LISP - JĘZYK PROGRAMOWANIA

Common Lisp - dialekt języka Lisp

Dariusz Sowiński Kamil Walkowicz

Języki i Systemy Sztucznej Inteligencji

Tarnów, 19 stycznia 2009

Informatyka Stosowana 4 rok
spec.: Informatyka w Sterowaniu i Zarzadzaniu
Instytut Politechniczny Państwowej Wyższej Szkoły Zawodowej
w Tarnowie

Zagadnienia

1. LISP - sam o sobie.
2. Wprowadzenie - podstawy.
3. Przykłady programów w LISP.
4. Inni o LISP.
5. Szybki start z LISP.

Zagadnienia

1. LISP - sam o sobie.
2. Wprowadzenie - podstawy.
3. Przykłady programów w LISP.
4. Inni o LISP.
5. Szybki start z LISP.

Zagadnienia

1. LISP - sam o sobie.
2. Wprowadzenie - podstawy.
3. Przykłady programów w LISP.
4. Inni o LISP.
5. Szybki start z LISP.

Zagadnienia

1. LISP - sam o sobie.
2. Wprowadzenie - podstawy.
3. Przykłady programów w LISP.
4. Inni o LISP.
5. Szybki start z LISP.

Zagadnienia

1. LISP - sam o sobie.
2. Wprowadzenie - podstawy.
3. Przykłady programów w LISP.
4. Inni o LISP.
5. Szybki start z LISP.

LISP - Sam o sobie

LISP jest drugim z kolei pod względem wieku językiem programowania wysokiego poziomu (starszy jest tylko Fortran). Lisp powstał jako wygodna matematyczna notacja dla programów komputerowych,szybko został najchętniej wybieranym językiem do badania i rozwoju sztucznej inteligencji. Ponieważ był jednym z pierwszych języków programowania wysokiego poziomu, wywodzi się z niego wiele technik programistycznych, takich jak struktury drzewiaste, garbage collection, dynamiczne typowanie czy nowe koncepcje w programowaniu obiektowym.

LISP - Sam o sobie (2)

Common Lisp jest najpopularniejszym dialektem Lispa, wieloparadygmatowym językiem programowania ogólnego przeznaczenia, skupiającym się na programowaniu funkcyjnym, pozwalając jednak stosować obiektowość, co daje programiście dużą swobodę.

LISP - rachunek lambda

Rachunek lambda i logika kombinatoryczna powstały w latach trzydziestych dwudziestego wieku. Początkowo miały stanowić alternatywne, wobec teorii mnogości podejście do podstaw matematyki, w którym funkcja (rozumiana intensjonalnie jako definicja) była pojęciem pierwotnym. Ten zamiar się nie powiódł, ale szybko okazało się, że rachunek lambda jest niezwykle użytecznym aparatem w teorii obliczeń. Definicję funkcji obliczalnej sformułowano wcześniej za pomocą rachunku lambda, niż w języku maszyn Turinga. Później, w miarę rozwoju informatyki, język rachunku lambda okazał się niezastąpionym narzędzien w teorii języków programowania.
A wreszcie i w praktyce, gdy pojawiły się języki programowania funkcyjnego.

LISP - rachunek lambda (2)

Rachunek lambda jest szczególnie przydatny przy badaniu algorytmów - każdy algorytm, który da się zapisać za pomocą rachunku lambda da się zaimplementować na maszynie Turinga.

Rachunek lambda:

- rachunek bez typów
- rachunek z typami - stanowi podstawę funkcyjnych języków programowania: haskell, Lisp itp.

Podstawy - lista

- kod lispowych programów jest zbudowany z list co pozwala na wygodne metaprogramowanie. W wyniku tego programy w Lispie mogą manipulować kodem źródłowym jak zwykłą strukturą danych, co umożliwia pisanie makr, pozwalających programiście tworzyć nową składnię lub nawet małe zagnieżdżone w Lispie języki.

np.

