Lisp – podstawy

Nazwa *Lisp*, podobnie jak *Prolog*, zawiera w sobie informacje o zasadniczym przeznaczeniu języka. W tym przypadku nazwa pochodzi od angielskich wyrazów **lis***t* **p***rocessing* co na język polski możemy przetłumaczyć jako *przetwarzanie list*. Lista miała być narzędziem pozwalającym na efektywne przetwarzanie danych symbolicznych.

Lisp jest drugim z kolei pod względem wieku językiem programowania wysokiego poziomu pozostającym w użyciu (starszy jest tylko Fortran). Choć pierwsza oficjalna wzmianka o Lispie pochodzi z 4 marca 1959 roku, to zwykle narodziny tego języka umiejscawia sie w roku 1958. W roku 1960 John McCarthy opublikował swój projekt w Communications of the ACM, w artykule pod tytułem *Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I* (*Rekursywne funkcje wyrażeń symbolicznych i ich maszynowe obliczanie, część I* ; części II nigdy nie opublikowano), w którym pokazał, ze za pomocą kilku operatorów i notacji dla funkcji można otrzymać kompletny w sensie Turinga język programowania (ang. *Turing-complete language*).

Współcześnie najbardziej popularną wersją Lispa jest Common Lisp. Co ciekawe, Common Lisp nie jest konkretna implementacja, ale specyfikacja stworzona przezANSI mającą na celu ujednolicenie powstałych do tego czasu implementacji Lispa. Obecnie dostępnych jest kilka implementacji Common Lisp, zarówno zamkniętych, jak i dostępnych jako FOSS (ang. *free and open source software*).

“Anyone could learn Lisp in one day, except that if they already knew Fortran, it would take three days.”

Marvin Minsky[1]

[1] http://lispers.org/

Charakterystyka języka

O Lispie mówimy, ze jest **językiem funkcjonalnym** lub **językiem programowania funkcyjnego**. Programowanie funkcyjne (lub programowanie funkcjonalne) to paradygmat programowania będący odmiana **programowania deklaratywnego**, w którym podstawowym elementem języka jest funkcja, a nacisk kładzie sie na wartościowanie (często rekurencyjnych) funkcji, a nie na wykonywanie poleceń. Podstawa teoretyczna programowania funkcyjnego jest opracowany w latach 30. XX wieku przez Alonzo Churcha *rachunek lambda*, a dokładnie rachunek
lambda z typami.
Lisp jest językiem, którego składnia opiera sie na wyrażeniach (ang. *expression-oriented language*).
W przeciwieństwie do większości innych języków, nie ma w nim podziału na wyrażenia (ang. *expressions*) i instrukcje (ang. *statements*). Wynikiem ewaluacji (wartościowania) wyrażeń jest wartość (lub lista wartości), która może być użyta jako argument dla innego wyrażenia. Pierwotnie John McCarthy wprowadził dwa typy wyrażeń: S-wyrażenia (ang. *S-expressions*), czyli wyrażenia symboliczne (ang. *Symbolic Expressions*) nazywane także sexpami (ang. *sexps*). Odzwierciedlały one wewnętrzna reprezentacje kodu i danych. M-wyrażenia (ang. *M-expressions*), czyli meta wyrażenia (ang. *Meta Expressions*). Opisywały one funkcje S-wyrażeń.
I choć w założeniach to M-wyrażenia miały tworzyć składnie Lispa, to okazało sie, ze S-wyrażenia zdobyły bardzo duża popularność.
Nieodłącznym elementem wyrażeń w Lispie są nawiasy. To dzięki nim nazwa Lisp rozwijana jest też jako *Lots of Irritating Superfluous Parentheses* (wiele irytujżcych zbytecznych nawiasów) czy też *Lost In Stupid Parentheses* (zagubiony w głupich nawiasach). Jednak nawet przeciwnicy Lispa musza przyznać, iż składnia oparta na S-wyrażeniach leży u podstaw jego możliwosci – jest ona niezwykle regularna, co znakomicie ułatwia jej przetwarzanie przez komputer. Podstawowym elementem składni Lispa jest lista. Zapisywana jest ona jako elementy rozdzielone białymi znakami i otoczone wspomnianymi juz nawiasami.

Na przykład,

(1 2 foo)

to lista, której elementami są trzy atomy: (domyślnie typowane) wartości 1, 2, i foo. Wyrażenia zapisywane są jako listy z wykorzystaniem **notacji polskiej**. Co ciekawe lista jako struktura danych służy w Lispie także do zapisu kodu źródłowego.

