasenie.

```
->(cdr '(a b))<cr>
:(b)
->(cdr 'b)<cr>
:CAN'T TAKE CDR OF b
->(cdr (cdr '(a b)))<cr>
:nil
```

Pre istotu pripominame, ze pouzivanie ' notacie ma funkciu quote - zabranit pokusu o vyhodnocovanie argumentu. Quote-argumenty pouzivame len pre nazornost - aby boli jasne vstupy funkcii. V poslednom priklade sa vyhodnotil argument a jeho hodnotou (hodnotou vnutornej funkcie cdr sa stalo (b). To bolo vstupom XX do vonkajsej funkcie cdr, ktora vratila nil, lebo zoznam mal 1 prvok. Keby sme zabudli dat quote pred argument (a b), nasledovalo by pravdepodobne:

```
->(cdr (a b))<cr>
:a IS NOT A FUNCTION
```

pretoze LISP pri pokuse o vyhodnotenie argumentu povazoval a za funkciu ale nenasiel pre a ziadnu funkcnu vazbu. (Pozri prilohu A - vyhodnocovanie SV). Naopak pri pridani quote:

```
->(car '(cdr '(a b)))<cr>
:cdr
```

sa hodnotou XX stane zoznam (cdr (quote (a b))), ktoreho prvy prvok je atom cdr. Podrobnejsie v A a C

ATOM - funkcia jedneho argumentu, predikat. (atom X) vrati true (vypis ako t) prave vtedy ked XX je atomicky symbol - atom alebo cislo, inac vrati nil.

```
->(atom 'a)<cr>
:t
->(atom '(a))<cr>
:nil
->(atom 11)<cr>
:t
->(atom '(11))<cr>
:nil
->(atom (11))<cr>
:11 IS NOT A FUNCTION
```

NULL - funkcia jedneho argumentu, predikat. (null X) vrati true prave vtedy ked XX je nil (alebo (), pozri prilohu A nil a systemovy nil), inac vrati nil.

```
->(null 'a)<cr>
:nil
->(null nil)<cr>
:t
->(null 'nil)<cr>
:nil
->(null ())<cr>
->t
```

V druhom priklade je hodnotou atomu nil, ktory je staticka premenna (pozri vlastnost APVAL) 'systemovy' nil (pozri A). V tretom priklade je hodnotou XX smernik na atom - teda nie nil. Prazdny zoznam je vzdy nil.

EQ - funkcia dvoch argumentov, predikat. (eq X Y) vrati t len v pripade, ked XX je atom totozny s YY, inac nil. Funkcia nekontroluje, ci XX a YY su naozaj atomy, ak nie su, vysledkom bude urcite nil. Vyuziva sa jedinecnost reprezentacie atomov, eq len porovna smerniky. Nefunguje pre cisla (vrati nil) - cisla nemaju jedinecnu reprezentaciu. Na porovnanie cisel a S-vyrazov treba pouzit funkciu EQUAL.

```
->(eq 'a 'a)<cr>
:t
->(eq 'a 'b)<cr>
:nil
->(eq 11 11)<cr>
:nil
```

Ak plati a=(b c), potom:

```
->(eq a a)<cr>
:t
```

pretoze hodnotou oboch je ten isty zoznam (b c) - smerniky ukazuju na fyzicky rovnaku strukturu - hodnota atomu a v NS. Ale

```
->(eq a '(b c))<cr>
:nil
->(eq '(b c)'(b c))<cr>
:nil
```

tu uz rovnost smernikov neplati.

QUOTE - funkcia jedneho argumentu. Vrati svoj nevyhodnoteny argument - teda (quote X) vrati X. Sluzi na zastavenie vyhodnocovacej funkcie pri jej rekurzivnom vnarani sa do struktury S-vyrazu. V microLISPe plati "quotovacia konvencia" a zapis (quote a) mozno skratit na 'a, ale funkcia quote je k dispozicii aj v neskratenom tvare - system interne preklada vstupy typu 'x do tvaru (quote x)

```
->(quote x)<cr>
:x
->(quote (a b c))<cr>
:(a b c)
->(quote 11)<cr>
:11
->(quote quote)<cr>
:quote
->'x<cr>
:x
->'(a b c)<cr>
:(a b c)
->(read)<cr>
_'x<cr>
:(quote x)
```

Posledny priklad je ukazkou prekladu 'x do tvaru (quote x). Po vyhodnoteni funkcie (read) caka lisp vstup S-vyrazu s vypisom '_' do noveho riadku. Hodnotou funkcie je nacitany SV.

SET - funkcia dvoch premennych, obdoba priradovacieho prikazu v Pascale. (set X Y) priradi atomu XX hodnotu YY. Ak sa X nevyhodnoti na atom, chybove hlasenie 'XX IS NOT SYMBOLIC'. Hodnota YY je 'dynamicky viazana hodnota ' atomu XX - cez NS. Ak atom XX este nebol viazany - t.zn ze pri zadani:

```
->XX<cr>
```

bola odpoved systemu

```
:XX IS AN UNBOUNDED VARIABLE
```

prida sa zlava do NS nova DTPR (XX.YY), a ak uz XX bol viazany zmeni sa hodnota jeho najlavejsej vazby na YY. Nech hodnotou atomu a je b, a hodnotou atomu c1 je d.

```
->a<cr>
:b
->(set a 11)<cr>
:11
->a<cr>
:b
->b<cr>
:11
```

Vidime ze hodnotou funkcie SET je hodnota druheho argumentu =YY. Priradenie hodnoty je 'vedlajsi efekt'. Ale pokracujme:

```
->c1<cr>
:d
->d<cr>
:d IS AN UNBOUNDED VARIABLE
->(set c1 '(a b c))<cr>
:(a b c)
->d<cr>
:(a b c)
->(set d b)<cr>
:(a b c) IS NOT SYMBOLIC
->b<cr>
:11
```

SETQ - funkcia dvoch premennych. Pracuje rovnako ako SET, ale prvy argument sa nevyhodnocuje. Teda (setq X Y) priradi atomu X hodnotu YY a vrati YY. Zapis (setq a b) je ekvivalentny zapisu (set 'a b). Ak prvy argument nie je atom, hlasenie 'X IS NOT SYMBOLIC'.

```
->(setq a 12)<cr>
:12
->a<cr>
:12
->(setq a '(q w e))<cr>
:(q w e)
->a<cr>
:(q w e)
Ale :
->(setq a (q w e))<cr>
:q IS NOT A FUNCTION
```

Je Vam jasne, preco ? Ak nie, skuste si predsa len precitat nejaku dobru ucebnicu LISPu (napriklad Winstonov LISP, alebo ovela dostupnejsie a velmi dobre a zrozumitelne pisane skripta EF SVST: Molnar-Navrat: Specialne programovacie jazyky - programovanie v jazyku LISp.) Ak Vam to ani potom nebude jasne, skuste sa nad sebou zamysliet...

DALSIE PREDIKATY:

NUMBERP - funkcia jedneho argumentu, (numberp X) vrati true ak XX je cislo, inac vrati nil. (Ako sa rozozna cislo od atomu, poz ri Prilohu A).

```
->(setq a 11)<cr>
:11
->(setq b 'a)<cr>
:a
->(numberp 11)<cr>
:t
->(numberp a)<cr>
```

```
:t
->(numberp b)<cr>
:nil
->(numberp '(11))<cr>
:nil
```

Nasledujuce dva predikaty pracuju velmi podobne. Oba argumenty sa musia vyhodnotit na cislo, inac chybove hlasenie 'XX IS NOT A NUMBER'. (lessp X Y) vrati t prave vtedy ked XX < YY, inac vrati nil. (greaterp X Y) vrati t prave vtedy, ked XX > YY, inac nil.