> () - lista pusta
> (list 3 8 17 33 2) - lista cyfr
> (list 'sloneczna 'pochmurna 'deszczowa) - lista słów
> (list '(a b (1 2 3) d e (4 5))) - lista mieszana

Podstawy - lista (2)

- Podstawowym budulcem listy jest struktura zwana **cons**. **Cons** sklada sie z dwóch pól: *car i cdr*.
- Kazda komórka listy to jeden **cons** - w polu *car* znajduje sie dana, w polu *cdr* - adres nastepnej komórki cons. NULL(NIL) w polu cdr oznacza koniec listy.
- W Lispie funkcja **cadr** jest skrótem dla ?car cdr?.

np.

```
> (cons 2 8)
> (car (cons 2 8))
> (cdr (cons 2 8))
> (cadr (list 1 2 3 4))) <=> (car (cdr (list 1 2 3 4))))
```

Podstawy - lista (3)

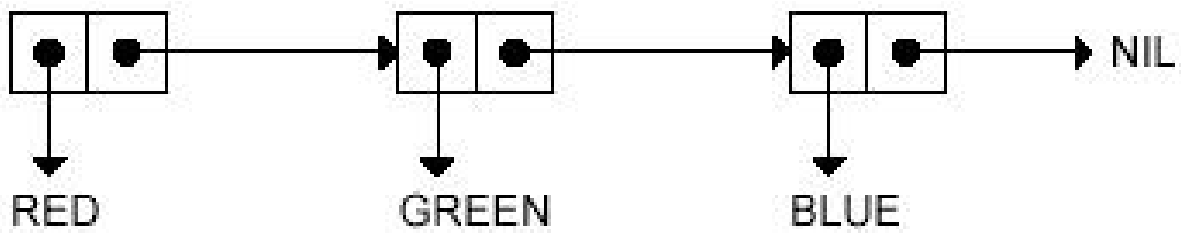
- Lisp dostarcza również pierwotnych funkcji dla wyciągania elementów z listy. Funkcje **FIRST, SECOND, THIRD** zwracają kolejno pierwszy, drugi i trzeci element z listy podanej na ich wejście.
  Funkcja **REST** uzupełnia funkcje FIRST - zwraca listę zawierającego wszystko oprócz pierwszego elementu.

np.

```
> (FIRST '(A B C D))
> (SECOND '(A B C D))
> (THIRD '(A B C D))
> (REST '(A B C D))
```

Podstawy - lista (4)

- Wewnątrz pamięci komputera, listy zorganizowane są jako łańcuchy komórek. Komórki te połączone są razem przez wskazówki. Każda komórka ma dwie wskazówki. Jedna z nich zawsze wskazuje do elementu listy, takie jak RED, zaś kolejna do następnej komórki w łańcuchu.



Podstawy - lista asocjacyjna

- Lista asocjacyjna to lista, która zawiera elementy zwane wpisami. Każdy wpis też jest listą zawierającą klucz i jego wartość. W LISP istnieje instrukcja pozwalająca w łatwy sposób docierać do elementów będących elementami listy asocjacyjnej - **assoc**. Zwraca pierwszy wpis, którego klucz pokrywa się z *nazwaZmiennej*.

np.

```
(assoc 'nazwaZmiennej nazwaListy)
```

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z
- **(FIND 'b '(a x b c))** podobnie jak *MEMBER*, ale jest bardziej elastyczny

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z
- **(FIND 'b '(a x b c))** podobnie jak *MEMBER*, ale jest bardziej elastyczny
- **(SUBSETP 'b '(a c b))** zawieranie elementu, zwraca T / NIL

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z
- **(FIND 'b '(a x b c))** podobnie jak *MEMBER*, ale jest bardziej elastyczny
- **(SUBSETP 'b '(a c b))** zawieranie elementu, zwraca T / NIL
- **(UNION '(a) '(b))** suma zbiorów

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z
- **(FIND 'b '(a x b c))** podobnie jak *MEMBER*, ale jest bardziej elastyczny
- **(SUBSETP 'b '(a c b))** zawieranie elementu, zwraca T / NIL
- **(UNION '(a) '(b))** suma zbiorów
- **(SET-DIFFERENCE '(a b) '(b))** różnica zbiorów

Podstawy - funkcje do operowania na listach

- **(REVERSE '(1 2 3))** zwraca odwróconą listę.
- **(LAST '(a b c))** zwraca ostatnią komórkę listy, zwraca NIL
- **(REMOVE '2 '(1 2 3))** usuwa element z listy
- **(APPEND x y)** łącz listy
- **(MEMBER 'z '(a z b))** ustaw członkostwo, zwraca pierwszy ogon, w którym jest z
- **(FIND 'b '(a x b c))** podobnie jak *MEMBER*, ale jest bardziej elastyczny
- **(SUBSETP 'b '(a c b))** zawieranie elementu, zwraca T / NIL
- **(UNION '(a) '(b))** suma zbiorów
- **(SET-DIFFERENCE '(a b) '(b))** różnica zbiorów
- **(INTERSECTION '(a b c) '(b))** przekrój zbiorów

Podstawy - zmienne

Istnieja dwa oddzielne typy zmiennych - leksykalne i dynamiczne (nazywane także specjalnymi). Zmienna dynamiczna zawsze ma tylko jedna wartosc - jej zmiana w jakimkolwiek miejscu w programie zmienia jej wartosc globalna.

