Poliechnika Poznańska

Instytut Automatyki

Marek Barłóg, Barbara Begier, Henryk Katulski

Jan Kniat, Zygmunt Liszyński, Jacek Martinek,

Maciej Stroinski, Witold Wojciechowski

SYSTEM PROGRAMOWANIA MINIKOMPUTERA K-202 W JEZYKU L I S P 1.5

INSTRUKCJA PROGRAMISTY

Wykonano na zlecenie Zakładu Doświadczalnego

Minikomputerów

Instytutu Maszyn Matematycznych

w Warszawie

Poznań, luty 1974

Spis treści

	Str.
Wstęp	3
1. Programowanie w języku LISP	5
1.1. Dane języka LISP	5
1.2. Funkcje elementarne	10
1.3. Język źródłowy, czyli metajęzyk	14
1.4. Przekształcanie wyrażeń metajęzyka w język danych	19
1.5. Baza danych systemu i funkcje systemowe	23
1.6. Interpretacja wyrażeń funkcyjnych	31
1.7. Kompilacja wyrażeń funkcyjnych	32
2. Współpraca z monitorem systemu	34
2.1. Uwagi wstępne	34
2.2. Uruchomienie systemu LISP	34
2.3. Dokonywanie przerwań	35
2.4. Wprowadzanie zadania do systemu	37
2.5. Przykład uruchomienia systemu i wykonania zadania	38
2.6. Rozszerzenie obszaru pamięci zajmowanego przez dane	40
3. Wprowadzanie i wyprowadzanie danych	42
3.1. RATOM	42
3.2. READ	48
3.3. READCH	49
3.4. PRINT	49
3.5. PRIN1	50
3.6. PRINCH	50

3.7. OUTBUF 51

3.8. FORMAT 51

3.9. INPUT i OUTPUT 55

3.10. START 55

4. Funkcje systemowe 56

5. Zasady interpretacji i kompilacji 105

5.1. Interpretacja funkcji zapisanych jako S-wyrażenia 105

5.2. Kompilacja i wykonywanie funkcji przetłumaczonych 118

6. Opis realizacji systemu 121

6.1. Podział pamięci 121

6.2. Struktury danych w pamięci 122

6.3. Realizacja programów. Makroinstrukcje 126

6.4. Odzyskiwanie wolnej pamięci 127

Wykaz sygnalizacji błędów 129

Literatura 135

Indeks 136

Wstęp

Język LISP jest przeznaczony przede wszystkim do przetwarzania danych symbolicznych i różni się od powszechnie znanych języków programowania kilkoma specyficznymi cechami. Wśród nich sposób zapisu programów może być najbardziej szokujący dla programistów nie znających LISP-u.

Autorem języka LISP jest John McCarthy, który także, wraz z grupą współpracowników, dokonał pierwszych jego implementacji. Przy opracowaniu systemu dla K-202 wykorzystano przede wszystkim materiał zawarty w najbardziej znanym źródle: LISP 1.5 Programmer's Manual, The MIT Press, 1962.

Znaczenie LISP-u jest duże i niektórzy autorzy /np.T.E.Cheatham/ uważają go na równi z ALGOL-em 60 za najważniejszy z powstałych dotychczas języków programowania. LISP wykorzystywano wielokrotnie jako podstawę do konstrukcji systemów specjalistycznych przez rozszerzanie go, lub zanurzanie w nim innych języków. Autorzy wielu języków opierali się w swych konstrukcjach na LISP-ie, a koncepcje stanowiące jego podstawę wywarły duży wpływ na prace teoretyczne dotyczące semantyki języków programowania.

Można wyodrębnić dwie obszerne dziedziny, w których stosuje się LISP. Są to zastosowania maszyn cyfrowych do "matematyki nienumerycznej" oraz programowanie heurystyczne.

W pierwszej z tych dziedzin stosowano LISP do różniczkowania symbolicznego, całkowania nieoznaczonego, upraszczania wyrażeń algebraicznych, dowodzenia twierdzeń.

W oparciu o LISP tworzono duże systemy ułatwiające przeprowadzanie obliczeń symbolicznych, np.MATHLAB, w którym istnieją programy dokonujące, oprócz już wymienionych działań, także obliczenia prostego i odwrotnego przekształcenia Laplace'a oraz rozwiązywania układów równań różniczkowych liniowych ze stałymi /symbolicznymi/ współczynnikami. Ponadto pisano w LISP-ie programy analizy obwodów elektrycznych, analizy i syntezy automatów skończonych oraz stosowano go do manipulacji informacją graficzną.

W programowaniu heurystycznym wykorzystywano LISP do pisania dużych i skomplikowanych programów o różnorodnym zastosowaniu. Były to programy dowodzące twierdzenia metodami heurystycznymi, przeprowadzające wnioskowania indukcyjne i przez analogie oraz grające w gry.

Dużą grupę zastosowań LISP-u stanowią systemy odpowiadające na pytania w języku naturalnym i przeprowadzające proste dedukcje. Stosuje się go także do programowania robotów.

System programowania w języku LISP dla K-202 został opracowany przez grupę pracowników Środowiskowego Ośrodka Informatyki oraz Instytutu Automatyki Politechniki Poznańskiej na zlecenie Zakładu Doświadczalnego Minikomputerów przy Instytucie Maszyn Matematycznych w Warszawie

Instrukcja składa się z 6 punktów oraz wykazu sygnalizacji błędów i indeksu. W punktach 1-3 i 5 omówiono zasady programowania w zrealizowanej wersji LISP-u oraz zasady współpracy programisty z systemem. Punkt 4 zawiera alfabetyczny wykaz funkcju systemowych, a punkt 6 informacje o ważniejszych szczegółach realizacji systemu.

1. Programowanie w języku LISP

1.1. Dane języka LISP

Podstawowymi elementami, z których konstruuje się dane są symbole atomowe i liczby.

Symbole atomowe. Symbol atomowy z punktu widzenia programisty jest to sznur złożony z liter i cyfr o długości nie większej niż 64 znaki, rozpoczynający się literą.

Przykładami symboli atomowych są sznury

ALA

B15

COTOBEDZIE

Symbol atomowy jest niepodzielny. Wprowadzony do systemu pozostaje w nim na stałe. Identyczne sznury pisane przez programistę odnoszą się zawsze do jednego symbolu atomowego.

Dowolny tekst może być także traktowany jak symbol atomowy, jeśli zastosuje się specjalny zapis omówiony w punkcie 3.

Liczby. Rozróżnia się trzy typy liczb: całkowite, ósemkowe i zmiennoprzecinkowe. Typy te są odrębnie reprezentowane w systemie.

Liczba ósemkowa rozpoczyna się cyfrą 0, po której może następować do 6 pozycji znaczących. Zapisuje się na nich cyfry ósemkowe 0,1,...,7, z tym ograniczeniem, że największą akceptowaną liczbą ósemkową jest 0177777. Wiąże się to z reprezentacją liczby przez jedno 16-bitowe słowo maszyny.

Przykład liczb ósemkowych:

0 021 0326 077777

Liczba całkowita składa się z cyfr dziesiętnych 0,1,...,9 poprzedzonych znakiem + lub -. Znak + można opuścić, jeśli najstarsza cyfra liczby jest różna od 0. Liczba całkowita może składać się co ~~najmniej~~ najwyżej z 5 cyfr, a jej wartość winna być zawarta w przedziale od -32768 do +32767.

Przykład liczb całkowitych:

+0 21 -326 32000

Liczba zmiennoprzecinkowa jest to liczba, która oprócz części całkowitej posiada część ułamkową (w szczególnym przypadku równą zero). Przedstawia się ją w dwóch postaciach: normalnej m.n lub półlogarytmicznej m.nEp, gdzie m,n,p są liczbami całkowitymi, przy czym n jest liczbą bez znaku, m jest częścią całkowitą liczby, n częścią ułamkową, natomiast p wykładnikiem potęgi o podstawie dziesiętnej, przez którą liczba m.n ma zostać pomnożona celem uzyskania jej rzeczywistej wartości. Liczba zmiennoprzecinkowa zajmuje 3 komórki pamięci: w pierwszej z nich jest pamiętana cecha wraz ze znakiem, w pozostałych dwóch mantysa i jej znak. Mantysa posiada około 10 cyfr dziesiętnych, cecha ma zakres $-2\uparrow32768$, $+2\uparrow32767$.

W liczbie zmiennoprzecinkowej posiadającej część całkowitą równą zero można pominąć tę część, o ile liczba zostanie poprzedzona znakiem. Zerowa część ułamkowa przy przedstawianiu liczby w postaci normalnej nie może być pominięta.

Przykład liczb zmiennoprzecinkowych:

5.4 5.0 0.1 5.4E+0 5.0E+1 1.E-1

Postać niedozwolona .1 5. .4E5

Pary kropkowe. Z symboli atomowych i liczb jako elementów tworzy się struktury danych zwane wyrażeniami symbolicznymi lub krótko S-wyrażeniami.

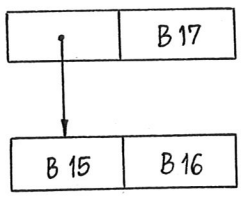
Do zbioru S-wyrażeń należą przede wszystkim symbole atomowe i liczby jako elementy podstawowe. Ponadto z dwóch S-wyrażeń można utworzyć tzw.parę kropkową, która jest również S-wyrażeniem. Parę kropkową zapisuje się następująco

(< S-wyrażenie > . < S-wyrażenie >)

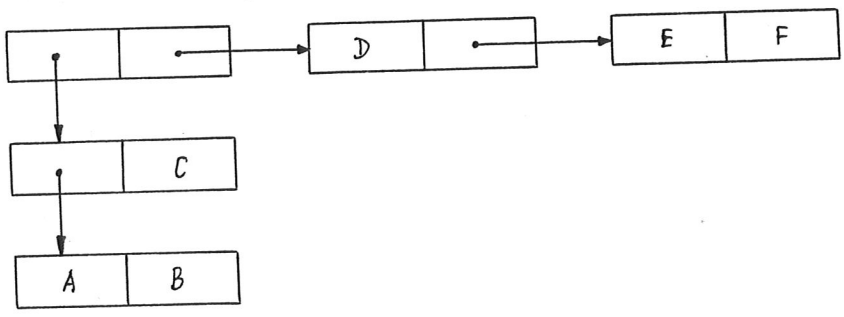
Przykładami S-wyrażeń są zatem

ALA (B15.B16) +153 (((1. 2). 3). A8)

W odczytaniu struktury złożonego S-wyrażenia może okazać się pomocne stosowanie jego reprezentacji graficznej. Zamiast nawiasów i kropki rysuje się pudełko składające się z 2 części i zamiast wkładać pudełka do pudełek umieszcza się je na zewnątrz siebie dodając strzałki. Na przykład, zamiast pisać ((B15.B16).B17) sporządza się rysunek



Analogiczne wyrażenie (((A.B).C).(D.(E.F))) będzie posiadać następującą reprezentację graficzną



Z rysunków widać, że złożone dane LISP-u mają strukturę drzew binarnych.

Listy. S-wyrażenia pewnego szczególnego rodzaju nazywa się listami i zapisuje się w sposób uproszczony w stosunku do notacji kropkowej.

Szczególną rolę pełni symbol atomowy NIL reprezentujący listę pustą. Listę pustą reprezentuje także zapis ( ) równoważny symbolowi atomowemu NIL. Najprostszą listą niepustą jest para kropkowa postaci

(< S-wyrażenie > . NIL)

którą zapisuje się skrótowo

(< S-wyrażenie >)

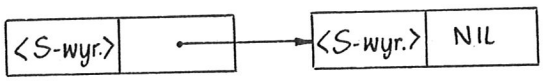
Jest to lista z jednym elementem. Analogicznie wyrażenie

(< S-wyrażenie > . (< S-wyrażenie > . NIL))

zapisuje się skrótowo w postaci

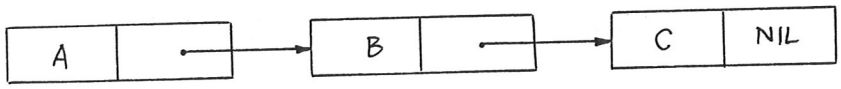
(< S-wyrażenie > < S-wyrażenie >)

i nazywa listą dwuelementową. Reprezentacja graficzna jest natomiast dla obu zapisów wspólna

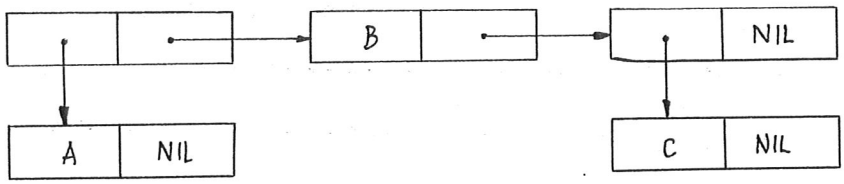


W podobny sposób zapisuje się listy z liczbą elementów większą niż 2. Poniżej podano przykłady S-wyrażeń w notacji kropkowej i listowej oraz ich reprezentację graficzną

(A . (B . (C. NIL))) (A B C)



((A . NIL) . (B . ((C . NIL) . NIL))) ((A) B (C))



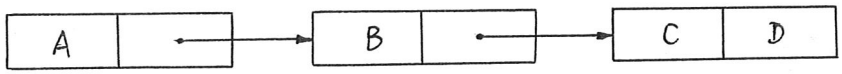
Zezwala się na stosowanie także notacji mieszanej, "listowo-kropkowej", np.

(A B C . D)

jest skrótowym zapisem wyrażenia

(A . (B . (C . D)))

o następującej reprezentacji graficznej



S-wyrażeniem jest zatem element zbioru określonego produkcjami gramatycznymi:

<S-wyrażenie> ::= <symbol atomowy> |
<liczba> |
(<elementy>)

<elementy> ::= <puste> |
<S-wyrażenie> . <S-wyrażenie>
<S-wyrażenie> <elementy>

1.2. Funkcje elementarne

Omówimy obecnie zestaw funkcji elementarnych LISP-u służących do przetwarzania S-wyrażeń. W przykładach, ilustrujących działanie funkcji, wartości jej argumentów będziemy zapisywać w kwadratowych nawiasach i oddzielać od siebie średnikami.

Konstruktor CONS. Jest to funkcja dwuargumentowa konstruująca z dwóch S-wyrażeń parę kropkową, np.

CONS [A ; B] = (A.B)

CONS [(A.B); C] = ((A.B).C)

CONS [A ; NIL] = (A)

CONS [A ; (B)] = (A B)

Selektory CAR i CDR. Są to funkcje jednoargumentowe wyciągające z pary kropkowej jej pierwszy element (CAR) lub drugi element (CDR), np.

CAR [(A.B)] = A

CDR [(A.B)] = B

CAR [(A)] = A

CDR [(A)] = NIL

CAR [(A B C)] = A

CDR [(A B C)] = (B C)

Wartość funkcji CAR i CDR dla argumentu będącego symbolem atomowym lub liczbą jest nieokreślona.

Predykaty ATOM, EQ, NUMBERP i EQUAL. Funkcja, której wartością jest albo prawda albo fałsz nazywa się predykatem.

W systemie LISP dla K202 prawdę reprezentuje symbol atomowy T, zaś fałsz symbol atomowy NIL.

ATOM jest predykatem jednoargumentowym, testującym, czy argument ten jest symbolem atomowym:

ATOM[A] = T

ATOM[(A.B)] = NIL

ATOM[15] = NIL

Podobnie NUMBERP bada, czy argument jest liczbą:

NUMBERP[1.5] = T

NUMBERP[A] = NIL

NUMBERP[(A. 1.5)] = NIL

Predykat EQ jest w zasadzie przeznaczony do badania równości dwóch symboli atomowych i powinien być nieokreślony, jeśli choćby jeden z argumentów nie jest symbolem atomowym. W realizacji dla K202 predykat ten porównuje adresy reprezentacji wewnętrznej argumentów, zatem jego wartość jest zawsze określona. Mamy, na przykład

EQ[A;B] = NIL

EQ[A;A] = T

Predykat EQUAL służy do badania równości dwóch dowolnych S-wyrażeń:

EQUAL [A;B] = NIL

EQUAL [5;5] = T

EQUAL [A;(A.B)] = NIL

EQUAL [(A.B);(A.B)] = T

EQUAL [03 ; 03] = T

Predykaty OCTALP, INTEGERP i FLOATP. Są to predykaty jednoargumentowe, badające czy argument jest liczbą odpowiedniego typu: ósemkową, całkowitą czy zmiennoprzecinkową. Przykłady:

```
OCTALP [0777] = T
OCTALP [1.5]  = NIL
INTEGERP [1]  = T
FLOATP [1]    = NIL
FLOATP [1.5]  = T
```

Funkcje i predykaty arytmetyczne. Do zestawu funkcji elementarnych wchodzi wiele funkcji i predykatów arytmetycznych, które omówiono systematycznie w punkcie 4. Są to funkcje: PLUS, TIMES, MAX, MIN, DIFFERENCE, DIVIDE, QUOTIENT, REMAINDER, EXPT, ADD1, SUB1, ABS, MINUS, RECIP, SIN, COS i ARCT, ENTIER, oraz predykaty: ONEP, ZEROP, MINUSP, GREATERP i LESSP.

Spośród nich funkcje QUOTIENT i REMAINDER obliczające odpowiednio iloraz oraz resztę z dzielenia całkowitego operują tylko na liczbach całkowitych; dziesiętnych i ósemkowych.

Pozostałe funkcje i predykaty operują na liczbach dowolnego typu, dopuszczając wprowadzanie argumentów o wartościach różnych typów do jednej funkcji. W przypadku pojawienia się mieszanych wartości argumentów, z których chociaż jedna była liczbą zmiennoprzecinkową, wynik będzie liczbą zmiennoprzecinkową. Jeśli funkcja operuje na liczbach całkowitych dziesiętnych oraz ósemkowych, to wynik będzie liczbą ósemkową.

Funkcje logiczne. Są to funkcje LOGOR, LOGAND, LOGXOR i LEFTSH operujące na liczbach ósemkowych lub całkowitych dziesiętnych. Liczby te są traktowane jako 16 bitowe wektory binarne.

Funkcje RPLACA i RPLACD. Są to funkcje dwuargumentowe o specyficznym działaniu. Wartością pierwszego argumentu musi być para kropkowa, a drugiego dowolne S-wyrażenie. Funkcje te posiadają wyniki formalne, jakie można uzyskać przez stosowanie funkcji CONS, jednak faktyczny efekt ich działania jest inny. bowiem zmienia się reprezentacja danej w pamięci systemu. Formalnie

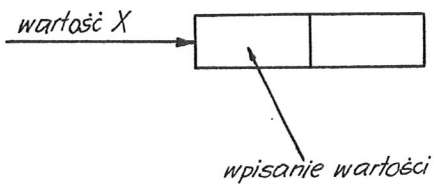
RPLACA[X ; Y]= CONS[Y ; CDR[X]]

RPLACD[X ; Y]= CONS[CAR[X]; Y]

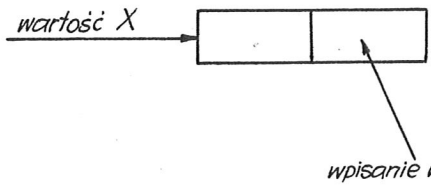
gdzie X i Y są zmiennymi o wartościach akceptowanych przez funkcje. Jednak przy wykonaniu funkcji CONS tworzy się w pamięci nowa reprezentacja wyniku - odpowiedniej pary kropkowej, natomiast przy wykonywaniu funkcji RPLACA i RPLACD zmienia się istniejąca reprezentacja argumentu X, tzn. dana będąca poprzednią wartością argumentu X przestaje istnieć.

Poniższe rysunki wyjaśniają efekt działania tych funkcji

RPLACA[X ; Y]



RPLACD[X ; Y]



1.3. Język źródłowy czyli metajęzyk

W LISP-ie programy przetwarzania danych tworzy się z funkcji elementarnych za pomocą pewnych operatorów, które obecnie omówimy.
Każdy program jest formalnie funkcją, tzn. przed jego wykonaniem należy podać wartość argumentów, a po zakończeniu programu otrzymuje się wartość tej formalnej funkcji jako wynik jej działania. Dlatego dalej będziemy używać w tym samym sensie terminów program i funkcja.

Język źródłowy, w którym zapisuje się funkcje, czyli reguły przetwarzania S-wyrażeń, nazywa się metajęzykiem, bowiem jego wyrażenia nie należą do zbioru danych. Cechą charakterystyczną LISP-u jest reprezentowanie wyrażeń metajęzyka przez S-wyrażenia. W ten sposób metajęzyk, czyli język programów, zostaje zanurzony w język danych i programy mogą być przetwarzane tak samo, jak dane.
W większości realizacji LISP-u wymaga się od programisty, by pisał funkcje wprost w języku S-wyrażeń. W realizacji dla K202 wprowadzono możliwość pisania funkcji bądź w metajęzyku, bądź jako S-wyrażenia.

Dalsza część punktu 1.3 jest poświęcona omówieniu metajęzyka. Omówienie to rozpoczniemy od podania sposobu konstruowania wyrażeń zwanych formami.

1. Nazwy funkcji i zmiennych zapisuje się tak samo, jak symbole atomowe. Aby odróżnić zmienną, np.X od symbolu atomowego X jako wartości stałej, stosuje się "cudzysłów", którym jest operator QUOTE. Zatem QUOTE[X]

oznacza stałą - symbol atomowy X. Nie ma potrzeby używania operatora QUOTE w przypadku liczb oraz wyróżnionych symboli atomowych T i NIL.