Nech a = 11 a b = 3.

```
->(lessp 11 3)<cr>
:nil
->(lessp 3 b)<cr>
:nil
->(lessp b a)<cr>
:t
->(greaterp 11 3)<cr>
:t
->(greaterp 3 3)<cr>
:nil
->(greaterp b a)<cr>
:nil
->(lessp 'c 11)<cr>
:c IS NOT A NUMBER
```

Dalsie dva predikaty su unarne.

ZEROP, MINUSP - funkcie jedneho argumentu, ktory sa musi vyhodnotit na cislo (inac hlasenie 'XX IS NOT A NUMBER'. (zerop X) vrati t prave vtedy ak XX je cislo 0, inac nil. (minusp X) vrati t prave vtedy ked XX<0, inac nil.

Nech a=0, b=-5, c=(a b).

```
->(zerop 11)<cr>
:nil
->(zerop b)<cr>
:nil
->(minusp b)<cr>
:t
->(minusp a)<cr>
:nil
->(zerop a)<cr>
:t
->(zerop c)<cr>
:(a b) IS NOT A NUMBER
```

EQUAL - funkcia dvoch argumentov. Argumentami mozu byt lubovolne SV. (equal X Y) vrati true ak S-vyrazy XX a YY su rovnake. EQUAL je 'nadmnozina' funkcie EQ, korektne porovna aj cisla, aj atomy.

Nech platia nasledovna vazba: a=(a b c).

```
->(equal a a)<cr>
:t
->(equal a '(a c b))<cr>
:nil
->(equal a '(a b c))<cr>
:t
->(equal '(a b c)'(a b c))<cr>
:t
->(equal 11 11)<cr>
:t
->(eq 11 11)<cr>
:nil
->(equal nil ())<cr>
:t
```

ARITMETICKE A LOGICKE FUNKCIE

Hlavnou oblastou pouzitia jazyka LISP rozhodne nie su numericke vypocty. Pouzitie LISPu tu naraza na malu rychlost vypoctov a pamatovo neefektivne ukladanie dat. MicroLISP je vybaveny zakladnymi aritmetickymi funkciami pracujucimi s celymi cislami. Cisla su v rozsahu 15 bitov + znamienkovy, v doplnkovom kode. Pouzitelny rozsah je <-32768,32767>. K dispozicii su nasledujuce aritmeticke funkcie:

```
binarne -  PLUS,DIFF,TIMES,DIV,REM
unarne  -  ADD1,SUB1
n-arne  -  MAX,MIN
```

Vysledkom vsetkych funkcii je cislo (ciselna CCB - vid A).

Binarne aritmeticke funkcie :

Ocakavaju dva argumenty, oba sa musia vyhodnotit na cislo, inac hlasenie 'XX IS NOT A NUMBER'. Vysledkom je:

```
(plus X Y)     .. vrati XX+YY
(diff X Y)     .. vrati XX-YY
(times X Y)    .. vrati XX*YY
(div X Y)      .. vrati celociselny podiel XX/YY
(rem X Y)      .. vrati zvysok po celociselnom deleni
```

Prekrocenie rozsahu sa signalizuje chybovym hlasenim 'ARITHMETIC OVERFLOW'.

```
->(plus 2 3)<cr>
:5
->(plus 3 -2)<cr>
:1
->(diff 3 2)<cr>
```

```
:1
->(times 3 4)<cr>
:12
->(times 1 -3)<cr>
:-3
->(div 7 3)<cr>
:2
->(div 3 4)<cr>
:0
->(times 300 300)<cr>
:ARITHMETIC OVERFLOW
->(rem 7 'a)<cr>
: a IS NOT A NUMBER
```

Unarne aritmeticke funkcie:

Vyzaduju jeden argument, ktory sa musi vyhodnotit na cislo. Ak sa nevyhodnoti na cislo, hlasenie 'XX IS NOT A NUMBER'. Ak ma 0 alebo viac nez 1 argument, hlasenie 'ARE INCORRECT ARGUMENT'.

```
(add1 X)  ..  vrati XX+1
(sub1 X)  ..  vrati XX-1
```

Rozsah zobrazitelnosti sa kontroluje ako u binarnych.

```
->(add1 1)<cr>
:2
->(sub1 3)<cr>
:2
->(add1 'a)<cr>
: a IS NOT A NUMBER
->(sub1 -32768)<cr>
:ARITHMETIC OVERFLOW
```

N-arne aritmeticke funkcie:

Nie su v pravom zmysle slova n-arne. Pozaduju jeden argument ktory sa musi vyhodnotit na zoznam obsahujuci lubovolne mnozstvo cisiel. Ich hodnotou je najvacsie/najmensie cislo zo zoznamu. Ak sa argument nevyhodnoti na zoznam, alebo ak sa pocas prehladavania zoznamu najde neciselny prvok, hlasi sa chyba.

```
->(max '(1 3 7 4 2))<cr>
:7
->(max 1)<cr>
:1
:ARE INCORRECT ARGUMENTS
->(min '(-1 2 3 -4))<cr>
:-4
->(min 'a)<cr>
: a
:ARE INCORRECT ARGUMENTS
->(max '(1 2 3 a 7))<cr>
: a IS NOT A NUMBER
```

LOGICKE FUNKCIE

NOT - ocakava jeden argument, vyhodnoti ho. Ak nesuhlasi pocet argumentov, chybove hlasenie 'ARE INCORRECT ARGUMENTS'. Funguje rovnako ako NULL - ten isty podprogram v strojovom kode.

AND - funkcia lubovolneho poctu argumentov. Vyhodnocuje svoje argumenty zlava doprava, kym nepride na koniec alebo na formu ktora sa vyhodnoti na nil. Ak taka forma existuje vrati nil, inac hodnotu poslednej vlozenej formy. Pripominame, ze LISP vsetko co nie je nil povazuje za "true" a teda funkcia bude aj v podmienkovej casti fungovat korektne, hoci v pripade pozitivneho vysledku nevracia explicitne atom t.

```
->(and 1 'a '(a b c))<cr>
:(a b c)
->(and 1 nil 2)<cr>
:nil
->(and 2 () 'a)<cr>
```

Ked si uvedomime, ze AND vrati nil len vtedy, ak najde aspon jednu formu (argument), ktora sa vyhodnoti na nil, neprekvapi nas:

```
->(and)<cr>
:t
```

OR - funkcia lubovolneho poctu argumentov. Vyhodnocuje svoje argumenty, kym nenarazi na argument ktoreho hodnota je rozna od nil, (vtedy vrati jeho hodnotu) alebo na koniec zoznamu (a vrati nil).

```
->(or nil 1)<cr>
:1
->(or () nil)<cr>
:nil
```

a v zmysle posledneho prikladu pri funkcii AND bude platit:

```
->(or)<cr>
:nil
```

DALSIE ZAKLADNE FUNKCIE

Odteraz v uvadzanych prikladoch nebudeme explicitne uvadzat znak konca riadku ENTER znaceny ako <cr>. Pri zadavani formy je po stlaceni ENTER jasne ci system povazuje vstup formy za ukonceny (a hned vrati vysledok), alebo mu chybaju prave zatvorky (a v takom pripade vypise do noveho riadku '_' a caka na dalsi vstup. Kazdy riadok odpovede systemu bude dosledne oznacovany pociatocnym sysmbolom ':'. Teda vstup z klavesnice pouzivatela je vsetko medzi promptom '->' a prvym riadkom s oznacenim ':' a medzi jednotlivymi primitivami S-vyrazu moze byt lubovolny pocet stlacenia klavesy ENTER.