Podstawy - zmienne dynamiczne

- Zmienne tworzone za pomoca formy **setq** sa globalne, czyli istnieja i mozna sie do nich odwolywac z dowolnego miejsca programu od chwili wykonania tej formy. Takie zmienne mozna rowniez jawnie zadeklarowac za pomoca makrodefinicji **defvar**. Aby 'ostrzec' innych programistów, że dana zmienna jest dynamiczna, powinno się otaczać jej nazwę gwiazdkami.

np.

```
> (setq *x* 10)
> (setq x "hello")
> (defvar *z* 10)
```

Podstawy - zmienne lokalne

- Definiujemy i określamy ich zakres przy pomocy **let**. Dzięki temu możemy odgrodzić pewne części programu od innych. Wartości początkowe wszystkich zmiennych są obliczane jednocześnie i dopiero potem zmienne te mogą być używane. *var1* to pierwsza zmienna, którą chcemy zawęzić leksykalnie, zaś *val1*, to wartość początkowa tej zmiennej. Podobnie jest z *var2*. *Body* to fragment kodu, który zostanie wykonany ze zmiennymi *var1, var2...* zawężonymi leksykalnie.

np.

```
> (let ((var1 val1)
(var2 val2)...)
body)
```

Podstawy - zmienne lokalne (2)

- W formie **let*** obliczanie i nadawanie wartości początkowych przebiega sekwencyjnie (w kolejności zapisu), a każda definiowana zmienna może być używana natychmiast po nadaniu jej wartości początkowej, czyli także w formach określających wartości początkowe kolejnych zmiennych.

np.

```
> (let* ((x 10)
(y (+ x 20)))
(setq y (- y x))
(* x y))
```

Podstawy - funkcje

Tak jak w każdym współczesnym języku programowania, podstawowymi elementami programu w języku Lisp są definiowane przez programistę funkcje.

- Definicja funkcji formułowana jest za pomocą makrodefinicji **defun**.
- Po słowie defun występuje nazwa funkcji, a następnie lista nazw jej argumentów. Po liście nazw argumentów może być zapisana kolejno dowolna liczba form składających się na treść funkcji.

np.

(defun nazwa-f (argum-a1 argum-a2 ...)
forma ...)
Wywolanie funkcji: ***(nazwa-f argum-a1 argum-a2...)***

Podstawy - funkcje matematyczne

Kilka wybranych funkcji:

- **(evenp x),(oddp x)** ? zwracają t gdy x jest liczbą parzystą i nieparzystą oraz nil w przeciwnym przypadku
- **(expt x y)** ? zwraca wartość x podniesioną do potęgi y
- **(max x1 x2 ...)** ? zwraca maksymalną z wartości dowolnej liczby argumentów
- **(sin x), (cos x), (tan x)** ? zwracają wartość funkcji sinus, cosinus, tangens dla argumentu x
- **(abs x)** ? zwraca wartość bezwzględną argumentu x
- **(floor x), (ceiling x)** ? zwracają wartość argumentu zaokrąglona odpowiednio w dół i w górę do najbliższej liczby całkowitej
- **(random x)** - zwraca nieujemną liczbę pseudolosową mniejszą od wartości x i typu jak x

Podstawy - funkcje matematyczne (2)

- > (+ 2 3 4) => 2+3+4
- > (* 2 3 4) => 2*3*4
- > (- 4 3 1)
- > (/ 4 2 1)
- > (/ (+ 2 4) (* 2 3))

Podstawy - operatory relacji

- Jeśli będzie więcej niż dwa argumenty, wynik jest równoważny koniunkcji warunków powstałych przez zastosowanie odpowiedniej operacji porównywania do każdej pary dwóch kolejnych argumentów.

np.

```
> (= 3 4) => NIL
> (= 3 3 3) => T
> (< 2 3 4) => T
> (<= 2 4 4) => T
> (> 4 3 3) => NIL
> (>= 6 5 5) => T
```

Podstawy - instrukcje warunkowe

- **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 zwraca wartość fałszywej części.