2. Aby wyrazić zastosowanie funkcji (np.CONS) do argumentów (np.X i QUOTE[A]) stosuje się zapis składający się z kwadratowych nawiasów i średników. W rozważanym przykładzie będzie to wyrażenie

   CONS[[X]; QUOTE[A]]

   Inne przykłady

   LENGTH[X]

   FF[X ; QUOTE[Y]; Z]

3. Superpozycja funkcji zapisuje się przez złożenie wyrażeń, np. CONS[FF[X; Y; Z]; QUOTE[A]]

4. Wyrażenie warunkowe posiada postać

   $[p_1 \rightarrow e_1 ; p_2 \rightarrow e_2 ; \ldots ; p_n \rightarrow e_n]$

   gdzie wyrażenia $p_1, p_2, \ldots, p_n$ nazywają się warunkami, zaś $e_1, e_2, \ldots, e_n$ są także wyrażeniami (czyli zostały skonstruowane według reguł 1-3 lub są wyrażeniami warunkowymi). Wartościowanie wyrażenia warunkowego odbywa się następująco. Jeśli wartość warunku p_1 jest różna od NIL, to wartością całego wyrażenia warunkowego jest wartość e_1. W przeciwnym razie bada się wartość warunku p_2 i ewentualnie dalszych, aż do znalezienia warunku o wartości różnej od NIL. Jeśli p_i jest pierwszym takim warunkiem, to wartość e_i jest wartością całego wyrażenia warunkowego. Jako przykład omówimy wartościowanie wyrażenia

   [EQ[CAR[X]; QUOTE[A]] → CONS[QUOTE[B]; CDR[X]]; T → X]

Zakładając, że zmiennej X przypisano wartość (A.B)

Warunek

EQ[CAR[X]; QUOTE[A]]

ma wtedy wartość T i wynik wartościowania całego wyrażenia warunkowego powstaje przez obliczenie wartości wyrażenia

CONS[QUOTE[B]; CDR[X]]

czyli jest równy (B.B)

Wyrażenia konstruowane według zasad 1-4 nazywają się formami. Jeśli w formie występują zmienne X, Y,...,Z , to nazywa się ją formą zmiennych X, Y,...,Z.

Z formy tworzy się funkcję przez związanie zmiennych operatorem LAMBDA. Otrzymuje się wyrażenie funkcyjne postaci

LAMBDA[[X; Y; ...; Z]; < forma >]

Zmienne [X; Y; ...; Z] wyszczególnione w kwadratowych nawiasach jako pierwszy argument operatora LAMBDA nazywają się zmiennymi związanymi. Jest to jakby deklarowanie zmiennych, z tym, że istotna jest kolejność, w jakiej je wypisano. Jeśli X pojawia się jako pierwsza zmienna za operatorem LAMBDA, wówczas znaczy to, że zmienna X reprezentuje pierwszy argument funkcji. Podobnie druga zmienna reprezentuje drugi argument, itd. Na przykład z dwóch różnych form

CONS[CAR[X]; Y] i

CONS[CAR[Y]; X]

można utworzyć tę samą funkcję, pisząc

LAMBDA[[X ; Y]; CONS[CAR[X]; Y]] i

LAMBDA[[Y ; X]; CONS[CAR[Y]; X]]

Wyrażeniom funkcyjnym można przypisywać nazwę stosując operator LABEL według składni

LABEL [< symbol atomowy > ; < funkcja >]

Np. LABEL [FF; LAMBDA [[X;Y]; CONS [CAR [X]; Y]]]

Stosowanie operatora LABEL oraz wyrażeń warunkowych pozwala definiować funkcje rekursywne. Można na przykład zdefiniować funkcję, która dla argumentu będącego dowolnie złożonym S-wyrażeniem wylicza pierwszy skrajny symbol atomowy w nim występujący.

Będzie to funkcja

LABEL [FF; LAMBDA [[X]; ATOM [X] → X ; T → FF [CAR [X]]]]]

W wyrażeniu definiującym pojawia się definiowana nazwa FF. Mimo to, funkcja FF jest sensownie określona. Jeśli, na przykład, podstawić jako jej argument S-wyrażenie (A.B), to otrzyma się A jako wynik.

Oprócz stosowania takich rekursywnych definicji można w LISP-ie pisać funkcje w sposób podobny do częściej spotykanych metod zapisu programów. Można tworzyć sekwencję instrukcji do wykonania, etykietować instrukcje w niej występujące i umieszczać instrukcje skoków do miejsc oznaczonych etykietami.

Temu celowi służy specjalny operator PROG wymagający następującej składni

PROG [[< ciąg zmiennych >] ; < ciąg form >]

Zmienne w ciągu zmiennych i formy w ciągu form oddziela się średnikami.

Widać, że operator PROG posiada składnię podobną do składni operatora LAMBDA z tą różnicą, że jako drugi formalny argument PROG-u pojawia się ciąg form zamiast formy. Formy, występujące w tym ciągu i będące symbolami atomowymi, nazywają się etykietami, zaś inne formy tego ciągu noszą nazwę instrukcji.

Zmienne zadeklarowane przez operator PROG nazywają się zmiennymi programowymi. Przed wykonaniem ciągu instrukcji każdej z tych zmiennych przypisuje się wartość NIL. Widać zatem, że kolejność wypisania zmiennych programowych nie odgrywa roli z punktu widzenia semantyki wyrażenia.

Oprócz dotąd omówionych rodzajów form jako instrukcja może wystąpić wyrażenie postaci

⟨symbol atomowy⟩ := ⟨forma⟩

zwane instrukcją podstawienia.

Ponadto do konstruowania instrukcji wykorzystuje się dwa operatory jednoargumentowe GO i RETURN zapisywane tak samo, jak funkcje elementarne, np.

GO[A]

RETURN[CAR[X]]

Operator GO służy do utworzenia instrukcji skoku bezwarunkowego, zaś operator RETURN do wyjścia z ciągu instrukcji z wynikiem będącym wartością argumentu operatora.

Wykorzystanie operatora PROG do konstruowania funkcji wyjaśnimy na przykładzie. Będzie to wyrażenie określające funkcję LENGTH, która służy do obliczania długości listy.

```
LABEL[LENGTH ; LAMBDA[[X]; PROG[[U; V];
                V:= +0 ;
                U:= X  ;
              A;[NULL[U] → RETURN[V]];
                U:= CDR[U];
                V:= ADD1[V];
                GO[A]]]]
```

Semantyka tego zapisu jest następująca. Funkcja o nazwie LENGTH jest funkcją od jednego argumentu, oznaczonego przez X. Utworzono ją wykorzystując operator PROG. Zmiennymi programowymi są U i V. Podstaw O za V, podstaw wartość X za U. Jeśli wartością U jest NIL, to wynikiem funkcji jest wartość zmiennej V. W przeciwnym razie podstaw za U wartość formy CDR[U], podstaw za V wartość formy ADD1[V] (dodawanie jedynki) i skocz do miejsca oznaczonego etykietą A.

1.4. Przekształcenie wyrażeń metajęzyka w język danych

Funkcję napisaną przez programistę w metajęzyku wprowadza się do systemu programem o nazwie READ. Program ten analizuje składnię wyrażenia funkcyjnego i równocześnie tłumaczy je na S-wyrażenie, czyli element zbioru danych. READ przekształca zatem metajęzyk w zbiór danych. Program ten akceptuje wyrażenia metajęzyka określone następującą składnią

<funkcja> ::= <symbol atomowy1>|
LAMBDA[[<ciąg zmiennych>]; <forma>]|
LABEL [<symbol atomowy>; <funkcja>]

<ciąg zmiennych> ::= <puste>|<niepusty ciąg zmiennych>

<niepusty ciąg zmiennych> ::= <symbol atomowy>|
<symbol atomowy>;
<niepusty ciąg zmiennych>

<forma> ::= <symbol atomowy>|
<liczba>|
QUOTE[<S-wyrażenie>]|
<funkcja>[<ciąg argumentów>]|
[<ciąg par form>]|
PROG[[<ciąg zmiennych>]; <ciąg form>]|
<symbol atomowy1> := <forma>|

<argument> ::= <forma>|<funkcja>

<ciąg argumentów> ::= <puste>|<niepusty ciąg argumentów>

<niepusty ciąg argumentów> ::= <argument>
<argument>;
<niepusty ciąg argumentów>

<ciąg par form> ::= <para form>|<para form>;

<para form> ::= <forma>→<forma>

<ciąg form> ::= <forma>|<forma>; <ciąg form>

<symbol atomowy1> oznacza zbiór symboli atomowych
z wyjątkiem LAMBDA i LABEL

Podamy teraz zasady przekładu M-wyrażeń na S-wyrażenia stosując następujący zapis. Jeśli <V> jest zmienną podanej wyżej gramatyki M-wyrażeń, to <V*> będzie zmienną oznaczającą odpowiedni zbiór wyrażeń przetłumaczonych

na język danych, np. <forma*> dotyczy zbioru form przetłumaczonych na język S-wyrażeń.

1. Symbole atomowe i liczby pozostawia się bez zmian.

2. Wyrażenie postaci

   LAMBDA [[<ciąg zmiennych>]; <forma>]

   tłumaczy się na

   (LAMBDA (<ciąg zmiennych*>) <forma*>)

   gdzie <ciąg zmiennych*> ::= <puste>|<niepusty ciąg zmiennych*>

   <niepusty ciąg zmiennych*> ::= <symbol atomowy>|<symbol atomowy> <niepusty ciąg zmiennych*>

3. LABEL [<symbol atomowy>; <funkcja>]

   tłumaczy się na

   (LABEL <symbol atomowy> <funkcja*>)

4. QUOTE [<S-wyrażenie>] tłumaczy się na

   (QUOTE <S-wyrażenie>)

5. <funkcja>[<ciąg argumentów>] tłumaczy się na

   (<funkcja*> <ciąg argumentów*>)

   gdzie <ciąg argumentów*> ::= <puste>|<niepusty ciąg argumentów*>

   <niepusty ciąg argumentów*> :: <argument*>|<argument*> <niepusty ciąg argumentów*>

   i <argument*> ::= <forma*>|(FUNCTION <funkcja*>)

Zatem, jeśli argumentem w wyrażeniu powstałym przez złożenie funkcji z argumentami jest znów funkcja, wtedy

dopisuje się wyróżniony symbol atomowy FUNCTION pełniący rolę operatora i dodatkową parę nawiasów.

6. [<ciąg par form>] tłumaczy się na

(COND <ciąg par form*>)

gdzie <ciąg par form*> ::= <para form*>|<para form*> <ciąg par form*>

<para form*> ::= (<forma><forma>)

7. PROG[[<ciąg zmiennych>];<ciąg form>]

tłumaczy się na

(PROG (<ciąg zmiennych*>)<ciąg form*>)

gdzie <ciąg form*> ::= <forma*>|<forma*><ciąg form*>

8. <symbol atomowy1> ::= <forma>

tłumaczy się na

(SETQ <symbol atomowy1><forma*>)

Zgodnie z tymi zasadami przekładu, funkcji LENGTH podanej w poprzednim punkcie będzie odpowiadać S-wyrażenie

```
(LABEL LENGTH
     (LAMBDA (X) (PROG (U V)
                       (SETQ V 0)
                       (SETQ U  X)
                     A (COND ((NULL U)(RETURN V)))
                       (SETQ U (CDR U))
                       (SETQ V (ADD1 V))
                       (GO A))))
```

Należy dodać, że funkcja READ akceptuje także dowolne S-wyrażenia (czyli również S-wyrażenia reprezentujące wyrażenia funkcyjne). Zatem można pisać funkcje wprost w języku S-wyrażeń.

1.5. Baza danych systemu i funkcje systemowe

Każdy symbol atomowy, bądź stale istniejący w systemie, bądź wprowadzony przez programistę, posiada swą reprezentację w pamięci i jest nieusuwalny.

Symbolowi atomowemu można przypisać dowolną liczbę cech i znaczników.

Cecha posiada nazwę i wartość. Pewne cechy mają w systemie zastrzeżony sens i w sposób istotny wpływają na działanie różnych części systemu. Poniżej zestawiono te systemowe cechy podając ich nazwę i sposób interpretowania wartości cechy.

APVAL wartość symbolu atomowego jako zmiennej globalnej

EXPR wyrażenie funkcyjne (w postaci S-wyrażenia) przypisane symbolowi atomowemu

SUBR adres programu przypisanego symbolowi atomowemu

PNAME kod znaków alfanumerycznych tworzących symbol atomowy

FEXPR, FSUBR } posiadają sens zbliżony do EXPR i SUBR. Wartości tych cech są w specjalny sposób wykorzystywane przez interpreter i kompilator.

Znacznik posiada tylko nazwę. Stały sens w systemie mają znaczniki TRACE, SPECIAL i COMMON. Przypisanie znacznika TRACE symbolowi atomowemu funkcji powoduje wydruk śladu obliczeń, czyli wydruk nazwy wykonywanej funkcji, jej aktualnych argumentów i wyniku.

Znaczniki SPECIAL i COMMON są wykorzystywane przez kompilator i ich znaczenie opiszemy w punkcie 5.

Oprócz funkcji elementarnych istnieje w systemie szereg funkcji pomocniczych. Wszystkie funkcje stale istniejące w systemie nazywają się systemowymi i wyróżnia się następujące ich rodzaje:

1. funkcje elementarne
2. funkcje wejścia-wyjścia
3. funkcje operujące na bazie danych
4. funkcje będące uzupełnieniem operatora COND
5. funkcje operujące na listach
6. funkcje interpretera i kompilator.

Funkcje elementarne omówiono już w punkcie 1.2. Obecnie dokonamy przeglądu pozostałych funkcji systemowych.

Funkcje wejścia - wyjścia

Do funkcji wejścia-wyjścia należą READ,RATOM,READCH wprowadzające informację, PRINT,PRIN1 i PRINCH wyprowadzające informację, funkcje OUTBUF i FORMAT ustalające format wyprowadzania oraz funkcję INPUT, OUTPUT i START ustalające aktywność urządzeń wejścia-wyjścia lub inicjujące ich pracę.

Funkcje wejścia-wyjścia opisano szczegółowo w punkcie 3 instrukcji. Tutaj podamy jedynie krótką charakterystykę funkcji wprowadzających i wyprowadzających informację.

READ funkcja zeroargumentowa powodująca wprowadzenie z aktualnego wejścia jednego S-wyrażenia lub

jednej funkcji zapisanej w metajęzyku. Wynikiem jest wczytane wyrażenie.

RATOM funkcja zeroargumentowa wczytująca jednostkę leksykalną (symbol atomowy, liczbę, dowolny tekst lub ogranicznik). Wynikiem jest wczytana jednostka.

PRINT funkcja jednoargumentowa dokonująca wyprowadzenia swego argumentu na aktualne wyjście. Wynikiem formalnym jest wyprowadzane wyrażenie.

PRIN1 Jednoargumentowa funkcja wyprowadzająca jednostkę leksykalną będącą jej argumentem. Wynikiem formalnym jest wyprowadzona jednostka.

PRINCH funkcja dwuargumentowa. Pierwszy argument winien być symbolem atomowym, drugi liczbą całkowitą lub ósemkową. Funkcja powoduje wydruk n-tego znaku symbolu atomowego, gdzie n jest wartością drugiego argumentu, lub nie wywołuje żadnych działań, gdy n jest większe od liczby znaków symbolu atomowego będącego pierwszym argumentem. W pierwszym przypadku wynikiem formalnym jest T, zaś w drugim NIL.

Na przykład wykonanie funkcji PRINCH na argumentach CAR i 3 spowoduje wyprowadzenie litery R i da wynik formalny T.

Funkcje systemowe operujące na bazie danych

W skład tej grupy wchodzą funkcje systemowe, które zmieniają stan bazy danych, lub odczytują informacje związane ze stanem aktualnym.

Funkcja DEFLIST jest dwuargumentowa i akceptuje jako pierwszy argument S-wyrażenie postaci

((<symbol atomowy><S-wyrażenie>)...

(<symbol atomowy><S-wyrażenie>))

czyli listę podlist dwuelementowych. Natomiast drugi swój argument DEFLIST traktuje jako nazwę cechy. W rezultacie wykonania DEFLIST wszystkim symbolom atomowym wypisanym w pierwszym argumencie jako pierwsze elementy podlist zostaje przypisana cecha o nazwie będącej drugim argumentem DEFLIST. Jako wartości tej cechy podstawiają się odpowiednie S-wyrażenia wypisane w podlistach na drugim miejscu.

Na przykład wykonanie funkcji DEFLIST na argumentach

((OLA BARDZO)(ALA SREDNIO)(ELA MALO))

URODNA

powoduje przypisanie symbolom atomowym OLA,ALA,ELA cech URODNA o odpowiednich wartościach: BARDZO,SREDNIO i MALO, które w tym przypadku są także symbolami atomowymi.

Wynikiem formalnym DEFLIST jest lista symboli atomowych, którym przypisano cechy.

Oprócz DEFLIST istnieją dodatkowo trzy funkcje: DEFINE, CSET i CSETQ, które ułatwiają realizację szczególnych przypadków, w jakich może być wykorzystywana DEFLIST.

DEFINE jest funkcją o jednym argumencie tej samej postaci, co pierwszy argument funkcji DEFLIST. DEFINE jest bowiem szczególnym przypadkiem DEFLIST i służy do definiowania cech EXPR symboli atomowych.

Funkcje CSET i CSETQ natomiast pełnią tę rolę w stosunku do cech APVAL. Są one dwuargumentowe - pierwszy argument

winien być symbolem atomowym, a drugi wartością przypisywaną cesze APVAL tego symbolu.

Różnicę między CSET i CSETQ wyjaśnia równość

CSETQ[X ; Y] = CSET[QUOTE[X]; Y]

Wynikiem formalnym obu funkcji jest wartość ich drugiego argumentu.

Ważną rolę pełni funkcja GET służąca do odczytywania wartości cech symboli atomowych lub do badania, czy symbolom atomowym przypisano znaczniki. GET jest dwuargumentowa i oba argumenty winny być symbolami atomowymi. Pierwszy z nich podlega testowaniu, a drugi jest nazwą cechy lub znacznika.

Gdy drugim argumentem GET jest nazwa cechy, to zasadniczo wynikiem jest albo wartość tej cechy, jeśli badany symbol atomowy ją posiada, albo NIL, gdy nie posiada tej cechy. Wyjątkiem są cechy SUBR i FSUBR bowiem dla nich wynikiem GET jest odpowiednio T lub NIL.

Gdy drugim argumentem GET jest nazwa znacznika, to wynikiem jest T, jeśli symbol atomowy był oznaczony tym znacznikiem, lub NIL w przeciwnym razie.

Do usuwania cech i ich wartości służy funkcja REMPROP. Pierwszy jej argument winien być symbolem atomowym, któremu odbiera się cechę, a drugi nazwą cechy. W wyniku działania funkcji REMPROP usuwa się cechę łącznie z jej wartością, jeśli symbol atomowy wogóle tę cechę posiadał, w przeciwnym razie nie wykonuje się żadnych działań. Wynikiem formalnym REMPROP jest wartość jej pierwszego argumentu.

Do manipulacji na dowolnych znacznikach służą funkcje FLAG i REMFLAG oraz, w charakterze ułatwienia, 6 funkcji obsługujących szczególne przypadki znaczników.

Funkcja FLAG jest dwuargumentowa. Pierwszy argument winien być listą symboli atomowych, którym przypisuje się znacznik, a drugi argument przypisywanym znacznikiem. Wynikiem formalnym jest NIL.

Identyczą postać argumentów posiada funkcja REMFLAG, która służy do usuwania znacznika. Jej wynikiem formalnym jest również NIL.

Funkcje TRACE, SPECIAL i COMMON służą do przypisywania symbolom atomowym znaczników o nazwach identycznych ze swymi nazwami, czyli TRACE, SPECIAL i COMMON. Akceptują one jeden argument o postaci tej samej co pierwszy argument funkcji FLAG (czyli listę symboli atomowych) i dają jako wynik formalny NIL.

Natomiast funkcje UNTRACE, UNSPECIAL i UNCOMMON usuwają te znaczniki, działając podobnie do REMFLAG. Postać argumentu i wyniku jest taka sama, jak w przypadku funkcji przypisujących znaczniki.

Interesującą funkcją rozszerzającą bazę danych jest zeroargumentowa funkcja GENSYM. Generuje ona nowy symbol atomowy o przypadkowym kodzie znaków alfanumerycznych, który nie istniał dotąd w systemie. Utworzony symbol atomowy jest wynikiem formalnym GENSYM.

Do funkcji operujących na bazie danych zaliczamy także funkcję EXCISE służącą do usuwania z systemu kompilatora (wykonanie EXCISE z argumentem o wartości COMPILE)

lub kompilatora i pełnego interpretera (wykonanie EXCISE z argumentem o wartości EVALQUOTE). Obszary pamięci zajęte normalnie przez te systemowe programy zostają wtedy włączone do obszaru, w którym zapisuje się dane.

Funkcje będące uzupełnieniem operatora COND

W systemie istnieje kilka funkcji, które są uzupełnieniem operatora COND. Pozwalają one wyrazić krócej efekt, jaki można uzyskać stosując jedynie operator COND.

Funkcje AND i OR mogą mieć dowolną liczbę argumentów. Spełniają one zależności

$$AND[X_1;X_2;\ldots;X_k] = [X_1 \rightarrow [X_2 \rightarrow \ldots [X_k \rightarrow T; T \rightarrow NIL]; \ldots T \rightarrow NIL]; T \rightarrow NIL]$$

oraz

$$OR[X_1;X_2;\ldots;X_k] = [X_1 \rightarrow T; X_2 \rightarrow T; \ldots; X_k \rightarrow T; T \rightarrow NIL]$$

Widać zatem, że jeśli argumenty $X_1,\ldots,X_k$ przyjmują wartości T albo NIL, to funkcje AND i OR realizują mnożenie lub sumowanie logiczne wartości swych argumentów odbywające się w dość szczególny sposób. AND bowiem szuka kolejno argumentu, który ma wartość NIL, zaś OR argumentu, który posiada wartość T. Przy tym obliczenie wartości argumentów odbywa się "wewnątrz" funkcji, a nie przed jej wywołaniem.