Elementy języka

Na początku wyjaśnijmy podstawowe pojęcia: funkcja i dane. Terminem dana określa się informacje takie jak liczby, słowa lub listy rzeczy. Funkcje można porównać do pudełka, przez które wychodzą dane.

Symbol – to każdy łańcuch znaków, cyfr i niektórych znaków specjalnych nie będący liczbą, np.: x, y, z, xyz, sdf-df, g7y7y, znak, heigh, . Zazwyczaj nazwy symboli biorą się z angielskich nazw (w naszym wypadku także polskich).

Zachowaj liczbę 4 jako wartość symbolu x:

```
(setq x 4)
```

Pobierz wartość symbolu x:

```
x
```

zwracane jest: 4

Użyj wartości symbolu jako argumentu funkcji +:

```
(+ x 7)
```

zwracane jest: 11

spróbuj pobrać wartość symbolu, któremu nie przypisano żadnej wartości:

```
m
```

wystąpi błąd: variable M has n value

Symbole specjalne: T i NIL

Lisp posiada dwa symbole posiadające przypisane im na stałe wartość. Są to: T oznaczający „prawda” , „tak” oraz NIL - „fałsz”, „nie”, „pusty”

Test symboli T i NIL:

```
(if  T 3 9)
```

zwracane jest: 3

```
(if  NIL 3 9)
```

zwracane jest: 9

(if 1 3 9)

zwracane jest: 3

Należy tu dodać pewne wyjaśnienie. Nil zawsze oznacza fałsz, czyli wszystko inne zawsze będzie oznaczało prawdę. Właściwie T używamy dla pewnej prostoty i przejrzystości.

Słowo kluczowe – dowolny symbol, którego nazwa rozpoczyna się od „:”, np.:

Słowa kluczowe

```
:to-jest-slowo-kluczowe
```

zwracane jest: :TO-JEST-SLOWO-KLUCZOWE

Liczby – to łańcuch cyfr od “0” do “9”, czasami poprzedzonych znakami “+” lub “-“. Wyróżniamy liczby: całkowite, dziesiętne (ma kropkę dziesiętną i może być zapisana w notacji naukowej), wymierna (dwie liczby całkowite przedzielone znakiem „/”), zespolone (są zapisywane jako #c(r i) -gdzie r jest częścią rzeczywistą, a i jest częścią urojoną), np:

Kilka liczb w LISPie

```
234
-62
+478
3.1415
3.45e-3
#c(6.13e-3 4.123)
```

Standardowe funkcje arytmetyczne są wciąż dostępne: +, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt, itd. Wszystkie one przyjmują dowolny rodzaj argumentu. +, -, *, i / zwracają liczbę zgodnie z następującymi zasadami: liczba całkowita plus wymierna daje wymierną, wymierna plus rzeczywistą daje rzeczywistą, a rzeczywista plus zespolona daje zespoloną.

podstawowe funkcje arytmetyczne

```
(+ 4 3/5)
```

zwracane jest: 23/5

```
(exp 1)
```

zwracane jest: 2.7182817

```
(exp 3)
```

zwracane jest: 20.085537

(expt 3 4.2)

zwracane jest: 100.90418

Jedynym ograniczeniem wielkości liczby całkowitej jest pamięć komputera. Nie mniej jednak operowanie na dużych liczbach bardzo spowalnia prace komputera.

EVAL - serce LISPa (notacja polska)

Funkcja EVAL jest sercem Lisp. Zadaniem EVAL jest ocenienie wyrażenia aby obliczyć jego skutek. Większość wyrażeń składa się z funkcji mający zbiór na wejściu. Jeśli dajemy do EVAL wyrażenie (+ 3 2), na przykład, to odwoła się ono do funkcji wewnętrznej +, z 2 i 3 na wejściu i zwróci 5. Dlatego mówimy że wyrażenie (+ 3 2) ma wartość 5.