np.

```
(if t ?tekst-prawdziwy ?tekst-falszywy)
(if nil ?tekst-prawdziwy ?tekst-falszywy)
```

Podstawy - instrukcje warunkowe (2)

- **WHEN, UNLESS** w przeciwieństwie do IF, zezwalają na dowolną liczbę instrukcji w swoich ciałach.

np.

```
> (when t 5)
> (when nil 5)
> (unless t 5)
> (unless nil 5)
> (when (< a b))
```

Podstawy - instrukcje warunkowe (3)

- Instrukcja **CASE** w LISP jest podobna do instrukcji switch w C. Klauzula *otherwise* oznacza, że jeśli x nie jest a, b, d, e lub f, instrukcja CASE ma zwrócić 9.

np.

```
> (setq x ?b)
> (case x
(a 5)
((d e) 7)
((b f) 3)
(otherwise 9))
```

Podstawy - instrukcje warunkowe (4)

- 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.

np.

```
> (setq a 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
```

Podstawy - Iteracje

- **(loop body)** - kolejno, w nieskonczonosc wykonuje formy zawarte w body. Przerwanie pętli nastąpi tylko przy pomocy instrukcji (return) lub (return var). W pierwszym przypadku zostanie zwrócona wartosc nil, w drugim zaś var.

np.

```
(setq a 4)
(loop (setq a (- a 1))
(when (< a 3) (return a)))
```

Podstawy - Iteracje

- **(loop body)** - kolejno, w nieskonczonosc wykonuje formy zawarte w body. Przerwanie pętli nastąpi tylko przy pomocy instrukcji (return) lub (return var). W pierwszym przypadku zostanie zwrócona wartosc nil, w drugim zaś var.

np.

```
(setq a 4)
(loop (setq a (- a 1))
(when (< a 3) (return a)))
```

- **dotimes** pozwala w łatwy sposób wykonać kilkakrotnie ten sam fragment kodu ze zmiennymi będącymi kolejnymi liczbami naturalnymi. Ma ona postać **(dotimes (var n) body)**.

np.

```
(dotimes (i 4) (print i))
```

Podstawy - Iteracje (2)

- **dolist** Ma ona postać **(dolist (var list) body)**. Każdy element list jest kolejno przypisywany do zmiennej *var*, po czym wykownywane jest *body*, aż do osiągnięcia końca listy.

np.

```
(dolist (x ?(a b c)) (print x))
```

Podstawy - Iteracje (2)

- **dolist** Ma ona postać **(dolist (var list) body)**. Każdy element list jest kolejno przypisywany do zmiennej *var*, po czym wykownywane jest *body*, aż do osiągnięcia końca listy.

np.

```
(dolist (x ?(a b c)) (print x))
```

- pozostałe iteracje
  - **do** - najbardziej skomplikowana instrukcja związana z iteracją
  - **mapcar** - wywołuje pewną funkcję na każdym elemencie listy
  - **mapcan** - usuwa wartości nil z listy wynikowej.

Podstawy - operacje wejścia/wyjścia

- **Napis** - w LISP zapisujemy przy pomocy tekstu ograniczonego podójnym cudzysłowem, *np.* *'napis'*

Podstawy - operacje wejścia/wyjścia

- **Napis** - w LISP zapisujemy przy pomocy tekstu ograniczonego podójnym cudzysłowem, *np. 'napis'*
  - **char** - pozwala na dostęp do n-tego elementu napisu, może przyjmować tylko dwa argumenty - napis oraz pozycję w napisie, indeksowanie jest od zera, ***np.(char napis pozycja)***

Podstawy - operacje wejścia/wyjścia

- **Napis** - w LISP zapisujemy przy pomocy tekstu ograniczonego podójnym cudzysłowem, *np. 'napis'*
  - **char** - pozwala na dostęp do n-tego elementu napisu, może przyjmować tylko dwa argumenty - napis oraz pozycję w napisie, indeksowanie jest od zera, ***np.(char napis pozycja)***
  - **format** - pozwala wydrukwać na dowolne wyjście napisy, których elementy mogą zostać zadane jako argumenty jej wywołania.
    np. (format destination string arg1 arg2 ...) - Destination przyjmuje najczęściej wartości nil oraz t. Argument ten określa gdzie wynikowy łańcuch ma zostać złożony. nil oznacza, że wynikowy łańcuch ma zostać zwrócony przez instrukcję format, podczas gdy t oznacza, że wynikowy łańcuch ma zostać wydrukowany na standardowym wyjściu, a instrukcja zwraca nil.