Funkcja jednoargumentowa NOT spełnia zależność

$$NOT[X] = [X \rightarrow NIL; T \rightarrow T]$$

Pewną modyfikacją operatora warunkowego COND jest również operator SELECT, który zostanie omówiony szczegółowo w punkcie 4.

Funkcje operujące na listach

System wyposażono w kilka funkcji dokonujących działań na listach

LENGTH służy do obliczania długości listy. Jej wynikiem jest liczba całkowita równa długości listy, jaka jest wartością argumentu.

LIST jest funkcją od dowolnej liczby argumentów. Z wartości tych argumentów zostaje utworzona lista, która jest wynikiem funkcji.

APPEND służy do łączenia dwóch list w jedną.

MEMBER jest funkcją od dwóch argumentów. Jest to predykat, który bada, czy wartość argumentu pierwszego jest elementem listy będącej wartością argumentu drugiego.

UNION jest funkcją od dwóch argumentów, których wartości są traktowane jak listowe zapisy dwóch zbiorów. UNION tworzy listę będącą sumą mnogościową tych dwóch zbiorów.

Ponadto do grupy tej można zaliczyć funkcje MAPLIST i SEARCH, o których informacje podano w zestawieniu alfabetycznym w punkcie 4.

Funkcje interpretera i kompilator

Jest to grupa funkcji systemowych, które omówiono w punktach 1.6 i 5.

1.6. Interpretacja wyrażeń funkcyjnych

S-wyrażenie będące zapisem funkcji w języku danych może być poddane interpretacji, co oznacza, że po podaniu wartości argumentów, na jakie funkcja ta ma działać, dokonuje się obliczenia jej wyniku.

Funkcja systemowa, która dokonuje interpretacji nazywa się EVALQUOTE i jest dwuargumentowa. Jej pierszym argumentem powinno być S-wyrażenie do zinterpretowania, zaś drugim lista wartości argumentów uczestniczących w interpretacji.

Użytkownik współpracuje z systemem zlecając mu realizację zadań. Jedno zadanie związane jest z jednym wykonaniem funkcji EVALQUOTE i jest realizowane w cyklu składającym się z następujących części

- dwukrotnego wykonania funkcji READ celem wprowadzenie obu argumentów EVALQUOTE,
- wykonanie funkcji EVALQUOTE,
- wykonanie funkcji PRINT celem wyprowadzenia wyniku interpretacji będącego także wynikiem zadania.

Jeśli pierwszy argument EVALQUOTE jest symbolem atomowym i posiada cechę SUBR, wówczas zadanie interpretera sprowadza się do wywołanie programu funkcji oznaczonej tym symbolem atomowym. Jeśli natomiast pierwszy argument EVALQUOTE jest symbolem atomowym nie posiadającym cechy SUBR, lecz cechę EXPR, lub jeśli argument ten jest złożonym S-wyrażeniem, wówczas ma miejsce pełna interpretacja złożonego S-wyrażenia.

Należy pamiętać, że elementy drugiego argumentu EVALQUOTE (czyli wartości argumentów interpretowanej funkcji) uczestniczą w obliczeniach "dosłownie" - bez żadnej uprzedniej interpretacji. Zatem następujące sformułowanie zadania

CONS (A B)

jest poprawne i da wynik (A.B).

Natomiast nie jest poprawny. jako sformułowanie zadania, zapis

CONS [A ; B]

Ponieważ funkcja READ akceptuje wyrażenia metajęzyka, można zadania zapisywać także w postaci

LAMBDA [[X,Y]; CONS [X,Y]] (A B)

EVALQUOTE współdziała z szeregiem funkcji pomocniczych, które będąc częściami interpretera są także dostępne dla użytkownika ~~współdziała~~. Są to funkcje APPLY, EVAL, EVCON, EVLIST, NCONC i PAIR.

Nieco więcej szczegółów na temat działania interpretera i zasad interpretacji podano w punkcie 5.

1.7. Kompilacja wyrażeń funkcyjnych

System wyposażono w kompilator, który jest funkcją systemową o nazwie COMPILE. Jest to funkcja jednoargumentowa - wartością argumentu winna być lista symboli atomowych.

Symbole atomowe znajdujące się na tej liście winny posiadać cechę EXPR lub FEXPR. Funkcja COMPILE dokonuje tłumaczenia wyrażeń funkcyjnych będących wartościami tych cech.

Rezultatem tłumaczenia są stałomiejscowe programy zapisane w pamięci w postaci ciągów makroinstrukcji.

Przetwarzane symbole atomowe uzyskują cechę SUBR lub FSUBR, nie tracąc cech EXPR lub FEXPR.

Wynikiem formalnym funkcji COMPILE jest NIL, lecz istotny jest jej efekt uboczny. Dzięki kompilacji bowiem funkcje zdefiniowane i przetłumaczone są wykonywane wielokrotnie szybciej w porównaniu z wykonaniem ich przez interpretację.

Sygnalizację błędów i szczegóły działania funkcji COMPILE omówiono w punkcie 5.

Przykład zdefiniowania, skompilowania i wykonania funkcji

```
DEFINE (((AZOR LAMBDA [[X];[NULL[X] → NIL;
                    T → CONS [ CONS[CAR[X]; NIL]; AZOR[CDR[X]]]]]
        )))
(AZOR)
COMPILE ((AZOR))
NIL
AZOR ((A B C D))
((A)(B)(C)(D))
```

2. Współpraca z monitorem systemu

2.1. Uwagi wstępne

Monitor jest programem zarządzającym pracą wszystkich programów systemu. Składa się z kilkunastu części, z których najważniejsze zarządzają przydziałem pamięci oraz organizują i obsługują struktury danych sterowania. Inne jego części zapewniają realizację ciągu makroinstrukcji oraz realizują programowy mechanizm przerwań.

Monitor przygotowuje system do wykonania pełnego cyklu zadania: wczytania funkcji z listą argumentów, wykonania funkcji i wyprowadzenia wyniku, oraz zarządza systemem podczas realizacji tego cyklu.

Istotną cechą monitora jest możliwość przerwania zadań celem wprowadzenia i wykonania innego zadania z zapewnieniem późniejszego powrotu do zadania przerwanego. Wielopoziomowy mechanizm przerwań daje użytkownikowi możliwość dynamicznej współpracy z systemem. Można np. na wyższym poziomie obliczeń (czyli po przerwaniu i zawieszeniu aktualnego programu) zmienić bazę danych wykorzystywaną przez zadanie na poziomie niższym.

2.2. Uruchomienie systemu LISP

Z punktu widzenia systemu operacyjnego SOK-1 system LISP-u traktowany jest jak normalny program użytkowy. Wszystkie uwagi zawarte w instrukcji "Język operacyjny maszyny JOM-1" , dotyczące programów użytkowych systemu

SOK-1, dotyczą także systemu LISP. W szczególności obowiązują zlecenia A*, G*<adres>;, RS*. Po wprowadzeniu taśmy binarnej systemu LISP, uruchamia się go zleceniem A* lub zleceniem G*0400;. Sterowanie przejmuje wówczas monitor, który sygnalizuje ten fakt wydrukiem na dalekopisie

LISP IS READY

W dalszym ciągu monitor uruchamia specjalny program systemowy STOPER będący zadaniem poziomu zerowego. Program ten zawiera pętlę i wykonuje się dowolnie długo, aż do chwili przerwania, w której system przechodzi w stan gotowości na poziomie pierwszym.

2.3. Dokonywanie przerwań

Aby wykonać zadanie, należy wprowadzić system LISP w stan gotowości.

Podczas pracy systemu występują okresy aktywności i nieaktywności mechanizmu przerwań. Uzyskanie stanu gotowości jest oczywiście możliwe tylko w okresie aktywności mechanizmu przerwań, czyli jeśli spełnione są dwa warunki:

- nie wykonuje się programów blokujących przerwania (COMPILE, READ, PRINT, zbieracza śmieci i funkcji systemowych zaprogramowanych w języku wewnętrznym)
- system przystępuje do wykonania kolejnej makroinstrukcji.

System wprowadza się w stan gotowości przez manipulowanie klawiszem KLO znajdującym się na pulpicie jednostki centralnej. Stany i zmiany stanów klawisza KLO będziemy

oznaczać następująco:

stan 0 klawisz w położeniu górnym

zmiana 01 klawisz przechodzi w położenie dolne

stan 1 klawisz pozostaje w położeniu dolnym

zmiana 10 klawisz przechodzi w położenie górne

System przechodzi w stan gotowości w następujących przypadkach:

1. Nastąpiła zmiana 01 stanu klawisza KLO
   a/ jeśli mechanizm przerwań jest aktywny, to nastąpi natychmiastowe przejście na wyższy poziom i sygnalizacja stanu gotowości wydrukiem na dalekopisie

   LISP < n >:

   gdzie < n > jest liczbą naturalną oznaczającą poziom przerwania; w tym stanie system jest gotów wczytać i wykonać zadanie,
   b/ jeśli mechanizm przerwań jest nieaktywny, a klawisz KLO pozostaje w stanie 1, wówczas przerwanie zostanie zrealizowane w najbliższej chwili aktywności.
2. Klawisz KLO trwał w stanie 1, a zostało zakończone zadanie na danym poziomie; wtedy system pozostaje na tym samym poziomie odnawiając stan gotowości i sygnalizując to wydrukiem

   LISP < n >:

Jeśli natomiast przy wykonywaniu zadania na poziomie < n >, klawisz KLO został sprowadzony do stanu 0, wówczas po zakończeniu zadania nastąpi zejście na poziom niższy sygnalizowane wydrukiem

*< n >

na dalekopisie. Wówczas system przejdzie automatycznie do kontynuacji zadania zawieszonego na poziomie niższym (o numerze <n>-1). W szczególnym przypadku może to być zadanie poziomu zerowego, czyli program systemowy STOPER. Użytkowink może zatem:

a/ wykonać jedno zadanie, jeśli przed jego zakończeniem sprowadzi klawisz KLO do stanu 0 (można to zrobić bezpośrednio po wprowadzeniu systemu w stań gotowości, czyli po wydruku LISP <n> :)

b/ wykonywać kolejne zadania na pewnym poziomie, jeśli klawisz KLO pozostaje trwale w stanie 1.

Zadanie wykonywane na pewnym poziomie może wpłynąć, poprzez zmiany w bazie danych, na tok dalszego wykonywania zadań zawieszonych na poziomach niższych. Użytkownik wykorzystujący ten sposób współpracy z systemem musi zdawać sobie sprawę z wynikających stąd skutków.

2.4. Wprowadzenie zadania do systemu

W systemie LISP wprowadzono pojęcie wejścia i wyjścia aktualnego. Jest to para urządzeń peryferyjnych wybrana aktualnie do współpracy z zadaniem. Programista może określać aktualność urządzeń przez wykonanie funkcji systemowych INPUT i OUTPUT. Poszczególnym urządzeniom przypisano numery. Numeruje się niezależnie urządzenia wejściowe i wyjściowe. Numeracja ta jest zgodna z przyjętą w ASSK-u, zatem w szczególności czytnik taśmy papierowej i jej perforator posiadają numer 1, zaś dalekopis numer 3.

Numery 10 i 11 wprowadzono dodatkowo aby umożliwić współpracę czytnika z buforem wejściowym. Szczegóły tej współpracy będą omówione w punkcie 3.9.
Argumentem funkcji INPUT (oraz OUTPUT) jest liczba całkowita lub ósemkowa określająca numer urządzenia aktualnego.

Bezpośrednio po wejściu systemu w stan gotowości uaktywnia się dalekopis. Stanowi on wtedy zarówno urządzenie wejściowe jak i wyjściowe. Uaktywnienie to ma charakter blokady dotychczas aktualnej pary urządzeń i trwa do momentu zakończenia zadania przerywającego. Wykonanie, jako zadań przerywających, funkcji INPUT i OUTPUT obejmuje wtedy swym skutkiem zadania zawieszone na poziomach niższych. Wyjątek w tej regule stanowi specjalna zeroargumentowa, bezargumentnikowa pseudofunkcja START. Po jej wykonaniu kontynuowany jest stan gotowości (bez wydruku tekstu LISP < n > :) i następuje wczytanie zadania z urządzenia określonego ostatnio wykonaną funkcją INPUT. Wyjściem aktualnym podczas realizacji tego zadania jest urządzenie określone ostatnio wykonaną funkcją OUTPUT.

2.5. Przykład uruchomienia systemu i wykonania zadania

Załóżmy, iż działa aktualnie system operacyjny SOK-1, a na dalekopisie pojawił się wydruk

```
K202
:
```

Użytkownik zakłada taśmę binarną systemu LISP do czytnika i pisze zlecenie

```
RB *
```

Następuje czytanie taśmy zakończone wydrukiem

```
:
```

Użytkownik uruchamia system zleceniem

A*

a na tabulogramie pojawia się sygnalizacja

LISP IS READY

Użytkownik naciska klawisz KL0, wprowadzając go w stan 1

LISP 1 :

Chcąc ustalić jako aktualne urządzenia we-wy czytnik i perforator, użytkownik pisze

INPUT (1)

Drukuje się wynik formalny funkcji INPUT (1)

1

ponieważ klawisz KL0 pozostał w stanie 1, system sygnalizuje stan gotowości

LISP 1 :

Użytkownik pisze

OUTPUT (1)

pojawia się wynik formalny

1

oraz podobnie, jak poprzednio

LISP 1 :

Użytkownik podnosi klawisz KL0 do stanu 0, zakłada taśmę z programem do czytnika i pisze na dalekopisie

START ( )

Następuje czytanie zadania i po zakończeniu czytania przejście do jego wykonania.

W trakcie wykonywania zadania użytkownik naciska klawisz KL0, wprowadzając go w stan 1.

Następuje przerwanie sygnalizowane napisem

LISP 2 :

Użytkownik sprowadza klawisz do stanu 0 i pisze zadanie, np.

CSET (A 15)

Drukuje się wynik

15

a ponieważ klawisz KLO jest w stanie 0, sygnalizowany jest powrót na niższy poziom

*2

Zadanie na poziomie pierwszym zostaje zakończone, wynik wyprowadza się na perforator i następuje powrót na poziom zerowy

*1

System czeka na przerwanie zadania z poziomu zerowego

Powrót do systemu operacyjnego SOK-1, czyli przerwanie działania systemu LISP dokumuje się w normalny sposób, przez naciśnięcie klawisza OPRQ. System SOK-1 daje możliwość restartu.

2.6. Rozszerzenie obszaru pamięci zajmowanego przez dane

Istnieje możliwość udostępnienia zadaniom użytkownika dodatkowych obszarów pamięci zajętych normalnie przez funkcje kompilatora i interpretera. Zadanie to realizuje jednoargumentowa funkcja EXCISE. Wykonanie jej z argumentem o wartości COMPILE powoduje nieodwracalne usunięcie z systemu

kompilatora, zaś wykonanie EXCISE z argumentem o wartości EVALQUOTE usunięcie równoczesne kompilatora i pełnego interpretera. W systemie pozostaje wówczas jedynie mini-interpreter, który interpretuje nazwy funkcji posiadające określoną cechę SUBR.

W obu przypadkach zwolnione obszary pamięci zwiększają obszar wykorzystywany przez dane.

3. Wprowadzanie i wyprowadzanie danych

W punkcie tym omówimy funkcje systemowe, zwane funkcjami wejścia-wyjścia, do których należą READ, RATOM i READCH (wprowadzanie informacji), PRINT, PRIN1 i PRINCH (wyprowadzanie informacji), OUTBUF i FORMAT (ustalanie formatu wyprowadzania) oraz INPUT, OUTPUT i START (ustalanie aktywności urządzeń wejścia-wyjścia lub inicjowanie ich pracy).

3.1. RATOM

RATOM jest funkcją zeroargumentową. Wykonanie jej powoduje wprowadzenie z aktualnego wejścia jednej z następujących jednostek leksykalnych:

a/ symbolu atomowego

b/ liczby,

c/ dowolnego tekstu,

d/ ogranicznika 1,

określonych gramatyką zamieszczoną w tabeli 3.1.

Sposób zapisu symboli atomowych i liczb przedstawiono już w punkcie 1. Tutaj omówimy jeszcze taki sposób wprowadzania dowolnego tekstu, aby był on traktowany jako symbol atomowy.

Dowolne teksty

Znak występujący po parze znaków $ $ traktuje się jako znacznik otwierający dowolny tekst (uwaga- w niektórych dalekopisach znak $ jest zastąpiony przez £). Ze wszystkich dalszych znaków występujących przed powtórzeniem się znacznika tworzy się jeden symbol atomowy.

Tabela 3.1.

Gramatyka jezyka zewnetrznego (poziom leksykalny)

1. < znak alfan. > ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|
S|T|U|V|W|X|Y|Z|$|/|:|=|.|,|↑|-|+|(|
)|[|]|cr|lf|sp|;|!|"|#|%|&|'|<|>|\|
|@|←|*|0|1|2|3|4|5|6|7|8|9

2. < litera > ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|
V|W|X|Y|Z|!|"|#|%|'|&|/|↑| |<|\|@|←|:

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

4. < cyfra1 > ::= 1|2|3|4|5|6|7|8|9

5. < cyfra osemkowa > ::= 0|1|2|3|4|5|6|7

6. <znak > ::= +|-

7. < ogranicznik > ::= (|)|;|.|[|]|,|cr|lf|sp|->|*|:=

8. < ogranicznik1 > ::= (|)|[|]|;|*|.|->|:=

9. < dowolny tekst > ::= $$ < znak alfan. > $\{$ < znak alfan. > $\}_{0}^{63}$

< znak alfan. >

10. < symbol atomowy > ::= < litera >
$\{$ < litera >|< cyfra >|< znak > $\}_{0}^{63}$

11. < liczba > ::= < liczba osemkowa >|< liczba dziesietna >

12. < liczba osemkowa > ::= 0 $\{0|1\}_{0}^{1}$ $\{$ < cyfra osemkowa > $\}_{0}^{5}$

13. < liczba dziesietna > ::= < liczba calkowita >|
< liczba zmiennop. >

14. < liczba calkowita > ::= < znak >< liczba1 >*< liczba2 >

15. < liczba1 > ::= $\{$ < cyfra > $\}_{1}^{5}$

16. < liczba2 > ::= < cyfra1 > $\{$ < cyfra > $\}_{0}^{4}$

17. < liczba zmiennop. > ::= $\{$< znak >$\}_{0}^{1}$ < liczba1 > .

< liczba1 > < ulamek > | < liczba pol >

18. < ulamek > ::= < znak > $\{0\}_{0}^{1}$. $\{$< liczba1 >$\}_{1}^{2}$ |

0 $\{$< liczba1 >$\}_{1}^{2}$

19. < liczba pol > ::= < liczba dziesietna > E < cecha >

20. < cecha > ::= $\{$< znak >$\}_{0}^{1}$ < liczba1 >

Na przykład wykorzystując ! jako znacznik można wprowadzić symbol atomowy JAN KOWALSKI pisząc

$$!JAN KOWALSKI!

lub symbol atomowy +++ pisząc

$$-+++-

Poprawa błędów

Wystąpienie znaku $ w trakcie czytania symbolu atomowego lub liczby (dla liczb identyczny efekt dają wszystkie litery wyjąwszy E) powoduje wykrycie błędu, przerwanie wprowadzania znaków alfanumerycznych z aktualnego wejścia i przejście do stanu "oczekiwanie na poprawę błędu". Wprowadzony dotąd sznur znaków traktowany jest jako błędny symbol atomowy, lub błędna liczba. Na przykład, jeśli na aktualnym wejściu pojawił się sznur znaków

AR $

to na dalekopis zostaje wyprowadzony napis

CORRECT:AR$

i funkcja RATOM pozostaje w stanie "oczekiwanie na poprawę błędu" aż do momentu napisania na dalekopisie znaku : lub * .

Podczas poprawy błędów można odczytywać dalsze znaki z aktualnego wejścia tak, aby nie były one poddawane analizie. Każdorazowe napisanie na dalekopisie jednego ze znaków cr, lf, sp powoduje odczytanie z aktualnego wejścia kolejnego znaku alfanumerycznego i wydrukowanie go na dalekopisie. Znak nie jest analizowany. Czynność może być powtarzana wielokrotnie.

W stanie "oczekiwanie na poprawę błędu" wszystkie znaki alfanumeryczne poza cr, lf, sp, :, * są ignorowane. Napisanie na dalekopisie znaku ignorowanego powoduje wydruk po nim znaku ? i powtórne przejście funkcji RATOM w stan "oczekiwanie na poprawę błędu".

Napisanie na dalekppisie znaku : przeprowadza funkcję RATOM ze stanu "oczekiwanie na poprawę błędu" do stanu "poprawa błędu", natomiast znaku * przeprowadza ze stanu "oczekiwanie na poprawę błędu" do stanu "błąd".

W stanie "poprawa błędu" błędny symbol atomowy lub błędna liczba może zostać poprawiona przez napisanie na dalekopisie prawidłowej jednostki leksykalnej zakończonej znakiem cr, lf, sp. Można także kończyć tę jednostkę innym ogranicznikiem, jednak należy pamiętać, że jest on pamiętany w buforze, co może być przyczyną błędnego działania funkcji wejścia wykonanych później.

Jeśli w trakcie poprawy błędu zostanie wykryty błąd, to funkcja RATOM ponownie przechodzi w stan "oczekiwanie na poprawę błędu", drukując, jak poprzednio
CORRECT:<błędna jednostka>

Przejście do stanu "błąd", czyli napisanie na dalekopisie znaku * oznacza rezygnację z dalszej poprawy błędu. Wtedy następuje wydruk na dalekopisie informacji o błędzie w postaci napisu

ERROR 2

i przerwanie wykonywania funkcji RATOM.