MEMBER - funkcia dvoch argumentov, (member X Y) testuje, ci zoznam alebo atom XX je prvkom zoznamu YY na najvyssej urovni. Ak ano, vrati zvysok zoznamu YY ktoreho car-cast obsahuje zoznam XX na najvyssej urovni.

```
->(member 'x '(a x b))
:(x b)
->(member 'x '(a s b))
:nil
->(member 'x '(a s (x b) c))
:nil
->(member 'x ())
:nil
->(member x x))
:nil
```

Z posledneho prikladu vidiet, ze member nie je obdoba podmnoziny a teda nikdy neplati (member x x) vrati nie nil. Skusme funkciu pre zoznam-zoznam:

```
->(member '(a b) '(a b (a b) s))
:((a b) s)
```

Mozno sa presvedcit, ze korektne vysledky ziskame len pre zoznamove struktury XX a YY. Skusme pouzit ako argument S-vyraz ktory nie je zoznam:

```
->(member 'f '(a s d.f))
:nil
->(member '(d.f)'(a s d.f))
```

LIST - fukcia lubovolneho poctu argumentov. (list X Y Z) vrati zoznam (XX YY ZZ).

```
->(list 'a)
:(a)
->(list 'a  '(s d) 1)
:(a (s d) 1)
->(list  (setq a '(b c)) (setq e '(c d)) )
:((a b)(c d))
->(list a e)
:((a b)(c d))
```

REVERSE - funkcia jedneho argumentu, ktory sa musi vyhodnotit na zoznam. (reverse X) vrati obrateny zoznam k XX na najvyssej urovni. XX musi byt S-vyraz so zoznamovou strukturou.

```
->(reverse '(a b s))
:(s b a)
->(reverse)
:nil
:ARE INCORRECT ARGUMENTS
->(reverse ())
:nil
```

Je to pravda - obratenym zoznamom k prazdnemu zoznamu je zasa prazdny zoznam. MicroLISP nerozlisuje () - prazdny zoznam a hodnotu atomu nil,a pri vystupe vrati vzdy nil aj ked sa jedna o ()
Prazdny zoznam je jedina struktura, ktora je sucastne zoznamom aj atomom.
Pokus o obratenie S-vyrazu, ktory nie je zoznamom, ma zaujimave dosledky.

```
->(reverse '(a s d.f))
:((apval nil pname (nil) apval t pname (t) ....
:           .
:           .
: pname f) d s a)
```

Pokus o obratenie DTPR na konci sposobil, ze miesto f sa vypisal plist kazdeho atomu v obliste (pozri prilohu A). Na konci vypisu sa objavil obrateny zvysok S-vyrazu. Tento jav nie je chyba systemu,ale dosledok nedodrzania vstupnej konvencie. Na zaver este ukazka toho, ze REVERSE obracia zoznam LEN na najvyssej urovni. (V casti C je popisana lispovska funkcia TREVERSE obracajuca vsetky neatomicke prvky zoznamu na vsetkych urovniach).

```
->(reverse '(a (s d) f))
:(f (s d) a)
```

AMONG - funkcia 2 argumentov. Ma podobnu funkciu ako MEMBER, no hlada vyskyt atomu v zozname na lubovolnej urovni. (among X Y) vrati nil ak atom XX nie je vobec prvkom zoznamu YY, inac vrati taky zvysok YY1 zoznamu YY, ze YY1 je (cdr 'YY2) - a YY2 je taky zvysok zoznamu YY, ze XX je prvkom (car 'YY2) na lubovolnej urovni. Ak by YY1 bol prazdny, vrati t. Co sa skryva za touto dost krkolomnou definiciou, ukazu priklady:

```
->(among 'a '(b a c))
:(c)
->(among 'x '(b a c))
:nil
->(among 'a '(b  ( c a b ) c))
:(c)
->(among 'a ())
:nil
->(among 'x '(a b x))
:t
```

Pokus o (among '(a b)'(x y z)) sposobi chybove hlasenie.

APPEND - funkcia 2 argumentov. (append X Y) pripoji zoznam YY za zoznam XX ako jeho linearne pokracovanie. Hodnoty X ani Y sa pritom nemenia. Pozor - XX aj YY nesmu byt atomy !!!

```
->(append (setq a '(b c)) (setq b '(e r)))
:(a b e r)
->a
:(b c)
->b
:(e r)
->(append b a)
:(e r a b)
```

Funkcie sprístupňovania častí S-výrazu:

Jednoduché funkcie CAR a CDR možno skladať do rôznych kombinovaných funkcií - napr. funkcia (cadr X) je ekvivalentná zloženiu funkcií (car (cdr X)). V mikroLISPe sú implementované niektoré zložené funkcie až do stupňa 3. Všetky tieto funkcie rešpektujú pravidlá CAR a CDR - zastavia s chybovým hlásením pri pokuse získať car alebo cdr atomickej štruktúry alebo (). Štandardne sú k dispozícii tieto zložené funkcie:

```
CDDR  -  (cddr X)    ... (cdr (cdr X))
CADR  -  (cdar X)    ... (car (cdr X))
CDAR  -  (cadr X)    ... (cdr (car X))
CAAR  -  (caar X)    ... (car (car X))
CDADR -  (cdadr X)   ... (cdr (car (cdr X)))
CAAAR -  (caaar X)   ... (car (car (car X)))
CAADR -  (caadr X)   ... (car (car (cdr X)))
CADDR -  (caddr X)   ... (car (cdr (cdr X)))
```

Používanie zložených funkcií namiesto kompozície jednoduchých je podstatne rýchlejšie a pamäťovo ovela menej náročné.

```
->(caar '((a b) c))
:a
->(cdar '((a b) c))
:(b)
->(caaar '((a b) c))
:CAN'T TAKE CAR OF a
```

SPECIÁLNE FUNKCIE

Táto skupina funkcií sa od doteraz spomínaných líši hlavne spôsobom spracovania argumentov. Hoci interne sú všetky funkcie mikroLISPu "typu FSUBR", (pozri prílohu A) všetky zatiaľ spomínané funkcie si vyhodnocovali všetky argumenty. Dalšou typickou črtou špeciálnych funkcií je, že často ovela väčší význam ako hodnota funkcie (hodnota, ktorú funkcia vráti) majú rôzne "vedľajšie efekty pri ich spracovaní.

DE - je implementáciou lispovskej funkcie DEFINE. Jej syntax je nasledovná: (de X Y Z) - požaduje 3 argumenty - inak chybové hlásenie. X sa nevyhodnocuje a musí byť atom. Y sa nevyhodnocuje a musí byť zoznam (aj prázdny), ktorého prvky sú výlučne atómy. Ani Z sa nevyhodnocuje a predstavuje telo užívateľom definovanej funkcie. Funkcia DE vráti ako svoju hodnotu meno funkcie a do P-listu atómu X sa zaradí vlastnosť s indikátorom EXPR a hodnotou lambda- výrazu definovaného zoznamom (lambda Y Z). Príklad:

```
->(de aa (a b)(cons (list a) (list b)))
:aa
```

Od tejto chvile je mikroLISP rozsireny o novu funkciu s menom aa - teda uvedenie mena aa na prvom mieste zoznamu sposobi aplikaciu funkcie na zvysok zoznamu ako na parametre.

```
->(aa 'a 'b)
:((a) b)
```

Na hodnoty formalnych parametrov sa naviazali hodnoty argumentov funkcie aa a vyhodnotilo sa jej telo. (Podrobnejsie pozri Lambda-konverzia v prilohe A). Na uvedenom priklade je vidiet, ze mikroLISP dosledne preferuje zoznamovu notaciu pred DTPR notaciou pokial je to mozne. Alternativny tvar vysledku (aa 'a 'b) by bol ((a).(b)) pripadne ((a.nil).(b.nil)).