Podstawy - operacje wejścia/wyjścia (2)

- W napisie wykorzystującym instrukcję *format* jednak muszą wystąpić odpowiednie znaki, które informują tą funkcję, gdzie mają się pojawić argumenty i jak mają zostać wydrukowane. Najczęściej stosowane zestawienia znaków, to ~D oraz ~S. Piewrszy z nich pozwala wstawić liczbę do napisu, drugi zaś inny napis bądź symbol. nnymi równie często pojawiającymi się kombinacjami znaków są ~& oraz ~%. Pierwsza z nich powoduje przejście do nowej lini, jeśli bieżąca linia nie jest pusta, podczas gdy druga, powoduje bezwarunkowe przejści do nowej linii.

Podstawy - operacje wejścia/wyjścia (2)

- - W napisie wykorzystującym instrukcję *format* jednak muszą wystąpić odpowiednie znaki, które informują tą funkcję, gdzie mają się pojawić argumenty i jak mają zostać wydrukowane. Najczęściej stosowane zestawienia znaków, to ~D oraz ~S. Piewrszy z nich pozwala wstawić liczbę do napisu, drugi zaś inny napis bądź symbol. nnymi równie często pojawiającymi się kombinacjami znaków są ~& oraz ~%. Pierwsza z nich powoduje przejście do nowej lini, jeśli bieżąca linia nie jest pusta, podczas gdy druga, powoduje bezwarunkowe przejści do nowej linii.
- Do wprowadzania informacji przez użytkownika służy instrukcja **read**. Zwraca ona po prostu wartość odczytaną ze standardowego wejścia.
  np. ... (let ((a (read))))...

Podstawy - operacje wejścia/wyjścia (3)

- **Odczyt z pliku** realizujemy za pomocą instrukcji **with-open-file**.
  Ma ona postać ***(with-open-file (var pathname) body)***, gdzie *var* to nazwa zmiennej, z którą będzie utożsamiany strumień do otwartego pliku, *pathname* to ścieżka dostępu do tego pliku, zaś *body* zawiera instukcje dokonujące operacji na otwartym pliku. Zaraz po opuszczeniu tej instrukcji, połączenie do pliku jest zamykane.

Podstawy - operacje wejścia/wyjścia (3)

- **Odczyt z pliku** realizujemy za pomocą instrukcji **with-open-file**.
  Ma ona postać ***(with-open-file (var pathname) body)***, gdzie *var* to nazwa zmiennej, z którą będzie utożsamiany strumień do otwartego pliku, *pathname* to ścieżka dostępu do tego pliku, zaś *body* zawiera instukcje dokonujące operacji na otwartym pliku. Zaraz po opuszczeniu tej instrukcji, połączenie do pliku jest zamykane.

otworz.lsp

```
(with-open-file (plik "project/testy.txt")
(let ((a (read plik)))
(format t "~&Wartosc wyswietlona = ~S" a)))
```

Podstawy - operacje wejścia/wyjścia (4)

- **Zapis do pliku** realizujemy za pomocą instrukcji **with-open-file** jednocześnie dodatkowo musimy zastosować dwa słowa kluczowe **:direction, :output**, całą zaś forma przyjmuje postać
(with-open-file (var pathname :direction :output) body).

Podstawy - operacje wejścia/wyjścia (4)

- **Zapis do pliku** realizujemy za pomocą instrukcji **with-open-file** jednocześnie dodatkowo musimy zastosować dwa słowa kluczowe **:direction, :output**, całą zaś forma przyjmuje postać ***(with-open-file (var pathname :direction :output) body)***.

np.

```
(with-open-file (plik "project/jeden_pierw.txt" :direction :output)
(format plik "~&Rownanie ~Sx+~Sy+~S ma 1 pierwiastek, x1=
~S" a b c x1))
```

Programiki - wykorzystanie listy asocjacyjnej

slownik.lsp

```
(setq slownik '((jeden one)
(dwa two)
(trzy three)
(cztery four)
(piec five)))
(defun tlumacz (slowo)
(cadr(assoc slowo slownik)))
```