Przykłady wykorzystywania mechanizmu poprawy błędów

1. Załóżmy, że z aktualnego wejścia przeczytano sznur

   ARE $

   wtedy na dalekopisie pojawi się wydruk

   CORRECT:ARE $

   Chcąc poprawić symbol atomowy programista pisze dalej na dalekopisie

   :ARSEN␣

   i funkcja RATOM wczytuje symbol atomowy ARSEN.

2. Jeżeli w sytuacji omówionej wyżej programista chce opuścić błędną jednostkę leksykalną pisze na dalekopisie

   : ␣

3. Jeżeli w tej samej sytuacji programista chce odczytać kolejny znak z aktualnego wejścia pisze

   ␣

   i kolejny znak zostaje wczytany i wydrukowany na dalekopisie. Jeżeli programista chce go wprowadzić do systemu pisze dalej : i powtarza znak odczytany. W rezultacie na dalekopisie powstaje wydruk

   ␣.:.

   i znak . zostaje wczytany przez funkcję RATOM.

Znak * jako znacznik końca taśmy

Jeśli ogranicznik przeczytany przez funkcję RATOM jest znakiem * , to następuje przejście w stan "zawieszenie". Na dalekopisie drukowany jest napis

TAPE

i oczekuje się na napisanie jednego znaku na dalekopisie.

Jeżeli tym znakiem będzie * , to następuje przerwanie wykonywania funkcji RATOM i druk napisu

ERROR 1

Napisanie dowolnego innego znaku powoduje powtórne wywołanie funkcji RATOM.

Dzięki temu można wykorzystać znak * jako znacznik końca taśmy przy pisaniu informacji na taśmie papierowej. Znak * chroni przed wypadnięciem taśmy z czytnika w przypadku, gdy na taśmie umieszczono błędnie zbyt małą liczbę jednostek leksykalnych, lub gdy informacje przygotowano na kilku taśmach i każda z nich kończy się znakiem * . Przeczytanie tego znaku zawiesza wykonanie funkcji RATOM. Można wymienić taśmę w czytniku i napisać na dalekopisie dowolny znak (poza znakiem *), co powoduje powtórne wywołanie funkcji RATOM. Jeśli napisano * to następuje przerwanie czytania i sygnalizacja błędu, a wydruk ma postać

TAPE *

ERROR 1

3.2. READ

READ jest funkcją zeroargumentową. Wykonanie jej powoduje wczytanie z aktualnego wejścia jednego wyrażenia zgodnie ze składnią:

<wyrażenie akceptowane przez READ> ::= <S-wyrażenie 1> |
<funkcja>

gdzie <S-wyrażenie 1> oznacza <S-wyrażenie> z wyjątkiem symboli atomowych LABEL i LAMBDA, natomiast produkcje gramatyczne dla zmiennych <S-wyrażenie> i <funkcja> podano

w punktach 1.1 i 1.4. Zmienna <funkcja> dotyczy zbioru wyrażeń metajęzyka.

Poszczególne jednostki leksykalne wchodzące w skład wczytywanego wyrażenia wprowadza się za pomocą funkcji RATOM. Zatem w trakcie wykonywania funkcji READ obowiązują zasady współpracy z funkcją RATOM opisane w poprzednim punkcie. Ponadto READ wykrywa błędy składniowe przeważnie dotyczące błędnej konstrukcji wyrażeń metajęzyka. Błędy te o numerach od 28 do 49, zamieszczono i opisano w wykazie błędów na końcu instrukcji. Wykrycie błędu powoduje wydruk postaci

ERROR <nr błędu>
<ostatnio wczytana jednostka leksykalna>

oraz przerwanie wykonywania funkcji READ.

3.3. READCH

READCH jest funkcją zeroargumentową służącą do wprowadzania z aktualnego wejścia jednego znaku alfanumerycznego. Wynikiem jej jest jednoznakowy symbol atomowy reprezentujący przeczytany znak.

3.4. PRINT

PRINT jest funkcją jednoargumentową. Wartość arguemntu może być dowolnym S-wyrażeniem, które zostaje wyprowadzone na aktualne urządzenie wyjściowe w postaci sznura znaków alfanumerycznych poprzedzonego znakami cr, lf. Wynikiem formalnym jest wyprowadzone wyrażenie. Funkcja PRINT korzysta z funkcji PRIN1 służącej do wyprowadzania symbolu atomowego lub liczby.

3.5. PRIN1

PRIN1 jest funkcją jednoargumentową. Wartością argumentu musi być symbol atomowy lub liczba, w przeciwnym razie wykonywanie funkcji zostaje przerwane i na dalekopisie pojawia się wydruk

ERROR 7

<wartość argumentu>

PRIN1 wyprowadza na aktualne wyjście wartość cechy PNAME symbolu atomowego lub wartość liczby. Wydruk nie jest poprzedzony znakami cr, lf. Liczbę drukuje się zgodnie z obowiązującym formatem (patrz opis funkcji FORMAT).
Wynikiem formalnym funkcji PRIN1 jest wyprowadzony symbol atomowy lub liczba.

3.6. PRINCH

PRINCH jest funkcją dwuargumentową. Wartością pierwszego argumentu winien być symbol atomowy, zaś drugiego liczba całkowita lub ósemkowa. Inne wartości argumentów prowadzą do sygnalizacji błędów

ERROR 3 - jeśli jako drugi argument funkcji nie pojawiła się liczba całkowita lub ósemkowa

ERROR 6 - jeśli jako pierwszy argument funkcji nie pojawił się symbol atomowy

Ponadto wyprowadza się wartość argumentu, która była przyczyną błędu i przerywa wykonywanie funkcji PRINCH.

Niech drugi argument funkcji posiada wartość n, a symbol atomowy podstawiony jako pierwszy argument niech posiada wartość cechy PNAME składającą się z r znaków.

Wykonanie funkcji PRINCH powoduje

a/ dla $n \leq r$ i $n > 0$ wydrukowanie na aktualnym wyjściu n-tego znaku alfanumerycznego symbolu atomowego. Wartością formalną funkcji PRINCH jest w tym przypadku symbol atomowy T

b/ dla $n > r$ lub $n \leq 0$ brak jakichkolwiek działań na aktualnym wyjściu. Wartością formalną funkcji PRINCH jest w tym przypadku symbol atomowy NIL.

3.7. OUTBUF

Funkcja jednoargumentowa. Wartością argumentu winna być liczba całkowita lub ósemkowa. Funkcja OUTBUF ustala wskaźnik liczby znaków alfanumerycznych wyprowadzanych na aktualnym wyjściu w jednej linii. Aż do jej powtórnego wykonania wskaźnik ten jest równy wartości argumentu. Wartość ta jest także wynikiem formalnym funkcji.

W przypadku, gdy wartość argumentu nie jest liczbą całkowitą ani ósemkową, następuje przerwanie wykonywania funkcji i wydruk na dalekopisie

ERROR 8

<wartość argumentu>

natomiast znak argumentu jest ignorowany.

3.8. FORMAT

FORMAT jest funkcją jednoargumentową. Jako argument akceptowane są wyrażenia postaci

(<n>)

(<n><n>)

gdzie <n> jest liczbą całkowitą lub ósemkową. Jeśli pierwszy element tej listy nie jest liczbą całkowitą lub ósemkową, na dalekopisie następuje wydruk

ERROR 4

<pierwszy element>

i przerwanie wykonywania funkcji. Jeśli natomiast na liście występuje drugi element i nie jest on jedną z tych liczb, to pojawia się wydruk

ERROR 5

<drugi element>

i przerywa się wykonywanie funkcji FORMAT.

FORMAT służy do ustalania parametrów określających postać wyprowadzania liczb.

Wynikiem formalnym jest NIL.

Ustalanie formatu wyprowadzania liczb całkowitych

Jeśli wartością argumentu jest wyrażenie postaci

(<n>)

to ustala się format wyprowadzania liczb całkowitych i to w sposób następujący:

a/ <n> > 0 wyprowadzanie liczby całkowitej ze znakiem i n miejscami znaczącymi (początkowe zera są zastępowane odstępami)

b/ <n> < 0 wyprowadzanie liczby całkowitej ze znakiem minus dla liczby ujemnej i zamianą znaku na odstęp dla liczb dodatnich. Wyprowądza się <n>miejsc znaczących (początkowe zera są zastępowane odstępami).

Jeśli liczba całkowita posiada więcej niż <n> miejsc znaczących, to ilość wyprowadzanych miejsc zostaje odpowiednio zwiększona.

Po zainicjowaniu pracy systemu parametr wyprowadzania <n> = 1.

Format wyprowadzania liczb ósemkowych

Liczbę ósemkową wyprowadza się zawsze w postaci

Oxxxxxx

to jest, poprzedza się ją odstępem, po czym następuje cyfra zero i sześć cyfr ósemkowych. Zmiana postaci wyprowadzania nie jest możliwa.

Ustalenie formatu wyprowadzania liczb zmiennoprzecinkowych

Jeśli wartością argumentu funkcji FORMAT jest wyrażenie postaci

(<m><p>)

gdzie <m> i <p> są liczbami całkowitymi lub ósemkowymi, to ustala się parametry wyprowadzania liczb zmiennoprzecinkowych:

a/ jeśli <m> jest liczbą ósemkową, to liczbę zmiennoprzecinkową wyprowadza się w postaci półlogarytmicznej,

b/ jeśli <m> jest liczbą całkowitą, to wyprowadza się ją w postaci normalnej.

W obu przypadkach <m> oznacza ilość cyfr drukowaną przed kropką, a <p> ilość cyfr po kropce, przy czym początkowe zera zastępuje się odstępami.

Dla <p> > 0 liczbę wyprowadza się ze znakiem, zaś
dla <p> < 0 liczba wyprowadzana jest ze znakiem minus dla liczby ujemnej i bez znaku dla liczby dodatniej (zamiast znaku wyprowadza się odstęp)

Obliczając ilość pól dla wyprowadzanej liczby zmiennoprzecinkowej należy uwzględnić:

- pole na znak
- <m> pól na część całkowitą,
- <p> pól na część ułamkową,
- pole na kropkę,
- 4 pola na wykładnik (dla postaci półlogarytmicznej).

Jeśli wyprowadzana liczba zmiennoprzecinkowa posiada więcej niż <m> miejsc znaczących dla części całkowitej, tzn. jest nie mniejsza niż 10^m, a parametr wyprowadzania liczby wskazuje na wyprowadzanie w postaci normalnej, to wyprowadza się ją w postaci półlogarytmicznej z

<m> = 6 i <p> = 0

Po zainicjowaniu pracy systemu parametry wyprowadzania są równe <m> = 6 i <p> = 0 i postać półlogarytmiczna.

Przykłady wartości argumentu funkcji FORMAT i ich interpretacja

(-3) Liczby całkowite wyprowadzaj ze znakiem minus i trzema miejscami znaczącymi (znak + zastąp odstępem).

(3) Liczby całkowite wyprowadzaj ze znakiem i trzema miejscami znaczącymi.

(02 -3) Liczby zmiennoprzecinkowe wyprowadzaj w postaci półlogarytmicznej z dwoma miejscami przed kropką i trzema po kropce. Znak + zastąp odstępem.

(2 -3) Liczby zmiennoprzecinkowe wyprowadzaj w postaci normalnej z dwoma miejscami przed kropką i trzema po kropce. Znak + zastąp odstępem.

3.9. INPUT i OUTPUT

Są to funkcje jednoargumentowe. Wartością argumentu winna być liczba całkowita lub ósemkowa. W przeciwnym razie wystąpi sygnalizacja

ERROR <k>

<wartość argumentu>

gdzie <k> = 55 dla funkcji INPUT

i <k> = 56 dla funkcji OUTPUT

Funkcje te służą do ustalania wejścia i wyjścia aktualnego którego numer jest określony wartością argumentu. Numeracja urządzeń wejściowych i wyjściowych jest w zasadzie zgodna z przyjętą w ASSK-u z wyjątkiem numerów 10 i 11 dla urządzeń wejściowych. Numery te służą do ustalenia współpracy czytnika, jako aktualnego wejścia, ze stu znakowym buforem wejściowym. Bufor ten dodano celem poprawy warunków pracy czytnika przy wprowadzaniu obszernych zadań. Wykonanie funkcji INPUT dla argumentu o wartości 10 powoduje ponadto wyzerowanie tego bufora.

3.10. START

Pseudofunkcja zeroargumentowa bezwynikowa. Po jej wykonaniu kontynuowany jest stan gotowości systemu i następuje wczytanie zadania z urządzenia określonego ostatnio wykonaną funkcją INPUT. Podczas wykonywania tego zadania aktualnym wyjściem jest urządzenie określone ostatnio wykonaną funkcją OUTPUT.

4. Funkcje systemowe

W rozdziale zestawiono w porządku alfabetycznym wszystkie funkcje systemowe.

Dla każdej z nich podano liczbę i akceptowane wartości argumentów oraz opisano działanie funkcji.

ABS posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentów:

<wartość argumentu > ::= <liczba >

Działanie - funkcja oblicza wartość bezwzględną liczby

$abs[x] = |x|$

Wynikiem formalnym funkcji jest obliczona wartość bezwzględna.

Wynik jest liczbą tego samego typu, co argument.

Sygnalizacja

ERROR 65

<wartość argumentu >

oznacza, że użyty argument nie jest akceptowany.

ADD1 posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentów:

<wartość argumentu> ::= <liczba>

Działanie - funkcja zwiększa wartość argumentu o 1. Wynikiem formalnym jest liczba tego samego typu, co wartość argumentu.

Sygnalizacja

ERROR 66

<wartość argumentu>

oznacza zastosowanie nieakceptowanej wartości argumentu.

AND posiada cechę FSUBR

Liczba argumentów: dowolna.

Akceptowane wartości argumentów:

<wartość argumentu> ::= <S-wyrażenie reprezentujące formę>

Działanie.

Funkcja jest predykatem będącym uzupełnieniem operatora COND i pozwala zapisywać skrótowo wyrażenia warunkowe podanej niżej postaci. Spełniona jest zależność

$$AND[X_1;X_2;\ \ldots;\ X_k] =$$

$$[X_1 \rightarrow [X_2 \rightarrow \ldots [X_k \rightarrow T;\ T \rightarrow NIL];\ \ldots T \rightarrow NIL];T \rightarrow NIL]$$

Argumenty funkcji AND są interpretowane po jej wywołaniu za pomocą funkcji EVAL.

APPEND posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - funkcja łączy wartości argumentów tworząc z nich nowe S-wyrażenie. Wyrażenie symboliczne, będące wartością argumentu 2, zostanie dołączone jako zakończenie kropkowe do listy, będącej wartością pierwszego argumentu. Funkcję tę definiuje wyrażenie

LABEL[APPEND; LAMBDA[[X;Y];

[NULL[X]→Y; T→ CONS[CAR[X]; APPEND[CDR[X]; Y]]]]]

Wynikiem formalnym jest utworzone S-wyrażenie.

APPLY posiada cechę SUBR

Liczba argumentów: 3.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <S-wyrażenie>

<wartość argumentu 2> ::= <lista>

<wartość argumentu 3> ::= <lista par kropkowych>

Działanie- funkcja stanowi część interpretera. Wartość pierwszego argumentu zostaje zinterpretowana jako wyrażenie funkcyjne. Wartość drugiego argumentu jest traktowana jako lista wartości argumentów interpretowanego wyrażenia. Wartość trzeciego argumentu traktuje się jako listę asocjacji. Szczegóły działanie i sygnalizacje błędów omówiono w punkcie 5.

ARCT posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

⟨wartość argumentu⟩ ::= ⟨liczba⟩

Działanie - funkcja oblicza wartość funkcji trygonometrycznej arcus tangens.

Wynikiem formalnym funkcji jest obliczona wartość, będąca ~~również~~ liczbą zmiennoprzecinkową.

Sygnalizacja

ERROR 70

⟨wartość argumentu⟩

oznacza błąd polegający na użyciu nieakceptowanej wartości argumentu.

ATOM posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

⟨wartość argumentu⟩::=⟨S-wyrażenie⟩

Działanie - predykat ten bada, czy wartością argumentu jest symbol atomowy, czy też nie.

Wynikiem formalnym jest T, jeżeli wartością argumentu jest symbol atomowy, natomiast NIL w przeciwnym wypadku.

CAR posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentów:

<wartość argumentu> ::= <para kropkowa>

Działanie - funkcja wyciąga pierwszy element pary kropkowej.

Wynikiem formalnym funkcji jest pierwszy element pary kropkowej.

Sygnalizacja błędu

ERROR 50

<wartość argumentu>

zostanie wyprowadzona przy błędnej wartości argumentu.

CDR posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentów:

<wartość argumentu> ::= <para kropkowa>

Działanie - funkcja wyciąga drugi element pary kropkowej

Wynikiem formalnym jest drugi element pary kropkowej.

Sygnalizacja błędu

ERROR 51

<wartość argumentu>

zostanie wyprowadzona przy błędnej wartości argumentu.

COMMON posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie - funkcja przypisuje symbolom atomowym znajdującym się na liście znacznik COMMON. Podczas swego działania wykorzystuje FLAG jako funkcję pomocniczą. Zatem sygnalizacja błędów jest taka sama, jak dla funkcji FLAG.

Wynikiem formalnym jest NIL.

COMPILE posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie. Przetwarzane są kolejno symbole atomowe znajdujące się na liście, która jest wartością argumentu. Funkcja dokonuje tłumaczenia wyrażeń funkcyjnych przypisanych tym symbolom atomowym jako wartości cechy EXPR lub FEXPR. Rezultatem tłumaczenia są stałomiejscowe programy zapisane w pamięci w postaci ciągów makroinstrukcji. Przetwarzane symbole atomowe uzyskują cechę SUBR lub FSUBR.

Wynikiem formalnym jest NIL.

Sygnalizację błędów i szczegóły działania funkcji omówiono w punkcie 5.

COND posiada cechę FSUBR

Liczba argumentów: dowolna

Akceptowane argumenty:

<argument> ::= (<S-wyrażenie><S-wyrażenie>)

Działanie - COND służy do zapisu wyrażenią warunkowego w języku S-wyrażeń. Jako funkcja jest częścią interpretera i podlega zasadom, jakie obowiązują funkcje służące do tworzenia form specjalnych. Zatem argumenty COND nie są interpretowane przed wywołaniem funkcji, lecz wewnątrz niej przy pomocy funkcji EVAL interpretera.

Argumenty są interpretowane kolejno w następujący sposób. Oblicza się wartość S-wyrażenia stojącego na pierwszym miejscu pary tworzącej argument. Jeśli wartość ta jest różna od NIL, to wynikiem funkcji jest wartość S-wyrażenia będącego drugim elementem odpowiedniej pary. W przeciwnym razie kontynuuje się interpretowanie argumentów COND-u. Jeśli interpretacji nie można kontynuować, bo brak dalszych argumentów, wtedy sygnalizowany jest błąd

ERROR 14

Wyrażenie warunkowe występujące wewnątrz konstrukcji PROG jest interpretowane w innym programie, w inny sposób. Bowiem przy braku dalszych argumentów nie jest sygnalizowany błąd, lecz wykonuje się kolejną instrukcję znajdującą się za wyrażeniem warunkowym.

CONS posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <S-wyrażenie>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - funkcja konstruuje parę kropkową z dwóch S-wyrażeń.

Wynikiem formalnym funkcji jest utworzona para kropkowa.

COS posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba>

Działanie - funkcja obliwza wartość funkcji trygonometrycznej cosinus dla podanej wartości argumentu.

Obliczona wartość jest wynikiem formalnym funkcji.

Sygnalizacja

ERROR 69

<wartość argumentu>

oznacza błąd polegający na użyciu nieakceptowanej wartości argumentu.

CSET posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < symbol atomowy >

< wartość argumentu 2 > ::= < S-wyrażenie >

Działanie - funkcja definiuje wartość cechy APVAL symbolu atomowego, będącego wartością pierwszego argumentu. Przypisywaną wartością cechy jest wartość drugiego argumentu. Jest ona także wynikiem formalnym funkcji. Jeśli wartość pierszego argumentu nie jest symbolem atomowym, to następuje przerwanie wykonywania funkcji i wyprowadzenie sygnalizacji

ERROR 54

< wartość argumentu >

CSETQ posiada cechę FSUBR

Liczba argumentów: 2.

Akceptowane argumenty:

< argument 1 > ::= < symbol atomowy >

< argument 2 > ::= < S-wyrażenie >

Działanie - drugi argument jest interpretowany po wywołaniu CSETQ za pomocą funkcji EVAL. Funkcja CSETQ zapisuje wartość argumentu drugiego jako wartość cechy APVAL podanego symbolu atomowego. Różnicę w porównaniu z działaniem funkcji CSET pokazuje zależność

CSETQ[X;Y]= CSET[QUOTE[X];Y]

Wynikiem formalnym jest wartość drugiego argumentu.

DEFINE posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= ((<symbol atomowy><S-wyrażenie>))|
((<symbol atomowy><S-wyrażenie>)...
(<symbol atomowy><S-wyrażenie>))

Działanie. Każdemu symbolowi atomowemu będącemu pierwszym elementem podlisty (<symbol atomowy><S-wyrażenie>) funkcja DEFINE przypisuje cechę EXPR o wartości określonej drugim elementem tej podlisty.

Wynikiem formalnym funkcji jest lista utworzona z podanych symboli atomowych.

Sygnalizacja błędów jest taka sama jak dla funkcji DEFLIST.