Poznamka: Hoci pri definovani funkcie sa neuvadza komplektny tvar Lambda-vyrazu, ako vlastnost EXPR sa ulozi standardny lispovsky Lambda-vyraz. Presvecte sa o tom:

```
->(get 'aa 'expr)
:(lambda (a b)(cons (list a)(list b)))
```

Funkcia GET je popisana dalej.

Pouzitim DE mozno predefinovat standardne funkcie mikroLISPU implementovane v strojovom kode - pri hladani funkcnej vazby sa najskor testuje vlastnost EXPR (vid popis vyhodnocovacej funkcie v A). Priklad:

```
->(car '(a b))
:a
->(de car (x)(cdr x))
:car
->(car '(a b))
:(b)
->(cdr '(a b))
:(b)
```

Povodny tvar funkcie - vlastnost SUBR sa predefinovanim nestrati - len sa "zatieni" a odstranenim vedlajsich ucinkov funkcie DE mame k dispozicii funkciu zasa v povodnom tvare.

```
->(remprop 'car 'expr)
:t
->(car '(a b))
:a
```

Funkcia REMPROP je popisana v dalsom. Napriek uvedenemu prikladu odporucame moznost predefinovania standardnych funkcii pouzivat az po dokladnej uvahe, lebo velmi casto vedie k zahadnym a tazko odhalitelnym chybam v programe.

COND - specialna funkcia premenneho poctu argumentov. Forma ma syntax: (cond (P1 D11 D12 .. D1n) ... (Ps Ds1 Ds2 ... Dsr)) - teda ma lubovolny pocet vetiev s uvedenou strukturou a kazda vetva ma aspon jeden clen. Forma sa vyhodnocuje nasledovnym sposobom : najprv sa vyhodnoti prva forma vetvy Pi a ak PPi je rozne od nil, vyhodnocuju sa postupne vsetky formy v i-tej vetve a vysledkom celej formy COND sa stava hodnota poslednej formy vo vetve i. (Ak ma vetva len jeden prvok, vrati sa jeho hodnota). Ak PPi bolo rovne nil, rovnakym sposobom sa postupuje na dalsiu vetvu - az po vycerpanie vetiev alebo najdenie takej vetvy j, ze PPj nie je nil. Ak predpoklad Pi ziadnej z vetiev sa nevyhodnotil na hodnotu roznu od nil, stava sa nil hodnotou celej formy COND (a nie chybove hlasenie ako napr. u LISPu 1.5). Toto plati rovnako v aj mimo formy PROG.

Forma COND v LISPe ma podobnu ulohu ako IF-THEN-ELSE struktury v jazykoch typu PASCAL.

```
->(de hanoi (n a b c)
      (cond  (  (onep n) (presun a b)  )
             (  t  (hanoi (subi n) a c b)
                   (presun a b)
                   (hanoi (subi n) c b a)   )
      )
   )
:hanoi
->(de presun (a b) (print (list 'presun 'z a 'na b) )   )
:presun
->(de onep (n) (zerop (subi n)))
:onep
->(hanoi 3 'a 'b 'c)
:(presun z a na b)
:(presun z a na c)
:(presun z b na c)
:(presun z a na b)
:(presun z c na a)
:(presun z c na b)
:(presun z a na b)
```

WHILE - je specialna funkcia premenneho poctu argumentov. V niektorych implementaciach LISPu sa nevyskytuje. Do mikroLISPu bola zaradena pretoze umoznuje velmi elegantne vytvaranie cyklov typu "while" - test pred vyhodnotenim. Syntax je nasledovna:

(while P F1 F2 .. Fn), kde P a Fi su lubovolne formy.

Vyhodnocovanie mozno charakterizovat takto.

1. Vyhodnoti sa podmienka P. Ak PP je nil, ukonci sa vykonavanie formy WHILE s hodnotou nil.
2. Postupne sa vyhodnotia formy F1 az Fn a pokracuje sa bodom 1.

Aj pri funkcii WHILE je dolezity skor vedlajsi efekt funkcii, pretoze vrati vzdy hodnotu nil. Nech n =0.

```
->(while  (lessp n 5)
            (print (setq n (addi n)))     )
:1
:2
:3
:4
:5
:nil
->n
:5
->(while p)
```

POZOR - ak je hodnota atomu p <> nil, dojde k nekonecnemu cyklu. Tento mozno kedykolvek prerusit stlacenim CS a SPACE (pozri BREAK mod). Ak je hodnota p =nil, while skonci a vrati nil.

SELECT - je zovseobecnenie formy cond. Forma ma syntax:
(select P (P1 D11 D12 ... D1k) .... (Pn Dn1 Dn2 ,,, Dns) E)
a takuto semantiku:
Vyhodnoti sa forma P a hlada sa taka vetva i, ze PP=PPi. Ak sa takato vetva najde, vyhodnotia sa postupne vsetky dosledky, ktore obsahuje, a hodnotou formy SELECT sa stava hodnota posledneho dosledku. Ak sa taka vetva nenajde, hodnotou formy select je EE.

Formu SELECT mozno pomocou formy COND vyjadrit nasledovne:

```
(cond   ((equal P P1) D11 D12 ... D1k)
            .
            .
        ((equal P Pn) Dn1 Dn2 ... Dns)
        (t E)  )
```

Pozor - ak ma SELECT menej ako 3 argumenty, hlasi chybu. Kazda z vetiev MUSI mat aspon 2 prvky !!!
Pouzitie formy select je v prilohe C.

PROGN - je specialna funkcia premenneho poctu argumentov plniaca v LISPe ulohu prikazovych zatvoriek begin a end. Umoznuje vytvorenie "zlozeneho prikazu" a vlozenie postupnosti foriem na miesto jedinej. Syntax : (progn Z F1 F2 F3 .. FN), Z musi byt zoznam obsahujuci iba atomy - zoznam lokalnych premennych formy PROGN (lokalne rovnako ako u formy PROG - pozri tam). Kazda lokalna premenna sa inicializuje na nil, postupne sa vyhodnotia vsetky formy Fi a hodnotou formy PROGN sa stava hodnota poslednej z nich. Zoznam lokalnych premennych moze byt aj prazdny.

```
->(progn () (print (plus  3 4))(print (list 'a 'b))  )
:7
:(a b)
:t
```

pretoze funkcia print vrati ako hodnotu t.

PROG - je specialna funkcia premenneho poctu argumentov, umoznujuca pisat v LISPe iteracne struktury. V mikroLISPe je implementovana uplna funkcia PROG so syntaxou:
(prog X F1 F2 ... Fn) kde X reprezentuje zoznam lokalnych premennych, platnych len v tele formy prog (prip. prazdny zoznam), ktore sa po vystupe z tela prog-u stratia. Ich pouzitie neovplyvni hodnotu premennej s rovnakym menom platnu aj mimo PROG (podrobnejsie pozri navazovanie hodnot premennych v A).

Kazda z foriem Fi moze mat jeden z tvarov:

- atom - nevyhodnocuje sa, sluzi ako navestie,
- S-vyraz - vyhodnocuje sa normalnym sposobom.