Programiki - delta

Programik wykorzystuje:
zmienne lokalne i globalne, operacje wejścia/wyjścia, zapis do pliku, działania matematyczne, instrukcje warunkowe, operatory relacji

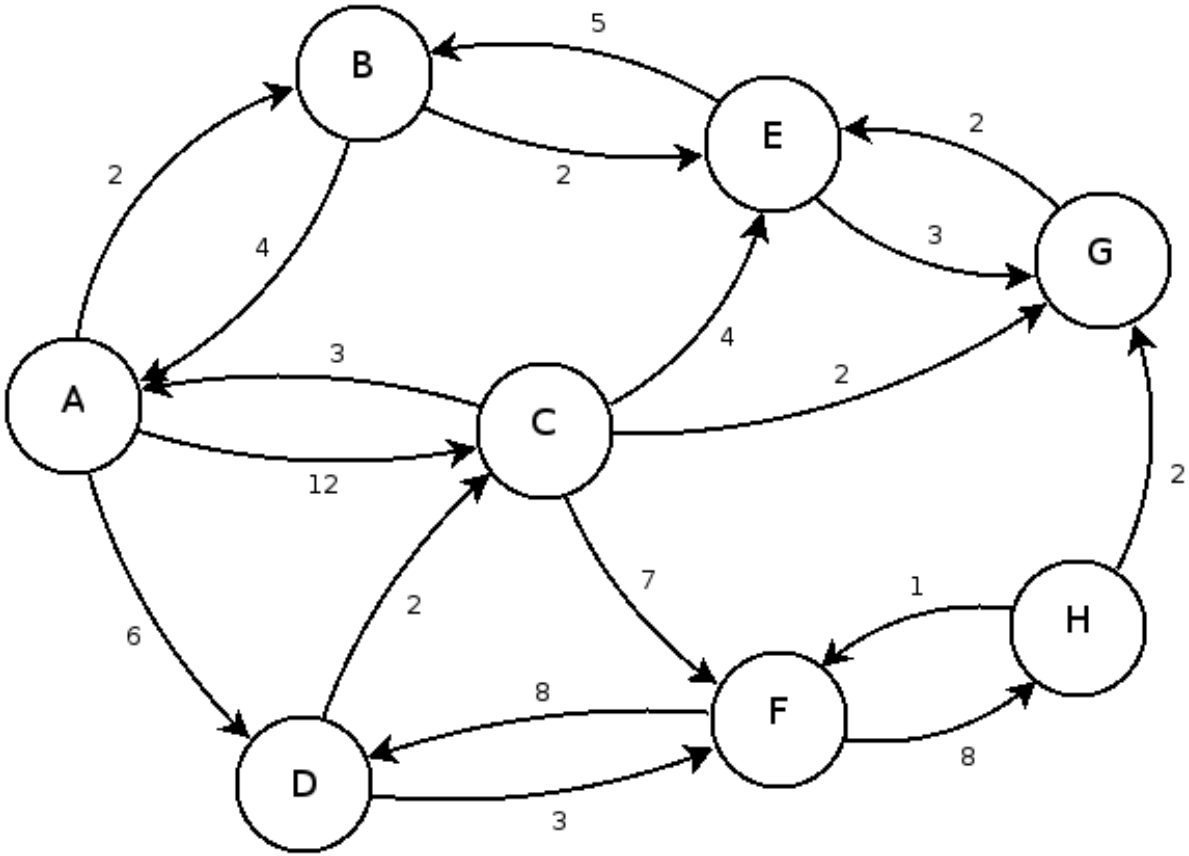
```
(defun delta ()
   (format t "~&Podaj parametr rownania - a:")
   (let ((a (read)))
        (format t "~&Podaj parametr rownania - b:")
        (let ((b (read)))
           (format t "~&Podaj parametr rownania - c:")
           (let ((c (read)))
                (setq wynik (- (* b b) (* 4 a c)))
                (setq sqrtdelt (sqrt wynik))

                (when (< wynik 0)
                   (format t "~&Rownanie ~Sx+~Sy+~S nie ma pierwiastkow" a b c)
                )
                (when (= wynik 0)
                   (setq x1 (/ (- b) (* 2 a)))
                   (format t "~&Rownanie ~Sx+~Sy+~S ma 1 pierwiastek, x1= ~S" a b c x1)
                   (with-open-file (plik "project/jeden_pierw.txt" :direction :output)
                        (format plik "~&Rownanie ~Sx+~Sy+~S ma 1 pierwiastek, x1= ~S" a b c x1)
                   )
                )
                (when (> wynik 0)
                   (setq x1 (/ (+ (- b) (sqrt wynik)) (* 2 a)))
                   (setq x2 (/ (- (- b) (sqrt wynik)) (* 2 a)))
                   (format t "~&Rownanie ~Sx+~Sy+~S ma 2 pierwiastki, p1: ~S oraz p2: ~S" a b c x1 x2)
                   (with-open-file (plik "project/dwa_pierw.txt" :direction :output)
                        (format plik "~&Rownanie ~Sx+~Sy+~S ma 2 pierwiastki, p1= ~S oraz p2= ~S" a b c x1 x2)
                   )
                )
           )
        )
   )
)
```