DEFLIST posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::=((<symbol atomowy><S-wyrażenie>)|
((<symbol atomowy><S-wyrażenie>)
...(<symbol atomowy><S-wyrażenie>))

<wartość argumentu 2> ::= <symbol atomowy>

Działanie. Każdemu symbolowi atomowemu będącemu pierwszym elementem podlisty (<symbol atomowy><S-wyrażenie>) funkcja DEFLIST przypisuje cechę o nazwie wskazanej wartością drugiego argumentu. Wartość przypisanej cechy określona jest drugim elementem podlisty. Wynikiem formalnym jest lista symboli atomowych, którym przypisano cechy.

Jeśli pierwszym elementem podlisty nie jest symbol atomowy, to nastąpi wyprowadzenie sygnalizacji

ERROR 93
<pierwszy element podlisty>

DIFFERENCE posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba>

<wartość argumentu 2> ::= <liczba>

Działanie - funkcja oblicza różnicę dwóch liczb.

Sygnalizacja

ERROR 82

<wartość argumentu>

oznacza, że wartością argumentu nie była liczba.~~dziesiętna~~.

DIVIDE posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba>

<wartość argumentu 2> ::= <liczba>

Działanie - funkcja oblicza iloraz dwóch liczb. Wynikiem funkcji jest liczba zmiennoprzecinkowa.

Sygnalizacja

ERROR 87

<wartość argumentu>

oznacza, że wartościami argumentów nie były liczby.

ENTIER posiada cechę SUBR

Liczba argumentów: 1

Akceptowana wartość argumentu:

<wartość argumentu> ::= <liczba>

Działanie - funkcja zamienia liczby zmiennoprzecinkowe i ósemkowe na liczby całkowite dziesiętne. ~~Liste~~ Liczba zmiennoprzecinkowa zostaje zaokrąglona.

Sygnalizacje

ERROR 88
<wartość argumentu>

oznacza, że wartością argumentu nie była liczba lub wartość liczby zmiennoprzecinkowej przekracza zakres liczb całkowitych krótkich.

EQ posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <S-wyrażenie>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - predykat EQ porównuje adresy reprezentacji wewnętrznej argumentów.

Wynikiem formalnym jest T, jeśli adresy argumentów były równe, a NIL w przeciwnym wypadku.

EQUAL posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <S-wyrażenie>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - predykat bada równość dwóch dowolnych S-wyrażeń. Wynikiem formalnym jest T, gdy badane S-wyrażenie okazały się identyczne, a NIL w przeciwnym wypadku.

ERROR posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

<wartość argumentu> ::= <S-wyrażenie>

Działanie - funkcja służy do przerywania obliczeń dotyczących jednego zadania i sygnalizacji błędu. Wykonanie jej powoduje wyprowadzenie na dalekopis wydruku

ERROR

<wartość argumentu>

i przerwanie realizacji zadania.

W zależności od stanu klucza KLO system przechodzi w stan gotowości na aktualnym poziomie (stan 1) lub do kontynuacji zadania z poziomu niższego (stan 0).

Jest to pseudofunkcja nie posiadająca wyniku formalnego.

EVAL posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < S-wyrażenie >

< wartość argumentu 2 > ::= < lista par kropkowych >

Działanie - EVAL jest funkcją pomocniczą interpretera dostępną dla użytkownika. Wartość pierwszego argumentu jest interpretowana jako S-wyrażenie peprezentujące formę, natomiast wartość drugiego argumentu jako lista asocjacji. Wynikiem formalnym jest wartość interpretowanej formy. Sygnalizację błędów i szczegóły działania opisano w punkcie 5.

EVALQUOTE posiada cechę FSUBR

Liczba argumentów: 2.

Akceptowane argumenty:

< argument 1 > ::= < S-wyrażenie >

< argument 2 > ::= < lista S-wyrażeń >

Działanie - EVALQUOTE jest główną funkcją interpretera. Pierwszy argument interpretuje jako S-wyrażenie reprezentujące funkcję, natomiast drugi jako listę wartości argumentów, na które funkcja (określona pierwszym argumentem) ma działać.

Argumenty funkcji EVALQUOTE nie są interpretowane przed jej wywołaniem.

Wynikiem jest rezultat interpretacji.

Sygnalizację błędów i szczegóły działania omówiono w punkcie 5.

EVCON posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista podlist dwuelementowych>

<wartość argumentu 2> ::= <lista par kropkowych>

Działanie - EVCON jest funkcją pomocniczą interpretera dostępną dla użytkownika. Wartość pierwszego argumentu interpretuje się jako listową reprezentację wyrażenia warunkowego, natomiast wartość drugiego argumentu jako listę asocjacji.

Funkcję EVCON określa następujące wyrażenie

```
LABEL[EVCON;LAMBDA[[C;A];[NULL[C] → ERROR[14];
      EVAL[CAR[CAR[C]];A] → EVAL[CAR[CDR[CAR[C]]]; A];
      T → EVCON[CDR[C]; A]]]]
```

Wynikiem jest rezultat interpretacji.

EVLIST posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista S-wyrażeń>

<wartość argumentu 2> ::= <lista par kropkowych>

Działanie - EVLIST jest funkcją pomocniczą interpretera dostępną dla użytkownika. Wartość pierwszego argumentu traktuje się jako listę S-wyrażeń reprezentujących formy, zaś wartość drugiego argumentu jako listę asocjacji.

S-wyrażenia reprezentujące formy są kolejno poddawane interpretacji przez wykonywanie funkcji EVAL. Wynikiem jest lista rezultatów interpretacji.

EXCISE posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

<wartość argumentu> ::= <S-wyrażenie>

Działanie - jeśli wartością argumentu jest symbol atomowy COMPILE, wówczas wykonanie funkcji EXCISE powoduje nieodwracalne usunięcie kompiltora z pamięci. Obszar pamięci zajęty przez programy kompilatora zostaje objęty działaniem programów monitora przydzielających pamięć poszczególnym strukturom danych. Podobnie, jeśli wartością argumentu jest symbol atomowy EVALQUOTE, następuje usunięcie kompilatora i pełnego interpretera. W systemie pozostaje wówczas miniinterpreter,który potrafi interpretować nazwy funkcji posiadające określoną cechę SUBR. Przy innych wartościach argumentu funkcja EXCISE nie dokonuje żadnych działań.

Wynikami formalnymi funkcji EXCISE są odpowiednio symbole atomowe COMPILE, EVALQUOTE lub NIL.

EXPT posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba>

<wartość argumentu 2> ::= <liczba>

Działanie - funkcja wykonuje operację potęgowania. Wartość pierwszego argumentu jest podstawą potęgi. Wartość drugiego argumentu jest wykładnikiem potęgi. Jeśli wykładnik jest liczbą całkowitą o wartości różnej od zera, to operacja potęgowania dokonuje się poprzez wielokrotne mnożenie. Jeśli wykładnikiem jest liczba zmiennoprzecinkowa, to wykonywane jest logarytmowanie i wówczas podstawa potęgi musi spełniać warunek > x 0.

$$EXPT[X;Y] = X^Y$$

Sygnalizacja

ERROR 71
<wartość argumentu>

oznacza, że wartość argumentu nie była liczbą lub $X \leq 0$ w przypadku zmiennoprzecinkowego wykładnika potęgi.

FLAG posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista symboli atomowych>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - funkcja zapisuje podany znacznik, będący wartością drugiego argumentu, na listę własności każdego z symboli atomowych zamieszczonych na liście, która jest wartością pierwszego argumentu. Wynikiem formalnym funkcji FLAG jest NIL.

FLOATP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <S-wyrażenie>

Działanie - predykat FLOATP bada, czy wartość argumentu jest liczbą zmiennoprzecinkową.

Wynikiem formalnym jest T, jeśli wartość argumentu jest liczbą zmiennoprzecinkową, a NIL w przeciwnym wypadku.

FORMAT posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

<wartość argumentu> ::= (<n>)|(<n><n>)

<n> - liczba całkowita lub ósemkowa

Działanie - funkcja służy do ustalenia parametrów określających postać wyprowadzania liczb.

Szczegóły dotyczące wykorzystania funkcji i sygnalizacji błędów podano w punkcie 3.8.

Wynikiem formalnym funkcji jest NIL.

FUNCTION posiada cechę FSUBR

Liczba argumentów: 1

Akceptowany argument:

<argument> ::= <S-wyrażenie>

Działanie - operator FUNCTION służy do umieszczenia S-wyrażenia reprezentującego funkcję jako argumentu przy superponowaniu funkcji i argumentów. Program związany z operatorem FUNCTION jest częścią interpretera i z argumentu i listy asocjacji tworzy listę

(FUNARG <argument> <lista asocjacji>)

GENSYM posiada cechę SUBR

Liczba argumentów: 0.

Działanie - funkcja generuje nowy symbol atomowy i nadaje mu nazwę, czyli określa wartość jego cechy PNAME.

Wynikiem funkcji jest utworzony symbol atomowy.

GET posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <symbol atomowy>

<wartość argumentu 2> ::= <symbol atomowy>

Działanie - funkcja odczytuje wartość cechy symbolu atomowego (jeśli wartością argumentu 2 jest nazwa cechy) lub służy do badania, czy symbolowi atomowemu przypisano znacznik, w przypadku gdy wartością argumentu 2 jest nazwa znacznika. Jeśli wartością argumentu 2 jest nazwa cechy, to wynikiem funkcji jest wartość tej cechy, gdy badany symbol atomowy ją posiada lub NIL, gdy jej nie posiada. Jeśli wartością drugiego argumentu jest nazwa znacznika, to wynikiem jest T, jeśli symbol atomowy był oznaczony tym znacznikiem lub NIL w przypadku przeciwnym.

Sygnalizacja

ERROR 90

<wartość argumentu 1>

oznacza, że wartością pierwszego argumentu nie był symbol atomowy.

GREATERP posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba>

<wartość argumentu 2> ::= <liczba>

Działanie - predykat porównuje wartości argumentów. Wynikiem formalnym jest T, jeśli wartość argumentu 1 jest większa od wartości argumentu 2, a NIL w przeciwnym wypadku.

Sygnalizacja

ERROR 73

<wartość argumentu>

nastąpi, gdy wartość jednego z argumentów nie jest liczbą.

INPUT posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba całkowita> |
<liczba ósemkowa>

Działanie - funkcja ustala aktualność urządzenia wejściowego. Wartość argumentu jest numerem operatorskim urządzenia, zgodnym z numeracją przyjętą w ASSK-u. Wyjątek stanowią numery 10 i 11 przeznaczone do ustalenia współpracy urządzenia o numerze 1 ze stu znakowym buforem wejściowym. Wykonanie funkcji INPUT dla wartości argumentu równej 10 powoduje ponadto wyzerowanie tego bufora.

INTEGERP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <S-wyrażenie>

Działanie - predykat ten bada, czy wartość argumentu jest liczbą całkowitą,

Wynikiem formalnym jest T, jeśli wartość argumentu jest liczbą całkowitą, a NIL w przeciwnym wypadku.

LEFTSH posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba całkowita> |
<liczba ósemkowa>

<wartość argumentu 2> ::= <liczba całkowita> |
<liczba ósemkowa>

Działanie - funkcja dokonuje przesunięcie logicznego liczby będącej wartością pierwszego argumentu o ilość pozycji wskazanej wartością drugiego argumentu. Jeśli wartością drugiego argumentu jest liczba dodatnia, to dokonywane jest przesunięcie w lewo, a jeśli ujemna, to w prawo. Wynikiem formalnym jest nowa liczba. W przypadku, kiedy wartością drugiego argumentu jest zero, wynik ma tę samą wartość, co argument pierwszy.

Sygnalizacja

ERROR 86

<wartość argumentu>

oznacza, że jeden z argumentów ma wartość nieakceptowaną przez program funkcji LEFTSH.

LENGTH posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu

⟨wartość argumentu⟩ ::= ⟨lista⟩

Działanie - funkcja ta oblicza ilość elementów listy, będącej wartością argumentu.

Wynikiem jest liczba całkowita.

Sygnalizacja

ERROR 96

⟨wartość argumentu⟩

wystąpi wówczas, kiedy wartością argumentu będzie liczba lub symbol atomowy. W pozostałych przypadkach błędnych wartości argumentu sygnalizacja jest taka sama, jak dla błędu w funkcji CDR.

LESSP posiada cechę SUBR

Liczba argumentów: 2

Akceptowane wartości argumentów:

⟨wartość argumentu 1⟩ ::= ⟨liczba⟩

⟨wartość argumentu 2⟩ ::= ⟨liczba⟩

Działanie - predykat LESSP porównuje wartości dwóch liczb. Wynikiem jest T, jeżeli wartość liczby, będącej pierwszym argumentem jest mniejsza od liczby będącej wartością argumentu drugiego, natomiast NIL w przeciwnym wypadku.

Sygnalizacja

ERROR 72

⟨wartość argumentu⟩

wystąpi, jeżeli co najmniej jedna z wartości argumentów nie będzie liczbą.

LIST posiada cechę SUBR

Liczba argumentów: n $n \geq 0$

Akceptowane wartości argumentów:

⟨wartość argumentu k⟩ ::= ⟨S-wyrażenie⟩ $1 \leq k \leq n$

Działanie - funkcja tworzy listę, której elementami są wartości argumentów:

$$LIST[X_1;X_2;\ \ldots;X_n] = CONS[X_1;\ CONS[X_2;\ \ldots\ CONS[X_n;\ NIL]\ \ldots]]$$

Wynikiem funkcji jest utwirzona lista.

LOGAND posiada cechę SUBR

Liczba argumentów: n $n \geq 2$

Akceptowane wartości argumentów:

⟨wartość argumentu k⟩ ::= ⟨liczba całkowita⟩ |
⟨liczba ósemkowa⟩ $1 \leq k \leq n$

Działanie - funkcja oblicza iloczyn logiczny n liczb całkowitych i oktalnych. Wynikiem jest nowa liczba.

Sygnalizacja

ERROR 77

⟨wartość argumentu⟩

oznacza, że wartością co najmniej jednego z argumentów nie jest liczba całkowita ani ósemkowa.

ERROR 64

oznacza, że liczba argumentów była mniejsza od 2.

LOGOR posiada cechę SUBR

Liczba argumentów: n. $n \geq 2$

Akceptowane wartości argumentów:

<wartość argumentu k> ::= <liczba całkowita> |
<liczba ósemkowa> $1 \leq k \leq n$

Działanie - funkcja oblicza sumę logiczną n liczb całkowitych i ósemkowych. Wynikiem jest liczba, której wartością jest obliczona suma logiczna

Sygnalizacja

ERROR 78
<wartość argumentu>

oznacza użycie nieakceptowanej wartości argumentu

ERROR 64

oznacza, że podano mniej niż 2 argumenty.

LOGXOR posiada cechę SUBR

Liczba argumentów: n $n \geq 2$

Akceptowane wartości argumentów:

<wartość argumentu k> ::= <liczba całkowita> |
<liczba ósemkowa> $1 \leq k \leq n$

Działanie - funkcja oblicza sumę modulo dwa n liczb ósemkowych i całkowitych.

Wynikiem jest liczba, której wartością jest obliczona suma modulo dwa.

Sygnalizacja

ERROR 79
<wartość argumentu>

oznacza użycie nieakceptowanej wartości argumentu

ERROR 64

oznacza, że podano mniej niż 2 argumenty.

MAPLIST posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista>

<wartość argumentu 2> ::= <symbol atomowy> | (FUNCTION <S-wyrażenie>)

Działanie - MAPLIST jest funkcjonałem, bowiem wartość jej drugiego argumentu jest traktowana jako reprezentacja funkcji. MAPLIST tworzy nową listę, której elementami są wyniki zastosowania funkcji podanej jako drugi argument do kolejnych CDR-ów listy będącej wartością pierwszego argumentu. Funkcję MAPLIST definiuje wyrażenie

LABEL[MAPLIST; LAMBDA[[X;F]; [NULL[X] → NIL;
T → CONS[F[X]; MAPLIST[CDR[X]; F]]]]]

Wynikiem jest nowo utworzona lista.

Funkcja współpracuje z interpreterem, stąd możliwość wystąpienia błędów dotyczących interpretacji.

MAX posiada cechę SUBR

Liczba argumentów: n $n \geq 2$

Akceptowane wartości argumentów:

<wartość argumentu k> ::= <liczba> $1 \leq k \leq n$

Działanie - funkcja wyszukuje największą spośród n liczb. ~~dziesiętnych.~~

Wynikiem jest wyszukana liczba o największej wartości. Jeśli wartość chociaż jednego argumentu była liczbą zmiennoprzecinkową, to wynik jest również liczbą zmiennoprzecinkową.

Sygnalizacja

ERROR 63

<wartość argumentu>

oznacza, że wartość jednego z argumentów nie została zaakceptowana

natomiast

ERROR 64

oznacza, że podano mniej niż 2 argumenty.

MEMBER posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <S-wyrażenie>

<wartość argumentu 2> ::= <lista>

Działanie - funkcja sprawdza, czy S-wyrażenie, będące wartością pierwszego argumentu, jest identyczne z którymkolwiek z elementów podanej listy.

Wynikiem formalnym jest T, jeśli znaleziono na liście element równy wartości pierwszego argumentu, a NIL w przeciwnym wypadku. Funkcję definiuje wyrażenie

```
LABEL[MEMBER; LAMBDA[[A;X];
    [NULL[X] → NIL; EQUAL[A; CAR[X]]→T;
     T → MEMBER[A; CDR[X]]]]]
```

MIN posiada cechę SUBR

Liczba argumentów: n $n \geqslant 2$

Akceptowane wartości argumentów:

<wartość argumentu k> ::= <liczba> $1 \leq k \leq n$

Działanie - funkcja wyszukuje najmniejszą spośród n liczb. Wynikiem jest wyszukana minimalna wartość liczbowa. Jeśli wartość któregokolwiek z argumentów była liczbą zmiennoprzecinkową, to wynik jest także liczbą zmiennoprzecinkową.

Sygnalizacja

ERROR 62

<wartość argumentu>

wystąpi wówczas, gdy wartość któregokolwiek z argumentów nie jest liczbą, natomiast

ERROR 64

oznacza, że ilość argumentów była mniejsza niż 2.

MINUS posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba>

Działanie - funkcja zmienia znak liczby.

Wynikiem jest nowa liczba o przeciwnym znaku w stosunku do wartości argumentu, przy czym jest ona tego samego typu, co argument.

Sygnalizacja

ERROR 74

<wartość argumentu>

oznacza, że wartość argumentu funkcji nie jest liczbą.

MINUSP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba>

Działanie - predykat MINUSP bada znak liczby.

Wynikiem jest T, jeśli wartością argumentu jest liczba ujemna, a NIL w przypadku liczby dodatniej lub równej zeru.

Sygnalizacja

ERROR 75

<wartość argumentu>

oznacza, że wartość argumentu funkcji nie jest liczbą.

NCONC posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista>

<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - funkcja łączy dwa S-wyrażenia w jedno, bez kopiowania pierwszego z nich. Definiuje ją następujące wyrażenie funkcyjne

```
LABEL[NCONC; LAMBDA[[X;Y]; PROG[[M];
              [NULL[X]→RETURN[Y]];
               M := X ;
             A;[NULL[CDR[M]]→GO[B]];
               M := CDR[M];
               GO[A];
             B; RPLACD[M;Y];
               RETURN[X]]]]
```

NOT posiada cechę SUBR

Lista argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <S-wyrażenie>

Działanie - predykat badający, czy wartość argumentu jest równe NIL. Wynikiem jest T, jeśli wartością argumentu jest NIL lub wynikiem jest NIL w przeciwnym wypadku.

NULL posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

< wartość argumentu > ::= < S-wyrażenie >

Działanie - predykat badający, czy wartość argumentu jest równa NIL. Wynikiem jest T, jeśli wartością argumentu jest NIL lub wynikiem jest NIL w przeciwnym wypadku.

NUMBERP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= < S-wyrażenie>

Działanie - predykat NUMBERP bada, czy argument jest liczbą. Wynikiem formalnym jest T, jeśli wartość argumentu jest liczbą, a NIL w przeciwnym wypadku.

OCTALP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

< wartość argumentu > ::= < S-wyrażenie >

Działanie - predykat ten bada, czy wartość argumentu jest liczbą ósemkową.

Wynikiem formalnym jest T, jeśli wartość argumentu jest liczbą ósemkową, a NIL w przeciwnym wypadku.

ONEP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

⟨wartość argumentu⟩ ::= ⟨liczba⟩

Działanie - predykat ONEP bada, czy wartością liczby jest jedynka.

Wynikiem jest T, jeżeli wartość argumentu wynosi 1, a NIL w przeciwnym wypadku.

Sygnalizacja

ERROR 81

⟨wartość argumentu⟩

oznacza, że wartością argumentu funkcji nie była liczba.

OR posiada cechę FSUBR

Liczba argumentów: dowolna

Akceptowane wartości argumentów:

⟨wartość argumentu⟩ ::= ⟨S-wyrażenie peprezentujące formę⟩

Działanie - funkcja jest predykatem będącym uzupełnieniem operatora COND i pozwala zapisywać skrótowo wyrażenia warunkowe podanej niżej postaci.

Spełniona jest zależność

$$OR[X_1; X_2; \ldots; X_k] = [X_1 \rightarrow T; X_2 \rightarrow T; \ldots; X_k \rightarrow T; T \rightarrow NIL]$$

Argumenty funkcji OR są interpretowane po jej wywołaniu za pomocą funkcji EVAL.

OUTPUT posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba całkowita> |
<liczba ósemkowa>

Działanie - funkcja ustala aktualne urządzenia wyjściowe.
Wartością argumentu jest numer operatorski urządzenia.
Wynikiem formalnym jest wartość argumentu.
Sygnalizacja błędu

ERROR 56
<wartość argumentu>

pojawi się w przypadku wartości argumentu nieakceptowanej przez funkcję OUTPUT.