MicroLISP pripusta lubovolne vnorenie foriem PROG, pricom formy GO a RETURN operuju len v ramci najvnuternejsej formy PROG, ktora ich obsahuje. O lokalnosti premennych plati pravidlo hierarchie blokov ako napr. v Algole. Hodnotou formy PROG je hodnota argumentu formy RETURN. Ak RETURN nema argumenty alebo ak forma PROG neobsahuje formu RETURN, hodnotou je nil. Pomocou funkcie GO mozno ovplyvnovat postupnost vyhodnocovania foriem. Priklad iteracneho zapisu funkcie pre scitanie zoznamu cisiel:

```
-> (de suma (z)
     (prog (s)
           (setq s 0)
      loop (cond ((null z)(return s)) )
           (setq s (plus s (car z)))
           (setq z (cdr z))
           (go loop)
     ))
:suma
->(suma '(1 2 3 4))
:10
```

Pri pouzivani formy nie je nutne pouzivat pomocnu premennu na okopirovanie hodnoty vstupnej premennej - vazba vstupnej premennej (napr. (suma a) sa vyhodnotenim (setq z (cdr z)) nezmeni - pozri A).

GO - specialna forma pouzitelna len v tele formy PROG. Ma jeden argument - (go X). Ak X je atom, nevyhodnocuje sa, inac sa musi X vyhodnotit na atom (ak ani X ani XX nie je atom, chybove hlasenim V tele formy PROG (najvnutornejsej obsahujucej GO) sa hlada navestie X /prip. XX/ a vyhodnocovanie foriem PROGu pokracuje prvou formou ZA navestim. Ak sa navestie nenajde, nasleduje chybove hlasenie 'LABEL NOT FOUND' . Pouzitie GO mimo formy PROG sposobi chybove hlasenie 'IN prog ONLY'.

RETURN - specialna forma pouzitelna len v tele formy PROG. Ukonci sa vykonavanie formy prog (najvnutornejsej obsahujucej RETURN) a hodnotou formy PROG hodnota argumentu formy RETURN (prip. nil ak RETURN bol bez argumentov). Po vyhodnoteni (return X) sa vykonavanie PROGu konci a jeho hodnotou sa stava XX. To je zaroven aj jediny sposob ako "vyviezt" hodnotu lokalnej premennej PROGu mimo jeho telo. Ako GO, aj RETURN pouzity mimo formy PROG sposobi chybove hlasenie.

FUNKCIONALY.

V mikroLISPe su implementovane 4 zakladne funkcionaly MAP, MAPCAR, MAPC a MAPLIST. Kazdy ma 2 argumenty - prvy je forma sprostredkujuca vazbu na funkciu a druhym je zoznam, ktory sa spracuva. Pozri v A moznosti funkcnej vazby - uvedene priklady nezahrnaju vsetky moznosti a su len ilustracne.

MAPCAR - je funkcional, (mapcar X Y) - XX musi sprostredkovat vazbu na funkciu jednej premennej. XX sa aplikuje na (CAR Y),na (CADR Y) atd - na jednotlive prvky zoznamu a vrati zoznam vysledkov jednotlivych aplikacii. Ak XX nereferencuje funkciu alebo YY nie je zoznam, hlasi sa chyba.

```
->(mapcar 'add1 '(1 2 3 4))
:(2 3 4 5)
->(mapcar (function print) '(TO TALK IN LISP))
:TO
:TALK
:IN
:LISP
:(t t t t)
```

pretoze funkcia PRINT vrati t.

MAP - analogia MAPCAR, ale miesto zoznamu vysledkov aplikacii sa vrati nil. Ma zmysel pri roznych vedlajsich ucinkoch XX.

MAPLIST - pozaduje argumenty ako MAPCAR, ale XX sa aplikuje na zoznamy - najprv na YY, potom na (CDR 'YY) atd az do vycerpania zoznamu. Vracia zoznam vysledkov aplikacii. Chybove hlasenie v rovnakych pripadoch ako MAPCAR.

```
->(maplist 'list '(TO TALK IN LISP))
:((TO TALK IN LISP)(TALK IN LISP)(IN LISP)(LISP))
```

MAP - funguje rovnako ako MAPLIST, ale vracia nil. Pouziva sa pri vedlajsich ucinkoch XX.

```
->(map 'print '(TO TALK IN LISP))
:TO TALK IN LISP
:TALK IN LISP
:IN LISP
:LISP
:nil
```

POZOR: Funkcie pouzivane vo funkcionaloch NESMU menit dynamicku hodnotu atomu subr !!! Pozri A.

DALSIE FUNKCIE

EVAL - je explicitne volanie vyhodnocovacej funkcie LISPu.Poza-duje jeden argument,ktory vyhodnoti a jeho hodnotu este raz vy-hodnoti.(eval X) - po vyhodnoteni X na XX sa este vyhodnoti (XX)

```
->(eval (cons 'cons '(1 2)))
:(1.2)
```

pretoze argument sa vyhodnotil na (cons 1 2).
Zaujimave moznosti pre modifikaciu programov poskytuje kombinacia (eval (read))

READ - funkcia bez argumentov. Po vyhodnoteni formy (read) caka system s vypisom '_' do noveho riadku na vstup S-vyrazu. Tento sa nevyhodnocuje, len sa prelozi do vnutornej lispovskej reprezentacie a vrati sa ako hodnota formy READ. Pre vstup platia pravidla ako pre vstup po '->' - teda caka sa na uzatvorenie vsetkych '('.

```
->(read)
'a                    /* toto je vstup zadany uzivatelom */
:(quote a)
```

PRINT - funkcia jedneho argumentu. (print X) vytlaci XX a ako svoju hodnotu vrati true.

```
->(print (plus 3 2))
:5
:t
```

LAMBDA - nie je funkcia, ale atom, ktoreho pouzitie na mieste funkcie umoznuje vkladanie tzv. lambda-vyrazov, co je v podstate priamy prikaz aplikacie funkcie a mozno pouzit ako ekvivalent volania funkcie typu EXPR. Rozdiel medzi programovou schemou

```
((LAMBDA ZOZNAMFORMALNYCHARGUMENTOV TELOFUNKCIE)
                ZOZNAMSKUTOCNYCHARGUMENTOV)
```

a prostupnostou

```
(DE MENOFUNKCIE ZOZNAMFORMALNYCHARGUMENTOV TELOFUNKCIE)
(MENOFUNKCIE ZOZNAMSKUTOCNYCHARGUMENTOV)
```

je len v tom, ze v prvom pripade sa funkcia definovana lambda vyrazom strati po vykonani, v druhom ostava v P-liste atomu MENOFUNKCIE ako hodnota vlastnosti s indikatorom EXPR.
MikroLISP dovoluje povzit lambda-vyraz na lubovolnom mieste pre refencovanie funkcie - vo funkcionaloch, forme FUNCTION (pozri dalej) a na mieste mena funkcie typu EXPR a SUBR.
Pri zapise lambda-vyrazu plati standardna syntax LISPu. Pri definovani novej funkcie pomocou DE sa pise len zoznam form. argumentov a telo funkcie a microLISP sam si doplni uvedenu strukturu na komplektny lambda vyraz.

```
->((lambda (x y)(cons (cons x y)(cons y x))) 1 2)
:((1.2) 2.1)
```

NOMEMB - pozaduje jeden argument, (nomemb X) vrati pocet prvkov zoznamu XX na najvyssej urovni. POZOR - ak XX je atom, vrati sa pocet atomov v obliste za nim nasledujucich - zretazenie atomov v obliste sa chape ako zoznam (pozri A).

```
->(nomemb '(1 2 3 4 5))
:5
```

FUNCTION - funkcia jedneho argumentu. (funktion X) nevyhodnocuje svoj argument. Okrem tejto "quotovacej" ulohy sposobi vyhodnotenie formy FUNCTION, ze pri vyhodnocovani funkcie X sa pouzije stav NAME-STACKu platny v okamihu vyhodnotenia formy FUNCTION. POZOR - X musi nejakym sposobom referencovat funkciu.
Pouzitie formy funktion demonstrujeme na priklade prevzatom zo skript Molnar L.,Navrat P.,: Specialne programovacie jazyky, strana 86 az 88.