Na przyszłość użyjemy tylko strzałki. A więc nasz przykład zapiszemy tak:

(+ 2 3) => 5

Oto więcej przykładów wyrażeń w notacji EVAL:

(+ 1 6) => 7
(* 3 (+ 1 6)) => 21
(/ (* 2 11) (+ 1 6)) => 22/7

Set Quantity – setq i setf

We wczesnych dialektach Lispa istniała tylko funkcja SETQ, zaś uogólnione zmienne były niedostępne. Obecnie funkcja SETQ jest nadal używana, jednak w nowszych wersjach programiści używają raczej podobnego znaczeniowo makra SETF, które pozwala na przechowywanie w zarówno zwykłych zmiennych takich jak X, jak i w uogólnionych zmiennych takich jak (SECOND X).

(setq x ’(wartosc)) => (WARTOSC)

Funkcja SETF, podobnie jak SETQ, wywodzi się z najwcześniejszych odmian Lispa. Jednak jej znaczenie zmieniło się. W Common Lisp SETF zachowuje wartość w odpowiedniej komórce, a dokładniej ustawia symbol jako nazwę zmiennej globalnej, nawet jeżeli istnieje już zmienna lokalna o tej samej nazwie. Nazwy zmiennych nie rozróżniają wielkości liter.

zmienna globalna KACZKA

(setf kaczka ’donald)

zmienna lokalna KACZKA

```
(defun test1 (kaczka)
  (list kaczka
    (symbol-value ' kaczka)
  )
)

(test1 'kwak)                    => (kwak donald)
```

zmiana zmiennej globalnej KACZKA

```
(defun test2 (kaczka)
  (set ' kaczka 'daffy)
  (list kaczka
    (symbol-value ' kaczka)
  )
)

(test2 ' kwak)          => (kwak daffy)
kaczka                        => daffy
```

Specjalna forma SETF używa swojego pierwszego argumentu do zdefiniowania miejsca w pamięci, analizuje drugi argument i zapisuje wynik w wynikowej pozycji pamięci.

```
(setq tablica (make-array 3)) =>    #(NIL NIL NIL)
(aref tablica 1)              =>    NIL
(setf (aref tablica 1) 3)     =>    3
 tablica              =>    #(NIL 3 NIL)
(aref tablica 1)              =>    3
(defstruct foo bar)           =>    FOO
(setq tablica (make-foo))     =>    #s(FOO :BAR NIL)
(foo-bar tablica)             =>    NIL
tablica              =>    #s(FOO :BAR NIL)
```

SETF jest również jedynym sposobem na ustawianie pól struktury lub tablicy.

Definiowanie Funkcji

Oto kilka przykładów funkcji:

Funkcja pobiera dowolną liczbę argumentów:

```
(+ 3 4 5 6)                  =>    18
(+ (+ 3 4) (+ (+ 4 5) 6))    =>    22
```

Definiowanie funkcji:

```
(defun foo (x y) (+ x y 5))     =>      FOO
```

Wywołanie stworzonej funkcji:

```
(foo 5 0)                       =>      10
```

Funkcja rekursywna:

```
(defun silnia (x)
  (if (> x 0)
    (* x (silnia (- x 1)))
    1
  )
)                               =>      SILNIA

          (silnia 5)            =>      120
```

Funkcja o wielu instrukcjach - zwróci wartość, zwracaną przez jej ostatnią instrukcję

```
(defun bar (x)
  (setq x (* x 3))
  (setq x (/ x 2))
  (+ x 4)
)                               =>      BAR

          (bar 6)               =>      13
```

W notacji EVAL używamy list, aby zdefiniować funkcje i odnosimy się do argumentów funkcji przez nadane im nazwy.

Funkcja AVERAGE jest zdefiniowana w notacji EVAL w ten sposób:

```
(defun average (x y)
  (/ (+ x y) 2.0)
)
```

Nazwa DEFUN pochodzi od define function. Jak wskazuje nazwa DEFUN używa się, by zdefiniować inne funkcje. Pierwszy wejście do DEFUN to nazwa funkcji zdefiniowanej. Drugi wejście jest listą argumentów, czyli nazwy, których funkcja użyje, by odnieść się do jego argumentów. Pozostała część wejścia do DEFUN definiuje ciało funkcji, a więc to, co jest wewnątrz pudełka.

```
  (defun square (n) (* n n))
```

Nazwa funkcji to SQUARE. Jego listą argumentów jest (N), czyli SQUARE przyjmuje jeden argument, do którego to odnosi się wartość N. Ciało funkcji jest wyrażeniem (* N N).