Programiki - przeszukiwanie grafu

Przyjrzyjmy się problemowi wielokrotnie poruszanemu jakim jest przeszukiwanie grafu. Również Lisp może sobie z nim poradzić na wiele sposobów. Przedstawimy prosty program rozwiązujący problem wyszukania najlepszej ścieżki w grafie(o najniższym koszcie).

Programiki - przeszukiwanie grafu



Programiki - przeszukiwanie grafu

Implementacja grafu

```
(defparameter *orig-db* '((a b 2 c 12 d 6)
(b a 4 e 2)
(c a 3 e 4 g 2 f 7)
(d c 2 f 3)
(e b 5 g 3)
(f d 8 h 8)
(h f 1 g 2)
(g e 2)))
(defvar *db* *orig-db*)
```

Programiki - przeszukiwanie grafu

Efekty testowania programu:

```
sh-3.2$ clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.44.1 (2008-02-23) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2008

Type :h and hit Enter for context help.

[1]> (load "sw.lisp")
;; Loading file sw.lisp ...
;; Loaded file sw.lisp
T
[2]> (find-paths 'A 'G)
((G H F C D A) (G H F C A) (G H F D A) (G E C D A) (G C D A) (G E C A)
 (G E B A) (G C A))
[3]> (find-best-path 'A 'G)
(A C G)
[4]>
```

Programiki - przeszukiwanie grafu

Komentarz

Przedstawione rozwiązanie jest bardzo proste, bardziej zaawansowane rozwiązanie problemu przeszukiwania ścieżki w grafie można znaleźć w książce “Paragigms of AI Programming” Peter’a Norviga.

LISP oczami innych

Richard Stallman

Lisp będący najpotężniejszym i najprzyzwoitszym z języków, to język, który projekt GNU zawsze preferuje.

Alan Perlis

Programista Lisp zna wartość wszystkich rzeczy, ale nie zna kosztu żadnej z nich.

Gregory Chaitin

Sądzę, że jest to jedyny język programowania, który można szanować pod względem matematycznym, gdyż tylko o nim jestem w stanie udowadniać teorie!

LISP oczami innych (2)

Kent Pitman

Proszę nie przyjmować, że Lisp nadaje się tylko do programowania Animacji i Grafiki, SI, Bioinformatyki, B2B i E-Commerce, Zbierania Danych, aplikacji EDA/Semiconductor, Systemów Eksperckich, Finansów, Inteligentnych Agentów, Zarządzania Wiedzą, Mechanicznych CAD, Modelowania i Symulacji, Naturalnych Języków, Optymalizacji, Badań i Rozwoju, Analizy Ryzyka, Planowania, Telekomunikacji i Tworzenia Stron WWW tylko dlatego, że te rzeczy zostały wymienione na liście.

- Komiksy z serii xkcd o LISP: http://www.xkcd.com/297/

Szybki start - przydatne linki

Informacje ogólne, tutoriale, kompilatory

- *Lisp wg wiki - http://pl.wikipedia.org/wiki/Lisp*
- *Common Lisp wg wiki*
- *Common Lisp wiki - http://www.cliki.net/index*
- *Wprowadzenie do języka Common Lisp*
- *LISPouczek - http://lisp.dziwisz.org/index.php*
- *Przewodnik po języku Common Lisp V 1.01 z komentarzami*
- *Oficjalna strona CLISP - http://clisp.cons.org/*
- http://cl-cookbook.sourceforge.net/index.html
- *interpretery LISP*
- *http://www.lispworks.com/*
- Rachunek lambda
- "Paragigms of AI Programming" Peter'a Norviga

Spis treści

Koniec

Dziękujemy za uwagę.