3.7.3. Forma FUNCTION

Pri pouzivani funkcionalov sme sa prvy raz stretli s funkciami, ktorych argumenty boli funkcie. Pretoze funkcie vyhodnocuju svoje argumenty, treba pri uvedeni funkcie (napr. v tvare lambda vyrazu) ako argumnetu treba zabranit jeho vyhodnoteniu. Jeden sposob, ako zabranit vyhodnoteniu, uz poznate: citovat pomocou formy QUOTE. Pouzitie formy QUOTE na citovanie funkcii ako argumentov vsak moze v niektorych pripadoch sposobit, ze citovana funkcia nadobudne iny zmysel, nez si predstavoval programator.
Uvazujme napr. takuto ulohu. Majme dane dva zoznamy. Treba vratit taky zvysok druheho zoznamu, ktory zacina prvkom na mieste zodpovedajucom miestu (podla poradia) posledneho prvku v prvom zozname. Napr. pre zoznamy

```
(A B)
(ALADIN BERNARDIN CINTORIN)
```

je vysledkom

```
(BERNARDIN CINTORIN)
```

Oba zoznamy prezerame sucastne dovtedy, kym nenajdeme koniec prveho zoznamu. Vysledkom je dosial neprezreta cast druheho zoznamu.

```
/* lispovske zapisy uvadzame v tvare pre mikroLISP */
(de rest (jeden druhy funkcia)
  (cond ((null jeden)(funkcia))
        (t (rest (cdr jeden)(cdr druhy) '(lambda()druhy) ))))
```

Sledujme teraz vyhodnotenie formy

```
(rest '(ano nie ) '(citron pomaranc banan) '(lambda()nil))
```

Ide o volanie funkcie REST, ktora ma tri (formalne) parametre. Je volana s uvedenymi hodnotami skutocnych parametrov (argumentov) (kontext 1):

```
jeden:    (ano nie)
druhy:    (citron pomaranc banan)
funkcia:  (lambda()nil)
```

Pretoze prvy argument nie je prazdny zoznam, dojde k rekurzivnemu volaniu funkcie REST s argumentami (kontext 2):

```
jeden:    (nie)
druhy:    (pomaranc banan)
funkcia:  (lambda () druhy)
```

(Vsimnime si, ze povodny funkcionalny argument sa stal irelevantny.) Pretoze prvy argument opat nie je prazdny zoznam, vola sa opat funkcia REST s argumentami (kontext 3):

```
jeden:    ()
druhy:    (banan)
funkcia:  (lambda () druhy)
```

Pretoze teraz je prvy argument prazdny zoznam, vyhodnoti sa forma (funkcia), t.j. (lambda () druhy) bez argumentov. Jej hodnotou je (banan), t.j. hodnota druheho argumentu funkcie rest v tomto tretom volani. Tuto hodnotu vracia funkcia rest.

Uvedeny vysledok je iny nez sme zamyslali. V druhom volani funkcie rest dochadza k vyhodnoteniu formy:

```
(rest (cdr jeden) (cdr druhy) '(lambda () druhy))
```

pricom hodnota premennej druhy je (pomaranc banan). Nasa predstava bola taka, ze ak sa v ramci vyhodnocovania tejto formy t.j. tohto volania funkcie rest, bude vyhodnocovat funkcia uvedena ako treti argument, bude v nej mat premenna druhy hodnotu aku ma v momente tohto volania funkcia rest. Chceli by sme, aby sa funkcia (lambda () druhy) vyhodnocovala v kontexte 2 a nie v kontexte 3.

Parameter druhy predstavuje z hladiska funkcionalneho argumentu globalnu premennu. Na stanovenie hodnoty takejto premennej mozno vo vseobecnosti zvolit viacero strategii. Podla jednej strategie sa stanovuje hodnota kazdej globalnej premennej podla kontextu, platneho v momente odovzdania funkcionalneho argumentu funkcionalu, t.j. v momente volania funkcionalu. Podla inej strategie sa stanovuje hodnota kazdej globalnej premennej podla kontextu platneho v momente volania funkcionalneho argumentu. V jazyku LISP sa zvycajne ako standardny sposob pouziva druha strategia (dynamicke viazanie). V nasom pripade by to zodpovedalo pouzitiu kontextu 3. Programator vsak ma moznost vyziadat si pouzitie prvej strategie (staticke viazanie). Dosiahne to tym, ze na citovanie funkcionalneho argumentu pouzije namiesto formy QUOTE formu FUNCTION. V nasom priklade by sa vtedy pouzil kontext 2.

Upravime v tomto zmysle definiciu funkcie REST.

```
->(de restf (jeden druhy funkcia)
       (cond ( (null jeden) (funkcia) )
             ( t  (restf (cdr jeden)
                         (cdr druhy)
                         (function (lambda () druhy)) ) )
       ))
:restf
```

Vysledkom vyhodnotenia formy (restf '(ano nie) '(citron pomaranc banan) (function (lambda () nil)))
bude tentoraz zoznam
(pomaranc banan)
co je v sulade s nasim zamyslom.

FUNKCIE PRE MANIPULACIU SO ZOZNAMOM VLASTNOSTI - PLISTOM.

Uz bolo povedane, ze car-cast CCB reprezentujucej atom, je smernikom na P-list, zoznam vlastnosti. Samotny P-list je zoznam s takouto strukturou:

(ind1 hodn1 ind2 hodn2 ..... indx hodnx)

kde indi je indikator (meno) vlastnosti (v microLISPe vzdy atom) a hodni je hodnota vlastnosti s indikatorom indi.
Okrem svojej hodnoty (dynamicka vazba cez NAME-STACK) moze mat kazdy atom lubovolny pocet vlastnosti, ktore s hodnotou nijako nesuvisia. Kazda vlastnost ma svoj indikator - meno vlastnosti (musi byt atom) a svoju hodnotu (lubovolny S-vyraz). Vsetky vlastnosti su obsiahnute v Pliste.

Niektore vlastnosti maju specialnu funkciu:

A - vlastnost s indikatorom APVAL:
    Ak ma atom vo svojom Pliste tuto vlastnost, jedna sa o tzv. staticku premennu. Pri hladani hodnoty atomu sa nepouzije dynamicka vazba cez NS (nastavena napr. pomocou SETQ), ale vzdy hodnota vlastnosti APVAL. Toto trva, kym sa vlastnost s indikatorom APVAL nachadza v Pliste.
    Na nastavenie statickeho charakteru atomu su k dispozicii 2 funkcie - analogie SET a SETQ.
    CSET - funkcia dvoch argumentov, (cset X Y) zaradi do Plistu atomu XX vlastnost APVAL s hodnotou YY. Vracia YY. Ak XX nie je atom, hlasi sa chyba.
    CSETQ - ako CSET, ale X sa nevyhodnocuje a musi byt atom.
B - vlastnost s indikatorom SUBR:
    jej hodnotou je cislo - adresa podprogramu v strojovom kode.
C - vlastnost s indikatorom EXPR:
    jej hodnotou je lambda-vyraz, telo uzivatelom definovanej funkcie. Tato vlastnost sa zaradi do p-listu ako vedlajsi efekt funkcie DE.
Vlastnosti EXPR a SUBR sprostredkuju funkcnu vazbu na atom. Podrobnejsie pozri v prilohe.
D - vlastnost s indikatorom PNAME:
    jej hodnotou je specialny S-vyraz, predstavujuci tlacitelnu podobu identifikatora atomu. Pozri prilohu.