Pewne funkcje wymagają ustalonej liczby argumentów na wejściu, np: ODDP, który przyjmuje dokładnie jedno wejście i EQUAL, który bierze dokładnie dwa. Ale istnieje wiele funkcji przyjmujących zmienną liczbę wejść. Przykładowo - funkcje arytmetyczne +, -, * i /.

```
(* 2 3 5)        =>      30
```

Aby pomnożyć trzy liczby, funkcja * mnoży najpierw pierwsze dwie liczby, a następnie mnoży do skutku kolejne przez liczby na podstawie wyniku poprzedniego działania.

Drukowanie

Niektóre funkcje mogą wysyłać wyniki do standardowego wyjścia. Najprostszą funkcją jest PRINT, która drukuje swój argument i zwraca go.

```
(print 3)       3
        =>      3
```

Pierwsze 3 zostało wydrukowane, drugie zwrócone.

Jeśli oczekujesz bardziej złożonego wyjścia, musisz użyć formatu.

```
(format t "An atom: ~S~%and a list: ~S~%and
        an integer: ~D~%"nil (list 5) 6)

An atom: NIL
and a list: (5)
and an integer: 6
```

Pierwszy argument instrukcji format to ‘T’, ‘NIL’, lub strumień´. ‘T’ wskazuje na wyjście na terminal. ‘Nil’ oznacza, że niczego nie można drukować´, a należy zamiast tego powrócić do łańcucha zawierającego wyjście. Strumienie są ogólnymi miejscami przepływu wyjścia: wskazują na pliki, terminale lub inne programy.

Kolejnym argumentem jest matryca formatująca, która jest łańcuchem opcjonalnie zawierającym dyrektywy formatujące. Wszystkie pozostałe argumenty mogą być używane przez dyrektywy formatujące. LISP zamieni dyrektywy na odpowiednie znaki, w oparciu argumenty, do których (dyrektywy) się one odnoszą. Następnie łańcuch zostanie wydrukowany. Format zawsze zwraca ‘NIL’, chyba że jego pierwszym argumentem jest ‘NIL’-wtedy nic nie drukuje i zwraca łańcuch.

W powyższym przykładzie istnieją trzy rożne dyrektywy: ~S, ~D i ~%. Pierwsza przyjmuje dowolny obiekt LISP i jest zamieniana drukowalną reprezentacją tego obiektu (taką samą, jak

produkowana przez PRINT). Druga przyjmuje tylko liczby całkowite. Ostatnia nie odnosi się do argumentu. Jest zawsze zamieniana na powrót karetki.

Predykaty

Predykat jest funkcją zwracającą odpowiedź. Predykat zwraca symbol T kiedy ma na myśli tak i symbol NIL, kiedy odpowiedź znaczy nie. Predykat z poniższego przykładu orzeka, czy na jego wejściu jest numer.

```
(NUMBERP 2)              =>    T
(NUMBERP ”KOT” )                =>    NIL
```

Oto inne przykłady predykatów: < zwraca T, jeśli jego pierwsze wejście jest mniej niż jego drugie wejście i analogicznie > zwraca T, jeśli jego pierwsze wejście jest większe niż jego drugie wejście. (To jest też pierwszy wyjątek od konwencji, że nazwy predykatów kończą się literą "'P".)

```
(< 2 3)        =>    T
(> 2 3)        =>    NIL
```

EQUAL to predykat porównujący dwie rzeczy, by zobaczyć, czy są one takie same. EQUAL zwraca T, jeśli jego dwa wejścia są równe; w przeciwnym wypadku zwraca NIL. Dialekt języka programowania Lisp też zawiera orzeczenia nazwane EQ, EQL i EQUALP, których zachowanie jest nieznacznie różne niż EQUAL; różnice nie będą dotyczyły nas tutaj. Dla początkujących, wystarczy znajomość EQUAL.

```
(EQUAL ”KOT” ”MYSZ”)  =>    NIL
(EQUAL ”KOT” ”KOT”)   =>    T
```

Predykat LISTP zwraca T, jeśli jego wejście jest listą oraz zwraca NIL dla nie - listy.

```
(LISTP ”kotek”)                          =>    NIL
(LISTP (list ”wlazl” ”kotek” ”na” ”plotek”))    =>    T
```