OUTBUF posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba całkowita> |
<liczba ósemkowa>

Działanie - funkcja OUTBUF ustala wskaźnik ilości znaków alfanumerycznych drukowanych w jednej linii na aktualnym wyjściu.
Wynikiem formalnym funkcji jest wartość jej argumentu.
Sygnalizacja błędu

ERROR 8
<wartość argumentu>

oznacza, że wartość argumentu była błędna.

PAIR posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < lista >

< wartość argumentu 2 > ::= < lista >

Działanie - funkcja PAIR przetwarza dwie listy o równej długości tworząc z nich listę par kropkowych utworzonych z odpowiadających sobie elementów obu list. Wynikiem jest utworzona lista.

Sygnalizacja błędów

ERROR 15

< lista par kropkowych >

oznacza, że lista będąca wartością drugiego argumentu jest dłuższa, natomiast

ERROR 16

< lista par kropkowych >

oznacza, że dłuższa jest lista będąca wartością pierwszego argumentu. W obu przypadkach wyprowadzona jest częściowo utworzona lista par kropkowych.

PLUS Posiada cechę SUBR

Liczba argumentów: n $n \geqslant 2$

Akceptowane wartośći argumentów:

<wartość argumentu k> ::= <liczba> $1 \leq k \leq n$

Działanie - funkcja oblicza sumę arytmetyczną n liczb.

Sygnalizacja błędów

Jeśli wartość któregokolwiek z argumentów nie była liczbą dziesiętną, to wyprowadzona zostanie sygnalizacja.

ERROR 60

<wartość argumentu>

Jeśli ilość argumentów będzie mniejsze niż dwa, to zostanie wyprowadzona sygnalizacja

ERROR 64

<ilość argumentów>

PRIN1 posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <symbol atomowy> | <liczba>

Działanie - funkcja wyprowadza wartość argumentu na aktualne wyjście.

Wynikiem formalnym jest wartość wyprowadzonego argumentu.

Sygnalizacje, które mogą się pojawić podczas wykonywania funkcji PRIN1, opisano w punkcie 3.

PRINCH posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < symbol atomowy >

< wartość argumentu 2 > ::= < liczba całkowita > |
< liczba ósemkowa >

Działanie - funkcja wyprowadza na aktualne wyjście n-ty znak symbolu atomowego, gdzie n jest wartością argumentu 2. Wynikiem formalnym jest T, jeśli n nie przekraczało ilości znaków w podanym symbolu atomowym, a NIL w przeciwnym wypadku (wtedy funkcja nie wywołuje żadnych działań).

Sygnalizacje, które mogą się pojawić podczas wykonywania funkcji PRINCH, opisano w punkcie 3.

PRINT posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

< wartość argumentu > ::= < S-wyrażenie >

Działanie - funkcja wyprowadza wartość argumentu na aktualne urządzenie wyjściowe.

Wynikiem formalnym jest wyprowadzone wyrażenie.

Sygnalizacje, które mogą się pojawić podczas wykonywania funkcji PRINT, opisano w punkcie 3.

PROG posiada cechę FSUBR

Liczba argumentów: n ⩾ 1.

Akceptowane argumenty:

< argument 1 > ::= < lista symboli atomowych >

< argument 2 > ::= < S-wyrażenie >

.....

< argument n > ::= < S-wyrażenie >

Działanie - funkcja PROG jest częścią interpretera i podlega zasadom, jakie obowiązują funkcje służące do tworzenia form specjalnych. Zatem argumenty PROG nie są interpretowane przed wywołaniem funkcji, lecz wewnątrz niej. Szczegóły działania funkcji opisano w punkcie 5.1. - podpunkt 13.

QUOTE posiada cechę FSUBR

Liczba argumentów: 1.

Akceptowany argument:

< argument > ::= < S-wyrażenie >

Działanie - funkcja QUOTE jest najprostszą funkcją służącą do tworzenia form specjalnych. Jej argument nie podlega interpretacji i jest wynikiem funkcji.

QUOTIENT posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

⟨wartość argumentu 1⟩ ::= ⟨liczba ósemkowa⟩ |
⟨liczba całkowita⟩

⟨wartość argumentu 2⟩ ::= ⟨liczba ósemkowa⟩ |
⟨liczba całkowita⟩

Działanie - funkcja wykonuje dzielenie całkowite, dla którego wartość pierwszego argumentu oznacza dzielną, a wartość drugiego argumentu jest dzielnikiem.

Wynikiem funkcji jest liczba całkowita lub ósemkowa.

Sygnalizacja

ERROR 83

⟨wartość argumentu⟩

oznacza, że wartości argumentów nie były liczbami całkowitymi ani oktalnymi.

RATOM Posiada cechę SUBR

Liczba argumentów: 0.

Działanie - funkcja czyta jednostkę leksykalną, tj. symbol atomowy, liczbę, dowolny tekst lub ogranicznik.

Wynikiem jest wczytana jednostka leksykalna.

Sygnalizacje, które mogą się pojawić podczas wykonywania funkcji RATOM, opisano w punkcie 3.

READ posiada cechę SUBR

Liczba argumentów: 0

Działanie - funkcja dokonuje wprowadzenia z aktualnego wejścia jednego S-wyrażenia lub jednej funkcji zapisanej w metajęzyku.

Wynikiem jest wczytane wyrażenie.

Sygnalizacje, które mogą się pojawić podczas wykonywania funkcji READ, opisano w punkcie 3.

READCH posiada cechę SUBR

Liczba argumentów: 0

Działanie - funkcja wprowadza z aktualnego wejścia jeden znak alfanumeryczny. Wynikiem jest jednoznakowy symbol atomowy reprezentujący przeczytany znak.

RECIP Posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

< wartość argumentu > ::= < liczba >

Działanie - funkcja oblicza odwrotność liczby.

Wynikiem funkcji jest liczba zmiennoprzecinkowa.

Sygnalizacja

ERROR 85

< wartość argumentu >

oznacza, że wartość argumentu funkcji RECIP nie jest liczbą.

REMAINDER posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <liczba całkowita>|
<liczba ósemkowa>

<wartość argumentu 2> ::= <liczba całkowita>|
<liczba ósemkowa>

Działanie - funkcja oblicza resztę z dzielenia całkowitego, w którym wartość pierwszego argumentu jest dzielną, a wartość drugiego argumentu dzielnikiem.

Wynikiem jest liczba całkowita lub ósemkowa.

Sygnalizacja

ERROR 84
<wartość argumentu>

oznacza, że wartościami argumentów funkcji REMAINDER nie były liczby całkowite ani ósemkowe.

REMPLAG posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista symboli atomowych>

<wartość argumentu 2> ::= <symbol atomowy>

Działanie - funkcja usuwa z list własności symboli atomowych umieszczonych na liście, która jest wartością pierwszego argumentu, znacznik, którego nazwa jest wartością drugiego argumentu.

Wynikiem formalnym funkcji jest NIL.

Sygnalizacja

ERROR 94
<element listy>

pojawi się wówczas, gdy nie wszystkie elementy listy będą symbolami atomowymi.

REMPROP posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < symbol atomowy >

< wartość argumentu 2 > ::= < symbol atomowy >

Działanie - funkcja odbiera symbolowi atomowemu, będącemu wartością pierwszego argumentu, cechę określoną wartością drugiego argumentu (usuwa nazwę i wartość cechy). Jeśli symbol atomowy nie posiada takiej cechy, wówczas nie wykonuje się żadych działań.

Wynikiem funkcji jest wartość jej pierwszego argumentu.

Sygnalizacja

ERROR 92

< wartość argumentu >

pojawi się wówczas, gdy wartość pierwszego argumentu nie jest symbolem atomowym.

RPLACA posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < para kropkowa >

< wartość argumentu 2 > ::= < S-wyrażenie >

Działanie - funkcja wpisuje wartość argumentu 2 jako nową wartość pierwszego elementu pary kropkowej.

Wynikiem jest utworzone wyrażenie kropkowe.

Sygnalizacja błędu

ERROR 52

< wartość argumentu 1 >

oznacza, że wartością pierwszego argumentu nie była para kropkowa.

RPLACD posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <para kropkowa>
<wartość argumentu 2> ::= <S-wyrażenie>

Działanie - funkcja wpisuje podane S-wyrażenie na miejsce drugiego elementu pary kropkowej.

Wynikiem jest utworzone wyrażenie kropkowe.

Zasygnalizowanie błędu

ERROR 53

<wartość argumentu 1>

oznacza, że wartością pierwszego argumentu nie była para kropkowa.

SEARCH posiada wechę SUBR

Liczba argumentów: 4.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista>
<wartość argumentu 2> ::= <S-wyrażenie>
<wartość argumentu 3> ::= <S-wyrażenie>
<wartość argumentu 4> ::= <S-wyrażenie>

Wartości argumentów 2 - 4 winny reprezentować funkcje.

Działanie - SEARCH jest funkcjonałem określonym następującym wyrażeniem

LABEL[SEARCH; LAMBDA[[X; P; F; U];
[NULL[X] → U[X]; P[X] → F[X]; T → SEARCH[CDR[X];P;F;U]]]]

SEARCH szuka na liście X podlisty o własności P. Jeśli ją znajdzie wykonuje na niej funkcję F, w przeciwnym razie wykonuje się funkcja U dla wartości argumentu NIL.

SELECT posiada cechę FSUBR

Liczba argumentów: n ⩾ 3.

Akceptowane argumenty:

< argument 1 > ::= < S-wyrażenie >

< argument 2 > ::= (< S-wyrażenie >< S-wyrażenie >)

.....

< argument n-1 > ::= (< S-wyrażenie >< S-wyrażenie >)

< argument n > ::= < S-wyrażenie >

Działanie - operator SELECT służy do tworzenia form specjalnych i można uważać, że jest on uzupełnieniem operatora COND. S-wyrażenia będące argumentami pierwszym i ostatnim, a także S-wyrażenia występujące parami w pozostałych argumentach są interpretowane wewnątrz programu SELECT przez wykonywanie funkcji EVAL.

Oblicza się kolejno wartości pierwszych S-wyrażeń w parach, poczynając od drugiego argumentu. Każdy uzyskany wynik porównuje się z wartością wyrażenia będącego pierwszym argumentem. Gdy wartości te są równe, wówczas wynikiem jest wartość wyrażenia stojącego na drugim miejscu w odpowiedniej parze. Jeśli natomiast porównywanie wartości kończy się negatywnie, to wynikiem jest wartość wyrażenia będącego ostatnim argumentem.

SET posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

< wartość argumentu 1 > ::= < symbol atomowy >

< wartość argumentu 2 > ::= < S-wyrażenie >

Działanie - funkcja zmienia asocjację na aktualnej liście asocjacji. Przegląda się tę listę, szukając pary kropkowej, w której pierwszy element jest równy wartości pierwszego argumentu funkcji. Po znalezieniu takiej pary zmienia się, przez zastosowanie funkcji RPLACD, drugi element pary, podstawiając wartość drugiego argumentu. Wynikiem formalnym jest utworzona asocjacja. Jeśli odpowiedniej pary kropkowej nie można odszukać, wtedy sygnalizowany jest błąd

ERROR 19

< wartość argumentu 1 >

SETQ posiada cechę FSUBR

Liczba argumentów: 2.

Akceptowane argumenty:

< argument 1 > ::= < symbol atomowy >

< argument 2 > ::= < S-wyrażenie >

Działanie - interpretuje się przy pomocy funkcji EVAL wartość drugiego argumentu, a następnie wykonuje się funkcję SET nad pierwszym argumentem i obliczoną wartością. Funkcja SET zmienia treść listy asocjacji wsposób opisany powyżej.

SIN posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

< wartość argumentu > ::= < liczba >

Działanie - funkcja oblicza wartość funkcji trygonometrycznej sinus dla podanej wartości argumentu.

Wynikiem jest liczba zmiennoprzecinkowa.

Sygnalizacja

ERROR 68

< wartość argumentu >

oznacza błąd polegający na wystąpieniu nieakceptowanej wartości argumentu.

SPECIAL posiada cechę SUBR

Liczba argumentów: 1.

Akceptowana wartość argumentu:

< wartość argumentu > ::= < lista symboli atomowych >

Działanie - funkcja przypisuje symbolom atomowym znajdującym się na liście znacznik SPECIAL oraz cechę APVAL z wartością NIL. Podczas swego działania wykorzystuje FLAG jako funkcję pomocniczą. Zatem sygnalizacja błędu jest taka sama jak dla funkcji FLAG.

Wynikiem formalnym jest NIL.

START posiada cechę SUBR

Liczba argumentów: 0.

Działanie - pseudofunkcja START inicjuje realizację funkcji bezpośrednio po niej następującej z jednoczesnym nakazem korzystania z urządzeń wejścia/wyjścia wybranych podczas ostatniego wykonywania funkcji INPUT i OUTPUT.

Pseudofunkcja START nie posiada wyniku formalnego.

SUB1 posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <liczba>

Działanie - funkcja zmniejsza wartość argumentu o 1.

Wynikiem funkcji jest nowa liczba tego samego typu, co argument.

Sygnalizacje

ERROR 67

<wartość argumentu>

oznacza, że wartością argumentu funkcji nie była liczba.

TIMES posiada cechę SUBR

Liczba argumentów: n $n \geqslant 2$

Akceptowane wartości argumentów:

<wartość argumentu k> ::= <liczba> $1 \leqslant k \leqslant n$

Działanie - funkcja oblicza iloczyn n liczb.

Sygnalizacja

ERROR 61

<wartość argumentu>

oznacza, że nie wszystkie wartości argumentów były liczbami. Natomiast sygnalizacja błędu

ERROR 64

<ilość argumentów>

wystąpi w przypadku, gdy podstawiono mniej niż 2 argumenty.

TRACE posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie - funkcja przypisuje symbolom atomowym znajdującym się na liście znacznik TRACE.

Wynikiem formalnym funkcji TRACE jest NIL.

Sygnalizacja błędu

ERROR 95

<element listy>

wystąpi w przypadku, kiedy elementami listy nie były wyłącznie symbole atomowe.

UNCOMMON posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie - funkcja usuwa znacznik COMMON przypisany symbolom atomowym. Działanie to dotyczy wszystkich symboli atomowych znajdujących się na liście, która jest wartością argumentu. Funkcja UNCOMMON wywołuje funkcję REMFLAG, zatem może wystąpić sygnalizacja błędu dotycząca REMFLAG.

UNION posiada cechę SUBR

Liczba argumentów: 2.

Akceptowane wartości argumentów:

<wartość argumentu 1> ::= <lista>

<wartość argumentu 2> ::= <lista>

Działanie - funkcja UNION tworzy nową listę, złożoną z tych elementów pierwszej listy, które nie występują na drugiej liście oraz wszystkich elementów drugiej listy.

Funkcję UNION określa następujące wyrażenie

```
LABEL[UNION; LAMBDA[[X;Y];[NULL[X] → Y;
                  MEMBER[CAR[X];Y] → UNION[CDR[X];Y];
                  T → CONS[CAR[X]; UNION[CDR[X]; Y]]]]]
```

Wynikiem jest utworzona lista.

UNSPECIAL posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie - funkcja usuwa znacznik SPECIAL przypisany symbolom atomowym i podstawia NIL jako wartość cechy APVAL. Działanie to dotyczy wszystkich symboli atomowych znajdujących się na liście, która jest wartością argumentu. Funkcja UNSPECIAL wywołuje funkcję REMFLAG, zatem może wystąpić sygnalizacja błędu dotycząca REMFLAG.

UNTRACE posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::= <lista symboli atomowych>

Działanie - funkcja usuwa znacznik TRACE przypisany symbolom atomowym. Działanie to dotyczy wszystkich symboli atomowych znajdujących się na liście, która jest wartością argumentu.

Sygnalizacja błędu

ERROR 95

<element listy>

pojawi się wówczas, kiedy elementami listy nie były wyłącznie symbole atomowe.

ZEROP posiada cechę SUBR

Liczba argumentów: 1.

Akceptowane wartości argumentu:

<wartość argumentu> ::=<liczba>

Działanie - predykat ZEROP bada, czy wartością argumentu jest zero.

Wynikiem formalnym jest T, jeśli wartość argumentu wynosiła zero, a NIL w przeciwnym przypadku.

Sygnalizacja

ERROR 80

<wartość argumentu>

oznacza, że wartośćargumentu funkcji nie była liczbą.

5. Zasady interpretacji i kompilacji

5.1. Interpretacja funkcji zapisanych jako S-wyrażenie

Funkcja systemowa EVALQUOTE jest interpreterem, który wykonuje funkcję zapisaną jako S-wyrażenie od podanych wartości argumentów. EVALQUOTE jest dwuargumentowa - przyjmuje się, że pierwszy argument określa funkcję, a drugi listę wartości argumentów tej funkcji. Należy pamiętać, że nie każde S-wyrażenie, wczytane bezpośrednio lub przetłumaczone przez funkcję READ z metajęzyka stanowi daną, która może być zaakceptowana przez interpreter lub kompilator jako wyrażenie funkcyjne. Zbiór danych akceptowanych w tym sensie można wskazać dokładniej przez podanie gramatyki, którą posłużono się przy układaniu algorytmów interpretera i kompilatora. Ponieważ jednak jest to gramatyka bezkontekstowa, zatem nie ma gwarancji, że wyrażenie z niej wygenerowane jest bezbłędne, czyli że nie spowoduje zatrzymania się interpretera lub kompilatora z sygnalizacją błędu.

Gramatyka ta posiada następujące produkcje:

<funkcja> ::= <symbol atomowy> |
(LAMBDA <lista zm> <forma>) |
(LABEL <symbol atomowy> <funkcja>)

<forma> ::= <symbol atomowy> |
<liczba> |
(QUOTE <S-wyrażenie>) |
(<funkcja>.<lista argum>) |
(COND .<lista par form>) |
(PROG <lista zm>.<lista form>)

<argum> ::= <forma> |
(FUNCTION <funkcja>)
<lista argum> ::= NIL | (<argum> . <lista argum>)
<lista zm> ::= NIL | (<symbol atomowy> . <lista zm>)
<lista par form> ::= ((<forma> <forma>)) |
((<forma> <forma>) .
<lista par form>)
<lista form> ::= NIL | (<forma1> . <lista form>)
<forma1> ::= <forma> |
(COND . <lista par form1>) |
(GO <symbol atomowy>) |
(SET <forma> <forma>) |
(SETQ <symbol atomowy> <forma>) |
(RETURN <forma>)
<lista par form1> ::= ((<forma> <forma1>)) |
((<forma> <forma1>) .
<lista par form1>)

Szczególną uwagę należy zwrócić na zmienną <forma1> wprowadzoną celem pokazania różnicy między zwykłą formą i taką, która może występować wewnątrz konstrukcji PROG. Aby uprościć gramatykę, w niektórych jej produkcjach zastosowano zapis kropkowy. Oczywiście częściej stosuje się równoważny zapis listowy wyrażeń.

Przejdziemy teraz do omówienie szczegółów związanych z działaniem interpretera.

1. Specjalna lista, zwana listą asocjacji interpretera pełni rolę dynamicznej tablicy zawierającej zmienne i ich wartości. Jest to lista par kropkowych zwanych asocjacjami. Pierwszy element pary jest nazwą zmiennej,

zaś drugi element jej aktualną wartością. Odczytując wartość przypisaną pewnej zmiennej, listę tą przegląda się od początku. Także do początku dopisuje się nowe asocjacje. Zatem lista ta działa jak stos.
Powrót z niższego na wyższy poziom interpretacji odbywa się zawsze z odtworzeniem poprzedniego stanu listy asocjacji.

2. Główna funkcja interpretera - EVALQUOTE - współpracuje z dwoma zasadniczymi funkcjami pomocniczymi: APPLY i EVAL.
APPLY jest funkcją trójargumentową i służy do interpretacji wyrażeń wygenerowanych przez zmienną <funkcja> ostatnio omówionej gramatyki. Interpretowane wyrażenie winno być wartością pierwszego argumentu APPLY. Jako drugi argument podstawia się listę wartości argumentów (interpretowanej funkcji), natomiast jako trzeci - aktualną listę asocjacji.
EVAL jest funkcją dwuargumentową służącą do interpretowania wyrażeń postaci <argum>. Wartością pierwszego argumentu EVAL winno być interpretowane wyrażenie, zaś wartością drugiego - aktualna lista asocjacji.

3. Wyjaśnimy teraz różnicę między cechami SUBR i EXPR a FSUBR i FEXPR.
Przedrostek F w nazwie oznacza, że symbol atomowy funkcji posiadający jedną z tych cech służy do tworzenia tzw. form specjalnych. Formą specjalną jest konstrukcja

(<symbol atomowy> . <lista argumentów>),

jeśli symbol atomowy otwierający tę formę posiada cechę FSUBR lub FEXPR.

Zwykłą formę tej postaci interpretuje się obliczając najpierw listę wartości dla wyrażenia typu <lista argumentów>, a następnie oblicza się funkcję od tych wartości, wskazaną symbolem atomowym. Natomiast formę specjalną oblicza się w odwrotnej kolejności: najpierw wywołuje się funkcję wskazaną symbolem atomowym. Wyrażenie postaci <lista argumentów> zostaje przekazane tej funkcji bez uprzedniej interpretacji. Odbywa się ona "wewnątrz" funkcji z uwzględnieniem aktualnej treści listy asocjacji.

4. Załóżmy, że funkcja EVALQUOTE zostaje wywołana dla argumentów postaci <funkcja> i <lista>, przy czym drugi argument jest rozumiany jako lista wartości argumentów, na które funkcja ma działać. Najpierw bada się, czy pierwszy argument EVALQUOTE jest symbolem atomowym i czy posiada cechę FSUBR lub FEXPR.