Okrem tychto si moze uzivatel zadefinovat lubovolny pocet dalsich vlastnosti s atomickym indikatorom a lubovolnou hodnotou Pre manipulaciu s Plistom ma k dispozicii taketo funkcie:

GET - funkcia dvoch argumentov, (get X Y) hlada v Pliste atomu XX (musi byt atom) vlastnost s indikatorom YY a vrati hodnotu tejto vlastnosti, alebo nil, ak atom taku vlastnost nema. Vyskusajte si:

```
->(setq a '(atom a))
:(atom a)
->a
:(atom a)
->(csetq a '(static a))
:(static a)
->a
:(static a)
->(get 'a  'apval)
:(static a)
->(de a (x) (car x))
:a
->(get 'a 'expr)
:(lambda (x) (car x))
->(a '(1 2))
:1
->(remprop 'a 'apval)
:t
->a
:(atom a)
```

PUTPROP - funkcia troch argumentov, umoznujuca zavedenie vseobecnej vlastnosti. (putprop X Y Z) umiestni do P-listu atomu XX (musi byt atom) vlastnost s indikatorom ZZ (musi byt atom) a hodnotou vlastnosti YY. Ak vlastnost s indikatorom YY uz existovala, zmeni sa jej hodnota. !!! Pozor - lahko mozno znicit standardne vlastnosti - predefinovanim vlastnosti EXPR stratite svoju funkciu, zmena PNAME sposobi nepredpovedatelne chovanie systemu a zmena vlastnosti SUBR ma takmer vzdy fatalne dosledky. Funkcia vrati YY ako svoju hodnotu.

REMPROP - funkcia 2 argumentov, (remprop X Z) odstrani z Plistu atomu XX vlastnost s indikatorom ZZ. XX aj ZZ musia byt atomy, inac sa hlasi chyba. Ak funkcia nasla a odstranila vlastnost ZZ vracia t, inac nil.

```
->(putprop 'a '(hodnota 1) 'ind1)
:(hodnota 1)
->(get 'a 'ind1)
:(hodnota 1)
->(remprop 'a 'ind2)
:nil
->(remprop 'a 'ind1)
:t
->(get 'a 'ind1)
:nil
```

Dalsie dve funkcie su 'zovseobecnenim' PUTPROP a REMPROP pre pracu so zoznamom atomov sucastne.

DEFLIST - funkcia dvoch argumentov (deflist X Y), XX musi byt zoznam tvaru ((atom1 hodnota1) (atom2 hodnota2) ....), YY musi byt atom. Pre kazdu dvojicu (atomi hodnotai) sa do P-listu atomu atomi zaradi do Plistu vlastnost s indikatorom YY a hodnotou vlastnosti hodnotai. Vracia nil.

REMFLAG - funkcia dvoch argumentov. (remflag X Y) - XX musi byt linearny zoznam, obsahujuci len atom a YY musi byt atom. Z Plistu kazdeho atomu obsiahnuteho v XX sa odstrani vlastnost s indikatorom YY.

POZOR - znovu varujeme pred odstranenim vlastnosti PNAME !!!

FUNKCIE 'CHIRURGIE PAMATI'

Zahrnaju 3 funkcie, ktore na rozdiel od predchadzajucich, sposobuju fyzicke prepisanie smernikov a tym zmenu v strukturach. Pri ich pouzivani treba postupovat opatrne a premyslene - pri troche stastia alebo hlbsich vedomosti mozno lahko stratit oblist, name stack alebo znicit system.

RPLCA, RPLCD
(rplca X Y) - car XX sa prepise smernikom na YY
(rplcd X Y) - cdr XX sa prepise smernikom na YY

Priklady destruktivneho efektu funkcii:

(rplcd 'a 'c) - odstrani z oblistu vsetky atomy, nachadzajuce sa medzi atomom a a atomom c. Referencie na ne sa v dalsom vypocte mozu stat nedefinovanymi.

(rplca 'a 'c) - znici Plist atomu a - pri pokuse napr. o hladanie funkcnej vazby atomu a moze system spadnut - novy 'Plist' nemusi mat predpokladanu strukturu.

CONCAT - funkcia dvoch argumentov. (concat X Y) vykonava podobnu funkciu ako APPEND, ale nekopiruje svoje argumenty. XX a YY sa fyzicky zretazia a vysledok bude novou hodnotou X.
Nech a ma honotu (a b) a b = (c d).

```
->(append a b)
:(a b c d)
->a
:(a b)
->b
:(c d)
->(concat a b)
:(a b c d)
->b
:(c d)
->a
:(a b c d)
```

DALSIE FUNKCIE

Manipulacia s oblistom:

OBLIST - (oblist) poskytne vypis vsetkych atomov znamych interpretu. Vypisuje sa len meno atomu. Vrati nil.

GETOB - (getob X), XX musi byt atom. Do NAME-STACKu sa zaradi vazba atomu XX na aktualny oblist. (hodnotou atomu bude momentalny sta Oblistu, pri vypise sa vypise len prvy atom)

SETOB - (setob X) nastavi sa novy oblist, zacinajuci od atomu XX. Pozor na neuvazene pouzitie.

REMOB - (remob X) odstrani atom XX z oblistu.
Vo vsetkych funkciach sa hlasi chyba, ak XX nie je atom.

Pozn: Po odstraneni atomu z oblistu ho budu vsetky smerniky nan nadalej referencovat, kym bude bunka, reprezentujuca ho, fyzicky existovat. Po cisteni pamati nebudu hodnoty referencujucich smernikov definovane (neodhadnutelne dosledky). Na druhej strane ak po zruseni atomu zavedieme novy atom s rovnakym menom, porusi sa jedinecna reprezentacia atomov a funkcie pouzivajuce napr. EQUAL nemusia fungovat. Majte to pri pouzivani funkcii REMOB a SETOB na pamati !

POP - funkcia bez argumentov. (pop) skrati oblist o prvy prvok (najlavejsi). Ma vyznam pri nedostatku pamati.
Pozn: Interpret aj pre vyhodnotenie formy POP potrebuje niekolko volnych CCB. Ak ani tie nie su k dispozicii, pozri BREAK mod v prilohe.

CLS - zmazanie obrazovky. (cls) zmaze pracovnu cast obrazovky a vrati nil.

RND - generator pseudonahodnych cisiel. Vo verzii 1.0 nefunguje

PCH - funkcia jedneho argumentu. (pch X) tlaci znak s ASCII kodom XX. Ak XX nie je cislo, hlasi sa chyba.

FUNKCIE VSTUPU A VYSTUPU

Vytvorene programy mozno zaznamenat na pasku standardnym sposobom. Subory maju standarnu hlavicku a su typu CODE. Zaznamenava sa cely aktualny stav interpreta (cely OBLIST a NAME-STACK). Po nahrati suboru z pasky teda ziskavame kopiu stavu interpreta z okamihu zaznamu. Vo verzii 1.0 nie je mozne urobit 'MERGE' suboru na paske s aktualnym stavom interpreta.

SAVE - (save X) zaznamena stav interpreta na pasku pod menom X. X sa nevyhodnocuje a musi to byt atom - inac nasleduje cybove hlasenie 'X IS NOT A FILENAME'. X musi byt uvedene.

Pretoze ZVP vyuziva adresny priestor az po RAMTOP-1000H, treba pred zaznamom urobit kompakciu ZVP - vytvorit suvisly blok obsadenych CCB, pri zachovani vsetkych vzajomnych vazieb. Tato kompakcia moze trvat radovo az minuty a je signalizovana vypisom '[WAITING FOR COMPACTER]'. Potom sa objavi hlasenie 'Start tape then press any key ..' a po stlaceni lubovolnej klavesy sa subor zaznamena.