Funkcje logiczne

NOT

NOT jest " przeciwieństwem" predykatów: zamienia tak z nie i nie z tak. W terminologii Lispa podanie na wejście NOT wartości T zwraca NIL, zaś podanie wartości NIL zwraca T. Ciekawą rzeczą jest fakt, iż możemy przywiązać NOT do jakiegoś innego predykatu, by

wyprowadzić jego przeciwność; na przykład , możemy zrobić predykat „nierówny” poprzez połączenie NOT i EQUAL, albo predykat „niezerowy” - NOT i ZEROP.

```
(not t)  =>     NIL
(not nil)       =>      T
```

Błędy
Chociaż nasz system funkcji jest bardzo prosty, już teraz możemy zrobić w nim kilka typów błędów. Możemy na przykład podać na wejście funkcji niewłaściwy typ danych. Na przykład, funkcja + może dodawać tylko liczby i nie przyjmuje symboli.

```
(+ “ania” 3)    =>      Error! Wrong type input.
```

Inny rodzaj błędu to podanie na wejście funkcji zbyt mało albo zbyt wiele argumentów.

```
(equal 2)       =>      Error! Too few inputs.
(oddp 4 7)      =>      Error! Too many inputs.
```

W końcu, błąd może zdarzyć się, gdy funkcja nie może zrobić tego, o co jest poproszona. To się może zdarzyć np. kiedy próbujemy podzielić liczbę przez zero.

AND, OR

Często potrzeba budować złożone predykaty z tych prostszych. Dzięki makrom AND i OR jest to możliwe. Popatrzmy na przykład. Pokazuje on jak zbudować funkcje sprawdzającą nieparzystość liczb z przedziału od 0 do 100 przy pomocy AND.

```
(defun small-positive-oddp (x)
  (and (< x 100)
    (> x 0)
    (oddp x)
  )
)
```

AND i OR w Lispie nieznacznie różnią się od podobnych funkcji logicznych.

Regułą dla AND jest: oceń po jednej klauzuli, jeśli zwrócą NIL, zatrzymaj i zwróć NIL; inaczej przejdź do następnej - jeśli wszystkie klauzule zwracają wartość różną od NIL, zwraca wartość ostatniej klauzuli.

```
(and 1 2 3 4 5)                 =>      5
(and nil t t)                   =>      NIL
(and ’ania NIL ’pawel)                  =>      NIL
(and ’ilona ’asia ’krzysiek)    =>      krzysiek
```

Regułą dla oceniania OR jest: oceń po jednej klauzuli, jeśli zwrócą coś innego niż NIL, zatrzymaj

i zwróć tę wartość, inaczej pójdź dalej do następnej klauzuli - zwróć NIL jeśli żadna klauzula już nie została.
10.6.

(or nil t t)	=>	T
(or 'ilona 'asia 'krzysiek)	=>	ilona
(or nil 'wojtek 'ania)	=>	wojtek

AND i OR posiadają wartościową własność kończenia działania funkcji po otrzymaniu odpowiedniej wartości nawet w połowie wykonywania się funkcji. Pozwala to na wyłapanie błędów, które inaczej zdarzyłyby się. Na przykład w przypadku predykatu POSNUMP.

```
(defun posnump (x)
  (and (numberp x) (plusp x))
)
```

POSNUMP zwraca T, jeśli jego wejście jest liczbą i jest pozytywne. Wbudowany predykat PLUSP może zostać użyty, by powiedzieć, czy liczba jest pozytywna, ale, jeśli PLUSP jest użyty na czymś innym niż liczba, to zasygnalizuje błąd. Dlatego ważne jest, by upewnić się, że wejście do POSNUMP jest liczbą przed odwoływaniem PLUSP. Jeśli wejście nie jest liczbą, nie wolno wywołać PLUSP.

Instrukcje warunkowe – IF, WHEN, UNLESS, CASE, COND

IF

LISP wyposażony jest również w specjalne formy dla wyrażeń warunkowych. IF jest najprostszą z nich. Specjalna funkcja IF bierze trzy argumenty: próba, prawdziwa część i fałszywa część. Jeśli próba jest prawdziwa, IF zwraca wartość prawdziwej części. Jeśli próba jest fałszywa, to opuszcza prawdziwą część i zamiast zwraca wartość fałszywej części.