   Jeśli tak jest, wtedy ten symbol atomowy dołącza się do listy będącej drugim argumentem EVALQUOTE i wzywa funkcję EVAL z pustą listą asocjacji. Jeśli tak nie jest, to wzywa się funkcję APPLY podstawiając za jej argumenty wartości obu argumentów EVALQUOTE i pustą listę asocjacji.

5. Wykonując funkcję APPLY bada się najpierw, czy wartością jej pierwszego argumentu jest symbol atomowy. Jeśli to jest prawdą i symbol ten posiada cechę SUBR, to wzywa się program będący wartością tej cechy podstawiając za argumenty wartości określone drugim argumentem APPLY. Może przy tym wystąpić sygnalizacja błędu.

   ERROR 17
   <symbol atomowy>

   jeśli podstawiono złą liczbę argumentów.

Jeśli zaś ów symbol atomowy posiada cechę EXPR, to odczytuje się wartość cechy i ponownie wzywa funkcję APPLY, aby wartość tą zinterpretować. Podstawia się te same (stare) wartości za dwa pozostałe argumenty APPLY.

Jeśli natomiast wspomniany symbol atomowy nie posiada żadnej z cech SUBR lub EXPR, to przegląda się listę asocjacji celem odszukania na niej jego znaczenia. Gdy to poszukiwanie kończy się sukcesem, wtedy znalezioną wartość traktuje się jak wyrażenie postaci ⟨funkcja⟩ i ponownie wzywa APPLY przy starych wartościach dwóch pozostałych argumentów. Gdy brak sukcesu w przeglądaniu listy asocjacji, wtedy zostaje zasygnalizowany błąd

ERROR 11

⟨symbol atomowy⟩

z wyprowadzeniem nieokreślonego symbolu atomowego i przerywa się interpretację.

6. Jeśli jako wartość pierwszego argumentu APPLY pojawi się wyrażenie postaci

(LABEL ⟨symbol atomowy⟩⟨funkcja⟩)

wtedy do aktualnej listy asocjacji dopisuje się parę kropkową

(⟨symbol atomowy⟩ . ⟨funkcja⟩),

której elementy powstały z odpowiednich elementów konstrukcji LABEL . Można zatem powiedzieć, że operator LABEL przypisuje dynamiczną etykietę wyrażeniu funkcyjnemu.

To wyrażenie funkcyjne ze starą listą wartości argumentów i nową listą asocjacji podstawia się znów do funkcji APPLY.

7. Jeśli jako wartość pierwszego argumentu APPLY pojawi się konstrukcja

(LAMBDA <lista zm><forma>)

wówczas najpierw następuje wiązanie zmiennych. Zmienne występujące w wyrażeniu postaci <lista zm> łączy się w pary kropkowe z odpowiednimi wartościami argumentów. Wartości te odczytuje się z listy, podstawionej jako drugi argument APPLY. Przy tym może zdarzyć się, że obie listy mają różną liczbę elementów. Nastąpi wtedy sygnalizacja

ERROR 15
<lista par kropkowych>

lub ERROR 16
<lista par kropkowych>

w zależności od tego, czy dłuższa była lista wartości argumentów (pierwszy przypadek) czy lista zmiennych. W obu przypadkach wyprowadza się ponadto skonstruowaną do tego momentu listę par kropkowych. Jeśli wiązanie zmiennych może się zakończyć, wtedy aktualną listę asocjacji powiększa się o dokonane asocjacje i przechodzi do interpretacji przez funkcję EVAL podwyrażenia postaci <forma> rozpatrywanej konstrukcji.

8. Jako wartość pierwszego argumentu APPLY może także pojawić się wyrażenie postaci

(FUNARG <funkcja> <lista asocjacji>)

wprowadzone do APPLY przez umyślną akcję funkcji EVAL. Wyrażenie to zostaje "rozpakowane". Jego część postaci <funkcja>, wartość drugiego argumentu APPLY i część postaci <lista asocjacji> stają się nowymi wartościami argumentów funkcji APPLY ponownie wywoływanej w tym miejscu.

9. Wykonując funkcję EVAL bada się najpierw, czy wartością jej pierwszego argumentu jest symbol atomowy T lub NIL, albo

czy jest nią liczba. W każdym z tych przypadków wartość pierwszego argumentu EVAL jest wynikiem.

Następnie bada się czy EVAL ma dokonać interpretacji symbolu atomowego. Jeśli tak, to symbol ten albo powinien posiadać cechę APVAL, której wartość jest wynikiem, albo szuka się jego znaczenia na aktualnej liście asocjacji. Gdy owego symbolu atomowego nie można na niej odszukać następuje sygnalizacja

ERROR 12
<symbol atomowy>

z wyprowadzeniem owego nieokreślonego symbolu atomowego.

10. Jeśli wartością pierwszego argumentu EVAL jest wyrażenie postaci

(QUOTE < S-wyrażenie>)

to wynikiem EVAL staje się część tego wyrażenia postaci <S-wyrażenie> .

11. Dla konstrukcji

(FUNCTION <funkcja>)

zostaje utworzona lista

(FUNARG <funkcja> <lista asocjacji>)

która jest wynikiem EVAL.

12. Jeśli wartość pierwszego argumentu EVAL ma postać

(COND .< lista par form>)

to wtedy z aktualną listą asocjacji przechodzi się do interpretowania listy par form. Czynności te wykonuje funkcja EVCON. Jest to dwuargumentowa funkcja pomocnicza: wartością jej pierwszego argumentu jest lista par form, a wartością drugiego lista asocjacji.

W EVCON bada się napierw, czy lista par form jest pusta i jeśli jest pusta sygnalizuje błąd

ERROR 14

W przeciwnym razie wykonuje się funkcję EVAL dla pierwszego elementu pierwszej pary form i bieżącej listy asocjacji. Jeśli wynik tego działania jest różny od NIL, to wynikiem EVCON jest rezultat obliczenia funkcji EVAL dla drugiego elementu pierwszej pary form i listy asocjacji. Jeśli natomiast jest inaczej, to kontynuuje się wykonywanie funkcji EVCON po skróceniu listy par form o jedną parę.

Zatem interpretacja wyrażenia z operatorem COND odbywa się w EVAL następująco: szuka się systematycznie warunku (pierwszego elementu w parze form), którego wartość jest różna od NIL. Po znalezieniu go, oblicza się wartość drugiego elementu tej pary form i jest ona wynikiem. Jeśli natomiast nie można znaleźć takiego ~~wyrażenia~~ warunku, to wartość całego wyrażenia warunkowego jest nieokreślona.

13. Jeśli wartością pierwszego argumentu EVAL jest konstrukcja postaci

(PROG <lista zm> <lista form>)

to interpretacji dokonuje się według następujących zasad:

- przetwarza się listę zmiennych znajdującą się za PROG. Każdą zmienną łączy się w parę kropkową z wartością początkową równą NIL i otrzymane w ten sposób pary kropkowe dołącza do aktualnej listy asocjacji,
- listę form będącą trzecim elementem konstrukcji PROG przeszukuje się, aby znaleźć symbole atomowe pełniące rolę etykiet miejsc na tej liście. Każdą napotkaną etykietę łączy się w parę kropkową z podlistą rozpoczynającą się bezpośrednio za tą etykietą. Z takich par tworzy się tzw. listę skoków (GO-listę),

- dokonuje się kolejno interpretacji form tworzących listę form,
- napotkanie operatora SETQ powoduje odszukanie zmiennej na liście asocjacji. Oblicza się wartość drugiego argumentu operatora SETQ (przy pomocy funkcji EVAL) i wartość tę przypisuje zmiennej, podstawiając ją w odpowiedniej parze kropkowej na liście asocjacji. Jeśli zmienna została związana na liście asocjacji kilkakrotnie, to podstawienie dotyczy najwyższej asocjacji. Może się zdarzyć, że związanie zmiennej dotyczy wyższego poziomu obliczeń. Wtedy wykonanie operatora SETQ zmieni wartość zmiennej na wyższym poziomie.

  Jeśli zmienna na liście asocjacji nie występuje, to nastąpi sygnalizacja błędu

  ERROR 19
  <zmienna>

  Podobną rolę spełnia operator SET różniący się od SETQ tym, że przed jego wykonaniem dokonuje się interpretacji obu argumentów, a nie tylko drugiego, jak w przypadku SETQ,
- napotkanie operatora RETURN powoduje obliczenie wartości jego argumentu (przy pomocy funkcji EVAL) i wyjście z interpretacji całego wyrażenia PROG z otrzymaną wartością jako wynikiem,
- wyrażenie warunkowe występujące na liście form wewnątrz PROG jest interpretowane nieco inaczej niż przez funkcję EVAL. Nieznalezienie bowiem warunku o wartości różnej od NIL nie powoduje sygnalizacji błędu, lecz przejście do interpretacji formy stojącej na liście form bezpośrednio za wyrażeniem warunkowym. Ponadto, jeżeli wyrażenie

warunkowe jest elementem listy form, to jego podwyrażenia nie będące warunkami mogą mieć postać <forma1> , czyli można je tworzyć z wykorzystaniem operatorów SET, SETQ, RETURN i GO.

- napotkanie na liście form operatora GO powoduje odszukanie etykiety na liście skoków i kontynuowanie interpretacji od miejsca wskazanego tą etykietą. Ponieważ lista skoków ma charakter lokalny, zatem skoki nie mogą dotyczyć etykiet z innych poziomów interpretacji.

  Jeśli skok ma być wykonany do miejsca niezdefiniowanego to wystąpi sygnalizacja błędu

  ERROR 18
  <etykieta>

- jeśli na liście form występuje wyrażenie innej postaci, niż dotąd omówione, to oblicza się jego wartość za pomocą funkcji EVAL, a wynik pomija,
- jeśli proces interpretacji dotrze do końca listy form, to wynikiem interpretacji całej konstrukcji PROG jest NIL.

14. W punktach 9-13 omówiliśmy początkowe przypadki działania funkcji EVAL. Jeśli żaden z nich nie zachodzi, wtedy bada się czy wartość pierwszego argumentu EVAL posiada postać

(<symbol atomowy> . <liste argum>)

czyli, czy jest listą rozpoczynającą się symbolem atomowym. Jeśli to jest prawdą, wtedy stwierdza się kolejno, czy symbol ten określa jedna z cech: SUBR, FSUBR, EXPR, FEXPR. Gdy jednak żadna z tych cech nie określa symbolu atomowego, wtedy szuka się jego znaczania na aktualnej liście asocjacji i odszukana wartość służy do skonstruowania formy postaci

(<odszukana wartość> . <liste argum>)

gdzie jako listę argumentów bierze się resztę rozpatrywanej formy (otrzymaną po pominięciu początkowego symbolu atomowego).

Może przy tym wystąpić sygnalizacja błędu

ERROR 13
<symbol atomowy>

jeśli ów symbol atomowy nie jest określony nawet na liście asocjacji.

Wróćmy teraz do rozpatrzenia przypadków, w których symbol atomowy stojący na początku listy jest określony jedną z cech: SUBR, FSUBR, EXPR, FEXPR.
Jeśli jest to cecha SUBR lub EXPR, to najpierw dokonuje się interpretacji listy argumentów (czyli podlisty powstającej przez odrzucenie pierwszego elementu). Interpretację tę przeprowadza się z wykorzystaniem funkcji pomocniczej EVLIST
W ten sposób przygotowuje się wartości argumentów. Jeśli stwierdzono występowanie cechy SUBR, to wzywa się program określony wartością tej cechy, jeśli natomiast występuje cecha EXPR, to dalsze czynności załatwia funkcja APPLY operując na: wartości cechy EXPR, przygotowanej liście wartości argumentów i aktualnej liście asocjacji.

Podobnie, choć nieco inaczej, ma się rzecz w przypadku występowania cechy FSUBR lub FEXPR. Wtedy nie dokonuje się interpretacji argumentów przed wywołaniem funkcji.

15. Wreszcie, jeśli wartość pierwszego argumentu EVAL nie rozpoczyna się symbolem atomowym, czyli ma postać

(<funkcja> . <lista argum>)

i funkcja nie jest symbolem atomowym, to dokonuje się (przy pomocy EVLIST)interpretacji listy argumentów, a następnie wzywa funkcję APPLY podstawiając odpowiednio jako

argumenty: wyrażenie postaci <funkcja> , zinterpretowaną listę argumentów i aktualną listę asocjacji.

Aby opis działania interpretera poprzeć przykładem przytoczymy wydruk otrzymany przy wykonywaniu zadania. Najpierw zdefiniowano funkcję MEMB, następnie wykonano funkcję TRACE nad listą (APPLY EVAL), aby otrzymać ślad obliczeń interpretera, a w końcu wykonano funkcję MEMB od E i (E)

```
LISP 1:
DEFINE ((
(MEMB LAMBDA[[A ; X];
[NULL[X] → NIL; EQ[A; CAR[X]]→T; T → MEMB[A; CDR[X]]]])
))
(MEMB)
LISP 1:
TRACE ((APPLY EVAL))
NIL
LISP 1:
MEMB (E(E))
ARGUMENTS OF APPLY
(MEMB (E(E)) NIL)
ARGUMENTS OF APPLY
((LAMBDA (A X)(COND ((NULL X) NIL)((EQ A(CAR X )) T)
(T (MEMB A (CDR X ))))) (E(E)) NIL)
ARGUMENTS OF EVAL
((COND ((NULL X)NIL) ((EQ A(CAR X)) T)
(T (MEMB A(CDR X)))) ((X E) (A.E)))
ARGUMENTS OF EVAL
((NULL X) ((X E)(A.E)))
ARGUMENTS OF EVAL
```

```
(X ((X E)(A.E)))
VALUE OF EVAL
(E)
VALUE OF EVAL
NIL
ARGUMENTS OF EVAL
((EQ A (CAR X)) ((X E) (A.E)))
ARGUMENTS OF EVAL
(A ((X E) (A.E)))
VALUE OF EVAL
E
ARGUMENTS OF EVAL
((CAR X) ((X E) (A.E)))
ARGUMENTS OF EVAL
(X ((X E) (A.E)))
VALUE OF EVAL
(E)
VALUE OF EVAL
E
VALUE OF EVAL
T
ARGUMENTS OF EVAL
(T ((X E) (A.E)))
VALUE OF EVAL
T
VALUE OF EVAL
T
VALUE OF APPLY
T
VALUE OF APPLY
T
T
LISP 1:
```

5.2. Kompilacja i wykonywanie funkcji przetłumaczonych

Kompilator jest funkcją systemową o nazwie COMPILE, jednoargumentową. Wartością argumentu powinna być lista symboli atomowych określonych cechą EXPR lub FEXPR. Jeśli jeden z elementów tej listy nie jest symbolem atomowym, wtedy nastąpi sygnalizacja błędu

ERROR 101

< element >

a w przypadku, gdy jest symbolem atomowym, lecz nie jest określony cechą EXPR lub FEXPR, wyprowadza się sygnalizację

ERROR 102

< symbol atomowy >

Ponadto czynności dotyczące każdego symbolu atomowego z tej listy są traktowane jako oddzielne podzadanie. Oznacza to, że wystąpienie błędu nie przerywa całego procesu kompilacji, lecz jedynie tłumaczenie jednej funkcji. Wtedy, oprócz normalnej sygnalizacji błędu wyprowadza się jeszcze symbol atomowy funkcji, której tłumaczenia nie można kontynuować.

Wynikiem formalnym funkcji COMPILE jest NIL, jednak istotny jest efekt uboczny jej działania, czyli wygenerowane programy. W rezultacie wykonania COMPILE symbole atomowe przetwarzane przez tę funkcję uzyskają nową cechę SUBR lub FSUBR nie tracąc posiadanej cechy EXPR lub FEXPR. Wartością nowej cechy jest adres programu zapisanego w pamięci jako ciąg makroinstrukcji.

W fazie wykonywania programu przetłumaczonego te makroinstrukcje są wykonywane przez specjalny program monitora zwany realizatorem makroinstrukcji.

Wyrażenie definiujące funkcję zostaje przetłumaczone tak, że program wynikowy nie zależy od przekładu innych funkcji. Ułatwia to wprowadzenie zmian i poprawek do definicji i przekładu funkcji będącej częścią większego problemu. W wyrażeniach poddawanych kompilacji mogą nawet występować nazwy funkcji niezedfiniowanych pod warunkiem, że zostaną później określone cechą EXPR /inaczej program przetłumaczony nie będzie działał poprawnie/.

Oznakowanie symbolu atomowego znacznikiem SPECIAL służy do poinformowania kompilatora, że podczas wykonywania programu przetłumaczonego, symbol ten będzie posiadał cechę APVAL /czyli będzie pełnił rolę zmiennej globalnej/.

Funkcje przetłumaczone mogą współdziałać z funkcjami nieprzetłumaczonymi dzięki możliwości wezwania interpretera z programu skompilowanego. Zasady tej współpracy są następujące.

Znacznik COMMON przypisany symbolowi atomowemu, będącemu nazwą zmiennej zawiadamia kompilator, że należy tak tłumaczyć program, by podczas wykonywania go aktualne wartości zmiennej były przekazywane interpreterowi. Dotyczy to jednak tylko zmiennych związanych operatorem LAMBDA.

Interpreter wzywając program przetłumaczony podstawia aktualną listę asocjacji jako wartość cechy APVAL symbolu atomowego ALIST. Dzięki temu program przetłumaczony może, jeśli to jest konieczne, dopisywać lub zmieniać asocjacje na liście asocjacji, albo odczytywać jej treść.

Jeśli w tłumaczonej funkcji występują formy specjalne, to zmienne tej funkcji winny posiadać znaczniki COMMON, bowiem do obliczenia form specjalnych wzywany jest interpreter. Uwaga ta nie dotyczy form specjalnych utworzonych przy pomocy QUOTE, PROG, COND, CSETQ, AND i OR.

Użycie znacznika COMMON jest także konieczne dla zmiennych, które występując w argumencie będącym funkcją są związane poza tą funkcją. Na przykład przed tłumaczeniem wyrażenia

```
(LAMBDA (X Y)(MAPLIST X (FUNCTION (LAMBDA
        (J)(CONS (CAR J) Y )) )))
```

należy zmiennej Y przypisać znacznik COMMON, ponieważ w argumencie funkcyjnym

```
(LAMBDA (J)(CONS (CAR J) Y))
```

jest ona zmienną niezwiązaną.

Pobobnie, jak funkcje nieprzetłumaczone, również funkcje skompilowane można oznaczać znacznikiem TRACE celem uzyskania śladu obliczeń.

Podczas kompilacji programu może wystąpić, oprócz podanych, jeszcze jeden z błędów o numerach 100 lub 103-108. Błędy te zestawiono w wykazie.

Sygnalizacja błędów pojawi się także w kilku sytuacjach podczas wykonywania programów przetłumaczonych /nie licząc sygnalizacji wewnątrz funkcji systemowych wzywanych przez programy przetłumaczone/. Sytuacje te zestawiono także w wykazie sygnalizacji błędów pod numerami 20-22 i 27.

6. Opis realizacji systemu

6.1. Podział pamięci

System LISP 1.5 opracowany został dla minikomputera K-202, wyposażonego w pamięć operacyjną o pojemności od 16 do 64 K słów (w wersji dla pamięci jednopoziomowej). Pojemność pamięci przyjęto za parametr, który można zmieniać.

Początkowy obszar bloku pamięci operacyjnej przeznaczono na programy systemu LISP. Wielkość tego obszaru wynosi około 11 K słów. Pozostałą część pamięci podzielono na stronice po 128 słów. Wprowadzenie paginacji uzasadnione jest istnieniem różnych struktur danych w pamięci. Paginacja zwalnia od konieczności ustalenia sztywnego podziału pamięci między poszczególne typy danych i umożliwia jej dynamiczną alokację.

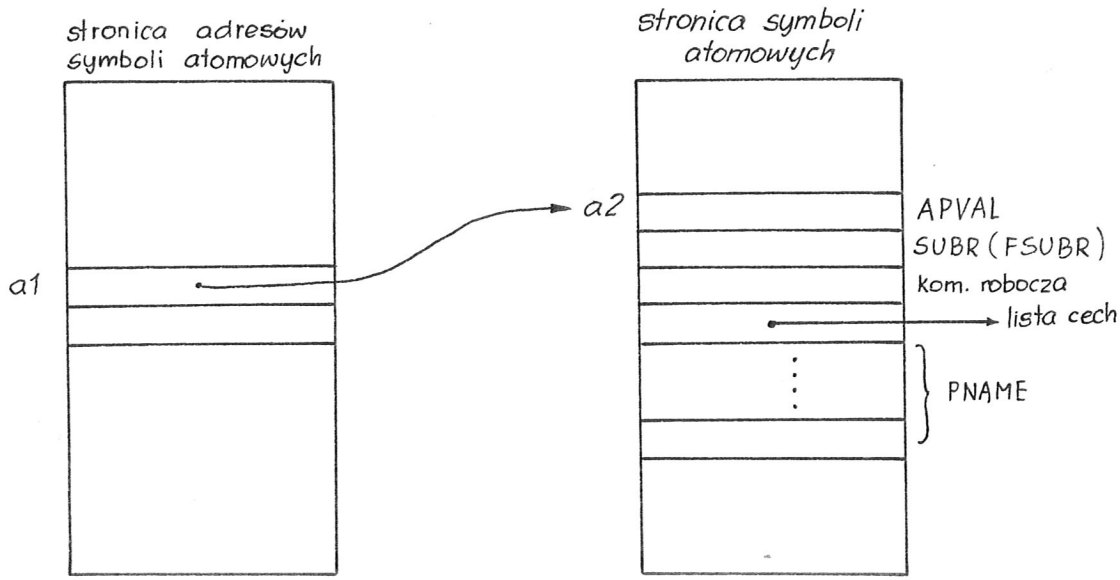
W systemie wyróżniono następujące typy stron (typ strony określony jest przez rodzaj struktury na niej zapisanej):

- stosu funkcyjnego,
- stosu argumentów,
- list (dla danych symbolicznych złożonych),
- tablic z adresami symboli atomowych,
- symboli atomowych,
- liczb całkowitych dziesiętnych,
- liczb całkowitych ósemkowych,
- liczb zmiennoprzecinkowych,
- przetłumaczonego programu użytkowego,
- puste.