LOAD, VERIFY - maju najviac jeden argument. Ak je argument uvedeny, nahra/ verifikuje sa subor s menom X. Ak argument chyba nahra/ verifikuje sa prvy subor.
Pozn: Ak sa nahra subor, ktory nebol vytvoreny microLISPom, moze system spadnut. (Nic tym vsak nestratite, kedze momentalny stav interpreta sa LOADom zmeni).

STACK - nAstavenie hlbky systemoveho zasobnika.
Typickym programovacim stylom v jazyku LISP je rekurzia. Kedze aj velka cast strojovych podprogramov je pisana rekurzivne, rezervuje system inicialne 4096 bytov pre systemovy zasobnik. Uzivatel ma moznost to zmenit - (stack X) nastavi velkost systemoveho zasobnika na XX bytov. Ak XX nie je cislo, chybove hlasenie. Po vyhodnoteni formy STACK sa prevedie cistenie pamati a kompakcie ZVP. Ak by novy zasobnik mal zasahovat do oblasti obsadenej po kompakcii pouzitymi CCB, nic sa nevykona. V opacnom pripade sa nastavi nova velkost zasobnika a vytvori sa nova ZVP zacinajuca za oblastou pouzitych CCB a konciaca na adrese RAMTOP - XX.

Pozn: Neodporucame prilis zmensovat oblast pre zasobnik - pripadne "pretecenie" zasobnika do ZVP by mohlo mat neprijemne dosledky.

LISPOVSKY EDITOR.

Uz sme povedali ze mikroLISP poskytuje moznost editovat priamo S-vyrazy. Kazdy SV mozno reprezentovat binarnym stromom - a z tejto reprezentacie budeme vychadzat. (edi X) vyhodnoti svoj argument, zobrazi XX a caka na vstup. Po strome postupujeme dvomi klavesami 'P' - vezme pravy podstrom (spristupni cdr-cast SV), klavesa 'O' zasa lavy podstrom (car-cast SV). Tymto sposobom sa dostaneme na uroven podstromu, ktory chceme modifikovat (pri kazdom pohybe po hranach stromu sa vypise aktualny podstrom). Klavesou CS + W prejde interpret do vstupneho rezimu, nacita S-vyraz, prelozi do vnutornej reprezentecie a zaradi na miesto aktualneho podstromu v okamihu stlacenia CS+W. Prechadzanie stromom mozno kedykolvek ukoncit klacesou ENTER - po jej stlaceni sa vypise cela editovana struktura XX a vrati sa XX.

Nie je mozne dostat sa 'za' listy stromu - ked pri prechadzani stromom je niektory podstrom atom, pri pokuse o 'P' alebo 'O' sa editovanie ukonci. Takisto nemozno prepisat priamo koren stromu - aspon raz musi byt pouzita 'P' alebo 'O'.

Ak je XX atom, po 'O' mozme editovat jeho Plist - to je jediny pripad ked editovanie nekonci pri narazeni na atom.

POZOR - nepokusajte sa editovat hodnotu vlastnosti PNAME - tento S-vyraz ma specialnu strukturu (pozri prilohy).

Nech a = (s d f (g h)) a chceme nahradit podzoznam (g h) zoznamom (g j h).

Dialog so systemom	Riadiace klavesy
->(edi a)	
:(s d f (g h))	P
:(d f (g h))	P
:(f (g h))	P
:((g h))	O
:(g h)	CS+W

teraz system s vypisom '->' caka na zadanie noveho SV

->(g j h)	po ENTER nasleduje
:(s d f (g j h))	
:(s d f (g j h))	

Zopakovanie noveho zoznamu sposobil system vypisom vratenej hodnoty funkcie edi. Zmeny su trvale :

```
->a
:(s d f (g j h))
```

TRACE - funkcia bez argumentov. (trace) invertuje priznak trasovania . Ak je trasovanie zapnute, vypisuju sa navazovane hodnoty na formalne argumenty. Trasovanie v mikroLISPe je velmi zjednodusene napr. oproti LISPu 1.5 - nevypisuju sa mena volanych funkcii ani navazovane hodnoty na formalne argumenty u funkcii typu SUBR. Tieto obmedzenia vyplyvaju zo sposobu implementacie. Napriek tomu casto aj tato ochudobnena informacia pomoze odhalit chybu. TRACE vrati vzdy t.

APPLY - funkcia dvoch argumentov. (apply F Z) ma takyto ucinok:

FF musi referencovat funkciu a ZZ musi byt zoznam argumentov. Funkcia FF sa aplikuje na argumenty v ZZ. Vrati vysledok podla (eval (cons F Z)).

Priklad treba scitat vsetky cisla obsiahnute v zozname a - napr. nech a = (1 2 3). Funkcia plus pozaduje n ciselnych argumentov a pokus o (plus a) sposobi chybove hlasenie. Ale:

```
->(apply 'plus a)
:6
```

vrati spravny vysledok.

To je vsetko k popisu funkcii. Neuvazene pouzitie niektorych moze mat fatalne dosledky (obycajne znovunahranie systemu) - na taketo moznosti sme upozornovali pri popise. (System LISP je mozne velmi lahko znicit jeho vlastnymi legalnymi prostriedkami) Preto odporucame "nebezpecne funkcie" zo zaciatku nepouzivat vobec, a potom po dokladnom uvazeni. Tie funkcie, ktore su potrebne pri vyucbe programovania, boli navrhovane podla zasady "foolproof" a mali by na nekorektne pouzitie reagovat chybovym hlasenim. "Nebezpecne funkcie" boli zahrnute do systemu pre uplnost, a aby sa v niektorych pripadoch obisli pamatove obmedzenia 8 bitovych mikropocitacov.

Poslednu z popisanych funkcii NEODPORUCAME POUZIVAT VOBEC !!! Bola zahrnuta do mikroLISPu pre pohodlnost - ako uzitocny nastroj pri ladeni systemu - povodne pre spolupracu s debuggerom

CALL - "nelispovska funkcia" - priame volanie strojovych rutin. Pri dokonalej znalosti architektury systemu je mozne obist vacsinu obmedzeni a chybovych hlaseni (vyplyvajucich z urcenia interpreta) priamym volanim podprogramov v strojovom kode. Funkcia CALL ma 2 modifikacie:

->(call A X Y Z)

vola podprogram na adrese AA. Registrovy par HL obsahuje smernik na XX, DE na YY a BC na ZZ. Ak AA nie je cislo, hlasi sa chyba.

->(call A) ako predchadzajuci pripad, no hodnoty registrov nie su definovane.

Kazda chyba pri pouziti tejto funkcie ma VZDY fatalne nasledky. Pozn: Ak Vam pojde mikroLISP na nervy, skuste (call 0).

1987, BRATISLAVA, EF SVST - Katedra pocitacov.

POST SCRIPTUM: Znamy Murphyho zakon hovori, ze v kazdom programe je aspon jedna chyba. Autori sa pokusili prelstit tento zakon zamernym zaradenim nefungujucej funkcie RND (resp. fungovala BY, KEBY neboli zakazane prerusenia), je velmi pravdepodobne, ze v systeme su chyby o ktorych nevieme. Boli by sme vdacny za upozornenie na chyby, ktore sa objavia pri praci s mikroLISPom. Adresa:

Ing. Miro ADAMY
EF.SVST Kat. pocitacov
Mlynska dolina
812 18 BRATISLAVA

Natištěno pro ZO Svazarm Karolinka, srpen 1989.