(if t 2 8)	=>	2
(if nil 2 8)	=>	8
(if 9 2 8)	=>	2
(if (oddp 7) 'odd 'even)	=>	odd
(if (oddp 6) 'odd 'even)	=>	even

(if t 'tekst-jest-prawdziwy 'tekst-jest-falszywy)

=> tekst-jest-prawdziwy

(if nil ’tekst-jest-prawdziwy ’tekst-jest-falszywy)

=> tekst-jest-falszywy

(if (symbolp ’slowo) (* 3 4) (+ 3 4)) => 12

(if (symbolp 6) (* 3 4) (+ 3 4)) => 7

W przypadku gdy potrzeba wykonać więcej niż jedną instrukcję w klauzuli then lub else IF’a, to można użyć specjalnej formy PROGN. PROGN wykonuje każdą instrukcję swojego ciała i zwraca ostatnią wartość.

WHEN i UNLESS

WHEN i UNLESS w przeciwieństwie do IF, zezwalają na dowolną liczbę instrukcji w swoich ciałach. (Np. (when x a b c) jest równoważne (if x (progn a b c)).

Instrukcja IF, której brak klauzuli then lub else, może być zapisana przy użyciu specjalnej formy UNLESS.

(when t 5) => 5

(when nil 5) => NIL

(unless t 5) => NIL

(unless nil 5) => 5

CASE

Instrukcja CASE w LISP jest podobna do instrukcji switch w C:

(setq x ’b) => B

(case x

(a 5)

((d e) 7)

((b f) 3)

(otherwise 9)

) => 3

Klauzula otherwise oznacza, że jeśli x nie jest a, b, d, e lub f, instrukcja CASE ma zwrócić 9.

Bardziej złożone warunki można definiując przy użyciu formy specjalnej COND która jest równoważna konstrukcji: if ... else if ...

COND

COND składa się z symbolu ‘COND’, za którym następują klauzule COND, z których każda jest listą. Pierwszy element klauzuli cond jest warunkiem; pozostałe elementy (jeśli istnieją) są akcją. Forma COND szuka pierwszej klauzuli, której warunek jest spełniony; potem wykonuje odpowiednią akcję i zwraca wartość wynikową. Żaden pozostały warunek nie jest już analizowany; nie są też wykonywane inne akcje niż ta, odpowiadająca warunkowi.

Ogólna forma wyrażenia COND wygląda tak:

(cond (test-1 consequent-1)

(test-2 consequent-2)

(test-3 consequent-3)

....

(test-n consequent-n))

```
(setq a 3)              =>      3
```

```
(cond
  ((evenp a) a)          ;jeśli a jest parzyste, zwróć a
  ((> a 7) (/ a 2))              ;inaczej, jeśli a jest > niż 7, zwróć a/2
  ((< a 5) (- a 1))              ;inaczej, jeśli a jest < niż 5, zwróć a-1
  (t 17)                 ;inaczej zwróć 17
)                        =>      2
```

Jeśli w danej klauzuli COND brakuje akcji, COND zwraca wartość, do której został zredukowany warunek:

```
(cond ((+ 3 4)))        =>      7
```

Instrukcje warunkowe to specjalne funkcje decydowania, które wybierają rezultat spośród zbioru wartości bazujących na wyniku jednego lub kilku predykatów. Tryby warunkowe pozwalają funkcji na zmianę zachowania w zależności od rodzaju wejścia. Odkąd możemy pisać funkcje, które robią dowolnie złożone decyzje.

Użyjmy COND do napisania funkcji COMPARE porównującej dwie liczby. Jeśli liczby są równe, COMPARE „powie” ‘liczby-sa-rowne’; jeśli pierwsza liczba będzie mniejsza niż druga, to „powie” ‘pierwsza-liczba-jest-mniejsza’; jeśli pierwszy numer będzie większe niż drugi, to „powie” ‘pierwsza-liczba-jest-wieksza’. Każdy przypadek jest obsłużony przez oddzielną klauzulę COND.

```
(defun compare (x y)
  (cond ((equal x y) 'liczby-sa-rowne)
    ((< x y) 'pierwsza-liczba-jest-mniejsza)
    ((> x y) 'pierwsza-liczba-jest-wieksza)
  )
```

)

Jedna ze standardowych sztuczek używania COND to umieszczenie na końcu COND klauzuli

(T consequent)

Ponieważ T jest zawsze prawdziwe, więc jeśli COND kiedykolwiek dotrze do tej klauzuli, to wykona consequent. Z drugiej strony klauzula ta zostanie osięgnięta tylko wtedy, jeśli zawiodą wszystkie poprzedzające ją klauzule.

```
(defun gdzie-jest (x)
  (cond ((equal x ’paryz) ’francja)
    ((equal x ’londyn) ’anglia)
    ((equal x ’pekin) ’chiny)
    (t ’nieznane)
  )
)
```