6.2. Struktury danych w pamięci

Symbole atomowe

Symbole atomowe systemu oraz symbole atomowe wprowadzone przez użytkownika traktowane są jednakowo i umieszczone na wspólnych stronicach. Wprowadzenie symbolu atomowego do pamięci wymaga dokonania zapisów na stronicach dwu typów (rys.6.1)

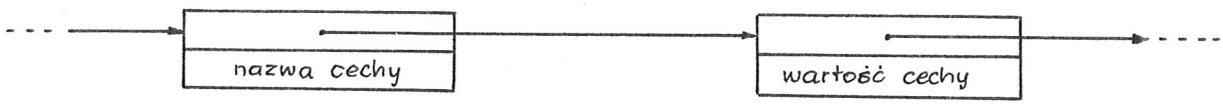


rys.6.1. zapis symbolu atomowego w pamięci.

Stronica adresów symboli atomowych używana jest tylko przez programy: wprowadzający i zbierania śmieci. Zapisane na niej adresy (adres a2 na rysunku) reprezentują poszczególne symbole atomowe w systemie. Rozmieszczenie tych zapisów na stronicy (wyliczanie adresu a1) odbywa się za pomocą techniki kodowania mieszającego. Pierwsze dwie stronice adresów symboli atomowych (zawierające m.in. adresy symboli atomowych systemu) znajdują się w obszarze pamięci zajętej przez system. Po ich zapełnieniu wydziela się

następne stronice adresów symboli atomowych już z obszaru stronicowanego.

Stronice symboli atomowych zawierają wybrane cechy symboli atomowych. Sposób zapisu cech jest identyczny dla wszystkich symboli atomowych. Począwszy od komórki wskazanej adresem a/2 umieszczane są kolejno: wartość cechy APVAL danego symbolu atomowego, wartość cechy SUBR (lub FSUBR), znacznik TRACE wraz z informacjami roboczymi wykorzystywanymi przez programy systemu, łącznik do listy cech tego symbolu oraz począwszy od komórki o adresie a2+4 wartość cechy PNAME, czyli ciąg znaków alfenumerycznych, tworzących dany symbol atomowy (po dwa znaki w jednej komórce). Inne cechy symbolu atomowego, jak cecha EXPR oraz cechy wprowadzone przez użytkownika, zapisywane są na liście cech symbolu atomowego w sposób pokazany na rys.6.2. Brak cechy APVAL, SUBR lub znacznika TRACE jest sygnalizowany przez zerową treść odpowiedniej komórki lub bitu.



rys.6.2. Sposób umieszczenia zapisów na liście cech symbolu atomowego.

Znaczniki (oprócz TRACE) umieszcza się również na liście cech.

Liczby

W reprezentacji wewnętrznej liczb wyodrębniono trzy ich rodzaje: dziesiętne całkowite, ósemkowe całkowite i zmienno-

przecinkowe. Liczby całkowite obu typów zapisywane są w pojedyńczych komórkach pamięci, co jednocześnie determinuje ich wartość bezwzględną mniejszą od 32768. Liczby zmiennoprzecinkowe zajmują po 3 kolejne komórki pamięci. Reprezentacją liczby w realizacji wewnętrznej jest adres komórki (pierwszej dla liczb zmiennoprzecinkowych) zawierającej liczbę.

Stosy

W systemie wyodrębniono dwa stosy: argumentów oraz funkcji. Oba stosy mają identyczną strukturę, różnią się jedynie sposobem wykorzystania. Na stosie funkcji umieszczane są adresy makroinstrukcji z ciągu makroinstrukcji (patrz 6.2) oraz adresy programów realizujących funkcje systemowe LISP-u. Na stosie argumentów zapisywane są adresy struktur danych będących wartościami argumentów lub wynikami poszczególnych funkcji.

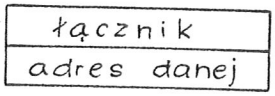
W momencie wywoływania dowolnej funkcji adres wartości jej argumentu (w przypadku funkcji wieloargumentowej - ostatniego) znajduje się na szczycie stosu argumentów. Po jej wykonaniu na szczycie stosu argumentów znajduje się adres struktury będącej wynikiem tej funkcji. Dokonywanie operacji zapisu i odczytu na obu stosach opiera się na zasadzie LIFO. Stos może zajmować kilka stron znajdujących się w różnych miejscach pamięci. Zerowa komórka każdej strony stosu zawiera łącznik do strony poprzedniej, tzn. przydzielonej stosowi bezpośrednio przed daną stroną. W zerowej komórce najwcześniej przydzielonej strony zapisany jest adres symbolu atomowego NIL. Przydział stronic

dla poszczególnych typów struktur oraz identyfikacja typów stron dokonuje się w programach monitora.

Listy

Budowę pojedyńczego elementu listy przedstawia rys. 6.3.
Pojedyńczy element listy zajmuje dwie komórki pamięci. W pierwszej z nich zapisany jest łącznik do następnego elementu listy (adres tego elementu), a w drugiej adres danej, czyli wartości elementu listy.

Identyfikatorem końca listy jest symbol atomowy NIL (czyli wartość łącznika równa adresowi tego symbolu atomowego).



rys.6.3. Budowa pojedyńczego elementu listy.

6.3. Realizacja programów. Makroinstrukcje

W omawianym systemie użytkownik ma do dyspozycji dwa sposoby wykonywania programów: interpretacje oraz wykonywanie programów przetłumaczonych kompilatorem. W tym drugim wypadku wyrażenie tłumaczone jest najpierw na ciąg makroinstrukcji, a następnie makroinstrukcje te są realizowane w fazie właściwego wykonania programu. Program tak skompilowany jest bardzo zwięzły, a jego realizacja trwa niewiele dłużej niż program zakodowanego wzrostu w języku wewnętrznym. Wykonanie skompilowanego programu odbywa się na drodze interpretacji makroinstrukcji przez program zwany realizatorem i będący częścią monitora systemu. Traktuje on makroinstrukcję jako wywołanie skojarzonego z nią podprogramu (w realizacji wewnętrznej makroinstrukcja jest wprost adresem tego podprogramu), wykonuje ten podprogram i wybiera następną makroinstrukcję, która ma podlegać interpretacji (chyba, że podprogram oddał sterowanie głównemu programowi sterującemu monitora - wtedy realizator oczekuje na uaktywnienie).

Zadaniem podprogramów makroinstrukcji jest dokonywanie różnych działań na stosach, przy ewentualnym wykorzystaniu w tych działaniach funkcji systemowych. Istnieją także makroinstrukcje sterujące działaniem realizatora (skoki i skoki warunkowe).

Można powiedzieć, że realizacja programów następuje w specjalnej "maszynie stosowej" posiadającej swoją listę rozkazów (makroinstrukcje), której działanie jest symulowane programowo.

Sprawą ogromnej wagi jest sposób reprezentowania makroinstrukcji w pamięci. Ma on wpływ zarówno na liczbę komórek zajętych przez program, jak i na szybkość jego realizacji.

Przyjęto następujące rozwiązanie:

- makroinstrukcja zajmuje jedną lub dwie kolejne komórki 16 bitowe,
- pierwsze słowo makroinstrukcji jest interpretowane zawsze jako adres programu, który należy wykonać (najczęściej jest to podprogram makroinstrukcji) i który zakodowano w języku wewnętrznym,
- jeżeli makroinstrukcja zajmuje dwie komórki, to druga z nich zawiera argument.

6.4. Odzyskiwanie wolnej pamięci

Integralną częścią systemu jest program automatycznego odzyskiwania nieużytków, zwany programem zbierania śmieci. Jest on wywoływany automatycznie z chwilą całkowitego zapełnienia obszaru danych w celu usunięcia z pamięci wszystkich liczb oraz struktur listowych nieużytecznych z punktu widzenia systemu i dalszego wykonywania programów. Uzyskana w ten sposób wolna pamięć udostępniona zostaje do dalszego wykorzystania.

Odzyskanie wolnej pamięci składa się z faz:

- markowania
- zgęszczania
- uaktualnienia

W pierwszej fazie następuje zaznaczenie słów zawierających informację użyteczną w celu ich ochrony przed zniszczeniem podczas dalszej pracy systemu. W tym celu używa się specjalnych tablic binarnych. Markowanie polega na zapisywaniu jedynki w odpowiednim miejscu tablicy, odpowiadającemu adresowi danej. Tablice te umieszczone są na początku każdej strony zawierającej liczby lub listy.

Paginacja pamięci i dynamiczny przydział stron różnym typom danych wymagają przeprowadzenia operacji zgęszczania, podczas której zmieniają się adresy niektórych aktywnych danych. Konieczne staje się więc uaktualnianie zawartości komórek. Zgęszczanie i uaktualnianie rozwiązano w sposób zbliżony do algorytmu Harta i Evansa.
Po operacji zgęszczania uzyskuje się wolne strony pamięci. Jeśli w chwili wywołania procesu zbierania śmieci wciśnięty był ostatni (piętnasty) klawisz pulpitu technicznego maszyny, działanie programu będzie sygnalizowane wyprowadzeniem na monitorze napisu: GC - w chwili rozpoczęcia i END - w chwili zakończenia działania.

Wykaz sygnalizacji błędów

Nr	Znaczenie	Wyprowadza się
1	RATOM- Symbol atomowy ma więcej niż 64 znaki	-
2	RATOM- błąd leksykalny. Operator zrezygnował z poprawy	-
3	Wartość drugiego argumentu funkcji PRINCH jest błędna	błędna wartość argumentu
4	pierwszy element listy będącej wartością argumentu funkcji FORMAT nie jest liczbą całkowitą	błędny element
5	drugi element listy będącej wartością argumentu funkcji FORMAT nie jest liczbą całkowitą	błędny element
6	Wartość pierwszego argumentu funkcji PRINCH nie jest symbolem atomowym	błędna wartość argumentu
7	PRIN1 błędna wartość argumentu	wartość argumentu
8	Wartość argumentu funkcji OUTBUF nie jest liczbą całkowitą	błędna wartość argumentu
10	Brak pełnego interpretera w pamięci	wyrażenie do zinterpretowania
11	W APPLY - nieokreślony symbol atomowy jako funkcja	nieokreślony symbol atomowy
12	W EVAL - forma jest niekoreślonym symbolem atomowym	nieokreślony symbol atomowy
13	W EVAL - forma rozpoczyna się nieokreślonym symbolem atomowym	nieokreślony symbol atomowy
14	W COND - żaden z warunków nie ma wartości różnej od NIL	-
15	W PAIR - lista wartości argumentów jest dłuższa niż lista zmiennych	dotąd utworzona asocjacje

16	W PAIR - odwrotnie niż w 15	dotąd utworzone asocjacje
17	W APPLY lub EVAL - program funkcji wywołany z błędną liczbą argumentów	funkcja
18	W PROG - niezdefiniowana etykieta	etykieta
19	Wykonanie operatora SET (lub SETQ) nie jest możliwe bo zmienna nie występuje na liście asocjacji	zmienna
20	Wykonanie operatora SET w programie przetłumaczonym nie jest możliwe, bo zmienna nie występuje na liście asocjacji	zmienna
21	Wezwanie funkcji o nieznanej liczbie argumentów	wyrażenie funkcyjne
22	niepoprawne wezwanie funkcji o dowolnej liczbie argumentów	symbol atomowy funkcji
27	wezwanie funkcji określonej błędną definicją	wyrażenie funkcyjne
28	READ - brak strzałki → w parze form wyrażenia warunkowego	ostatnia jednostka leksykalna
29	READ - brak średnika ; w ciągu par form wyrażenia warunkowego lub nawiasu] ograniczającego ten ciąg	ostatnia jednostka leksykalna
30	READ - błędna konstrukcja wyrażenia symbolicznego	ostatnia jednostka leksykalna
31	READ - brak nawiasu zamykającego) w wyrażeniu symbolicznym	ostatnia jednostka leksykalna
32	READ - brak nawiasu [w wyrażeniu LAMBDA	ostatnia jednostka leksykalna
33	READ - brak nawiasu [w wyrażeniu LABEL	ostatnia jednostka leksykalna
34	READ - błędna konstrukcja funkcji	ostatnia jednostka leksykalna
35	READ - brak nawiasu] zamykającego wyrażenie LAMBDA	ostatnia jednostka leksykalna

36	READ - pierwszy argument w wyrażeniu LABEL nie jest atomem	ostatnia jednostka leksykalna
37	READ - brak średnika ; oddzielającego argumenty wyrażenia LABEL	ostatnia jednostka leksykalna
38	READ - brak nawiasu] zamykającego wyrażenie LABEL	ostatnia jednostka leksykalna
39	READ - brak nawiasu [ograniczającego ciąg zmiennych w wyrażeniu LAMBDA lub PROG	ostatnia jednostka leksykalna
40	READ - brak średnika ; w wyrażeniu LAMBDA lub PROG	ostatnia jednostka leksykalna
41	READ - zmienna nie jest atomem	ostatnia jednostka leksykalna
42	READ - brak średnika ; oddzielającego w ciągu zmiennych lub nawiasu] ograniczającego ten ciąg	ostatnia jednostka leksykalna
43	READ - brak nawiasu [ograniczającego ciąg argumentów	ostatnia jednostka leksykalna
44	READ - brak nawiasu [po operatorze QUOTE	ostatnia jednostka leksykalna
45	READ - brak nawiasu [ograniczającego argument operatora QUOTE	ostatnia jednostka leksykalna
46	READ - brak nawiasu [w wyrażeniu PROG	ostatnia jednostka leksykalna
47	READ - błędna konstrukcja formy	ostatnia jednostka leksykalna
48	READ - brak średnika ; w ciągu argumentów lub nawiasu] ograniczającego ten ciąg	ostatnia jednostka leksykalna
49	READ - brak średnika ; w ciągu form lub nawiasu] ograniczającego ten ciąg	ostatnia jednostka leksykalna
50	CAR - błędna wartość argumentu	wartość argumentu
51	CDR - błędna wartość argumentu	wartość argumentu
52	RPLACA - błędna wartość pierwszego argumentu	wartość pierwszego argumentu

53	RPLACD - błędna wartość pierwszego argumentu	wartość pierwszego argumentu
54	CSET - błędna wartość pierwszego argumentu	wartość pierwszego argumentu
55	INPUT - błędna wartość argumentu	wartość argumentu
56	OUTPUT - błędna wartość argumentu	wartość argumentu
60	PLUS - błędna wartość argumentu	wartość argumentu
61	TIMES - błędna wartość argumentu	wartość argumentu
62	MIN - błędna wartość argumentu	wartość argumentu
63	MAX - błędna wartość argumentu	wartość argumentu
64	MAX, MIN, PLUS, TIMES - liczba argumentów <2	-
65	ABS - błędna wartość argumentu	wartość argumentu
66	ADD1 - błędna wartość argumentu	wartość argumentu
67	SUB1 - błędna wartość argumentu	wartość argumentu
68	SIN - błędna wartość argumentu	wartość argumentu
69	COS - błędna wartość argumentu	wartość argumentu
70	ARCT - błędna wartość argumentu	wartość argumentu
71	EXPT - błędna wartość argumentu	wartość argumentu
72	LESSP - błędna wartość argumentu	wartość argumentu
73	GREATERP - błędna wartość argumentu	wartość argumentu
74	MINUS - błędna wartość argumentu	wartość argumentu
75	MINUSP - błędna wartość argumentu	wartość argumentu
76	LOGAND, LOGOR, LOGXOR - liczba argumentów <2	-
77	LOGAND - błędna wartość argumentu	wartość argumentu
78	LOGOR - błędna wartość argumentu	wartość argumentu
79	LOGXOR - błędna wartość argumentu	wartość argumentu
80	ZEROP - błędna wartość argumentu	wartość argumentu
81	ONEP - błędna wartość argumentu	wartość argumentu

82	DIFFERENCE - błędna wartość argumentu	wartość argumentu
83	QUOTIENT - błędna wartość argumentu	wartość argumentu
84	REMAINDER - błędna wartość argumentu	wartość argumentu
85	RECIP - błędna wartość argumentu	wartość argumentu
86	LEFTSH - błędna wartość argumentu	wartość argumentu
87	DIVIDE - błędna wartość argumentu	wartość argumentu
88	ENTIER - błędna wartość argumentu	wartość argumentu
90	GET - wartością pierwszego argumentu nie jest symbol atomowy	wartość pierwszego argumentu
91	FLAG - element listy będącej wartością argumentu nie jest symbolem atomowym	element listy
92	REMPROP - wartością pierwszego argumentu nie jest symbol atomowy	wartość pierwszego argumentu
93	DEFLIST - element definiowany nie jest symbolem atomowym	element definiowany
94	REMFLAG - element listy będącej wartością argumentu nie jest symbolem atomowym	element listy
95	TRACE, UNTRACE - element listy będącej wartością argumentu nie jest symbolem atomowym	element listy
96	LENGTH - błędna wartosć argumentu	wartość argumentu
100	COMPILE - niezdefiniowana etykieta w PROG	etykieta
101	COMPILE - element listy będącej wartością argumentu nie jest symbolem atomowym	element listy
102	COMPILE - symbol atomowy nie posiada cechy EXPR lub FEXPR	symbol atomowy
103	COMPILE - zła składnia funkcji	wyrażenie funkcyjne
104	COMPILE - nieokreślona zmienna	zmienna
105	COMPILE - pierwszy argument CSETQ nie jest atomem	argument
106	COMPILE - pierwszy argument SETQ nieokreślony	argument

107	COMPILE - pusta lista par za COND	NIL
108	COMPILE - błędna liczba argumentów funkcji	funkcja
109	przepełnienie notatnika programu zbierania śmieci	-
110	proces zbierania śmieci nie zwolnił ani jednej strony w systemie	-

Literatura

1. Berkeley E.C., Bobrow D.G., The Programming Language LISP: Its Operation and Application, The MIT Press, Cambridge, Massachusetts 1966.

2. McCarthy J., Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part One, Commun.ACM, vol.3 no.4, April 1960, pp. 184–195.

3. McCarthy J., Abrahams P.W., Edwards D.J., Hart T.P., Levin M.I., LISP 1.5 Programmers Manual, MIT Press, Cambridge, Mass. 1962.

4. Maurer W.D., - The Programmers Introduction to LISP. McDonnald American Elsvier Inc. London, N.York 1972.

5. Turski W.M., Podstawy użytkowania maszyn cyfrowych, PWN, Warszawa 1968.

6. Weissman C., LISP 1.5 Primmer, Dickenson Publ. Comp. Inc., Belmont, Calif. 1967.

Indeks

Symbol atomowy	strony instrukcji
ABS	12, 56
ADD1	12, 57
ALIST	120
AND	29, 57
APPEND	30, 58
APPLY	32, 58, 108
APVAL	23, 111, 122
ARCT	12, 59
ATOM	11, 59
CAR	10,60
CDR	10,60
COMMON	23, 24, 28, 61, 119
COMPILE	32, 40, 61
COND	22, 29, 62, 111
CONS	10, 13, 63
COS	12, 63
CSET	26, 64
CSETQ	26, 64
DEFINE	26, 65
DEFLIST	26, 65
DIFFERENCE	12, 66
DIVIDE	12, 66
ENTIER	12, 67
EQ	11, 67
EQUAL	11, 68
ERROR	68
EVAL	32, 69, 110

EVALQUOTE 31, 41, 69, 105, 107

EVCON 32, 70, 111

EVLIST 32, 70, 115

EXCISE 28, 40, 71

EXPT 12, 72

EXPR 23,32,107,109,115,118

FLAG 28,61,72

FEXPR 23,32,107,109,118

FLOATP 12, 73

FORMAT 24,42,51,73

FSUBR 23,27,107,109,118,122

FUNARG 110, 111

FUNCTION 21,73,111

GENSYM 28, 74

GET 27,74

GO 18,114

GREATERP 12,75

INPUT 24,37,42,55,75

INTEGERP 12, 76

LABEL 17, 21, 109

LAMBDA 16,21,109

LEFTSH 12,76

LENGTH 19,30,77

LESSP 12,77

LIST 30,78

LOGAND 13,78

LOGOR 13,79

LOGXOR 13,79

MAPLIST 30,80

MAX 12,81

MEMBER 30,82
MIN 12,82
MINUS 12,83
MINUSP 12,83
NCONC 32,84
NIL 8,11
NOT 29,84
NULL 85
NUMBERP 11,85
OCTALP 12,85
ONEP 12,86
OR 29,86
OUTBUF 24,42,51,87
OUTPUT 24,37,42,55,87
PAIR 32,88
PLUS 12,89
PNAME 23,122
PRINCH 25,42,50,90
PRINT 25,49,90
PRIN1 25,42,50,89
PROG 17,19,22,91,112
QUOTE 14,21,91,111
QUOTIENT 12,92
RATOM 25,42,49,92
READ 19,24,42,48,93
READCH 42,49,93
RECIP 12,93
REMAINDER 12,94
REMFLAG 28,94
REMPROP 27,95

RETURN	18,113
RPLACA	13,95
RPLACD	13,96
SEARCH	30,96
SELECT	29,97
SET	98,113
SETQ	22,98,113
SIN	12,99
SPECIAL	23,28,99,119
START	24,38,42,55,100
SUB1	12,100
SUBR	23,27,41,107,109,118
T	11
TAPE	47
TIMES	12,101
TRACE	23,28,101,116,120,123
UNCOMMON	28,102
UNION	30,102
UNSPECIAL	28,103
UNTRACE	28,103
ZEROP	12,104