Warto zauważyć, że ostatnia klauzula COND zaczyna się z T. Oznacza to, że jeśli żadna z poprzedzających klauzuli nie zostanie wykonana, wykona się ostatnia klauzula i funkcja zwróci NIEZNANE.

```
(gdzie-jest ’londyn )  =>  anglia
(gdzie-jest ’pekin  )  =>  chiny
(gdzie-jest ’parasol)  =>  nieznane
```

Równość obiektów

W LISP zmienne można porównywać na kilka sposobów. Każdy kolejny przedstawiony predykat równości obiektów będzie prawdziwy, jeśli poprzednie predykaty też były prawdziwe dla zadanych zmiennych. Osobno omówiony jest predykat =, służący do porównywania liczb.

Eq

Predykat ten służy do porównywania symboli i obiektów przez adres. Ponieważ każdy symbol może mieć tylko jedną reprezentację w pamięci, to dla identycznie wyglądających symboli predykat ten jest zawsze prawdziwy. Podobnie jeżeli dwie zmienne wskazują ten sam obiekt w pamięci(np. dzięki zastosowaniu instrukcji setq), to predykat ten będzie prawdziwy dla takich obiektów. W przypadku liczba sprawa wygląda inaczej. Liczby nie muszą mieć pojedynczej reprezentacji w różnych implementacjach LISP i z tego względu predykat eq będzie zawodny w przypadku liczb. Również dwie identycznie wyglądające listy, posiadające jednak inny adres w pamięci będę różne dla predykatu eq.

Przykłady:

> (eq 'a 'a) => t

> (eq 'a 'b) => nil

> (eq 1 1) => nil

> (eq 1 1.0) => nil

> (setq a '(1 2 3)) => (1 2 3)

> (setq b a) => (1 2 3)

> (eq a b) => t

> (setq b '(1 2 3)) => (1 2 3)

> (eq a b) => nil

eql

Predykat ten jest stosowany do badania liczb należących do jednego typu, np. całkowitego. Warto zwrócić na niego uwagę, ponieważ jest standardowo stosowany w takich instrukcjach jak np. member. Poza tą własnością nie różni się niczym od predykatu eq.

Przykłady:

> (eql 'a 'a) => t

> (eql 'a 'b) => nil

> (eql 1 1) => t

> (eql 1 1.0) => nil

> (setq a '(1 2 3)) => (1 2 3)

> (setq b a) => (1 2 3)

> (eql a b) => t

> (setq b '(1 2 3)) => (1 2 3)

> (eql a b) => nil

equal

Predykat ten jest dobry do badania identycznie wyglądających obiektów, gdyż w przypadku, gdy dwa obiekty wyglądają identycznie, to wynik tego predykatu będzie prawdziwy. Mówiąc ściślej, jeśli mamy dwie listy zawierające identyczne symbole, lecz posiadające inny adres w pamięci komputera, to wynikiem zastosowania dla nich tego predykatu będzie prawda. Predykat ten jest oczywiście wolniejszy od eq, dlatego do porównywania symboli znacznie lepiej jest używać tego drugiego.

Przykłady:

> (setq a '(1 2 3)) => (1 2 3)

> (setq b a) => (1 2 3)

> (equal a b) => t

> (setq b '(1 2 3)) => (1 2 3)

> (equal a b) => t

equalp

Predykat ten jest jeszcze mniej restrykcyjny niż equal, ponieważ dopuszcza by porównywane łańcuchy miały różne wielkości liter, natomiast gdy łańcuchy mają identyczny wygląd (bez względu na wielkość liter), to predykat ten zwraca wartość prawda.

Przykłady:

> (equal "Ala ma kota" "Ala Ma Kota") => nil

> (equal "Ala ma kota" "Ala Ma Kota") => t

=

Predykat ten służy do porównywania liczb, bez względu na to do jakiego typu należą. Można więc porównywać liczby całkowite z wymiernymi czy też rzeczywistymi. Jeśli ich wartość jest taka sama to predykat ten zwraca wartość t. Nie można go jednak stosować do porównywania żadnych innych obiektów, w szczególności symboli.

Przykład:

> (= 1 2) => nil

> (= 1 1.0) => t

> (= 1/2 0.5) => t

> (= 'a 'a) => nil