Common LISP
Instrukcje Sterujące

Jakub Kowalski

Instytut Informatyki, Uniwersytet Wrocławski

27 października 2010

Reguły obliczania programu

```
(list
  (format t "~A " 1)
  (format t "~A " 2)
  (format t "~A " 3))
```

Reguły obliczania programu

```
(list
  (format t "~A " 1)
  (format t "~A " 2)
  (format t "~A " 3))
```

```
1 2 3
(NIL NIL NIL)
```

Reguły obliczania programu

```
(list
  (format t "~A " 1)
  (format t "~A " 2)
  (format t "~A " 3))
```

```
1 2 3
(NIL NIL NIL)
```

Instrukcje sterujące łamią standardowe reguły obliczania i pozwalają na sterowanie kolejnością wykonania instrukcji w programie.

Plan prezentacji

1. Grupowanie instrukcji
   - Bloki: `progn`, `block`, `tagbody`
   - Kontekst: `let`, `let*`, `destructing-bind`, `flet`, `labels`
2. Instrukcje warunkowe
   - Pochodne `if`: `when`, `unless`, `cond`, `case`
3. Instrukcje iterujące
   - `loop`
   - `do`, `dolist`, `dotimes`, `mapc`, `mapcar`
4. Wyjątki
   - `throw/catch`, `error`, `unwind-protect`

Grupowanie instrukcji

Konstrukcja progn

Wywołuje po kolei wyrażenia podane jako argumenty i zwraca wartość ostatniego.

```
(progn
  (format t   "~A " 1)
  (format nil "~A " 2) ; Zwraca napis jako string
  (format t   "~A " 3)
  (format nil "~A " 4))
```

```
1 3
"4 "
```

Konstrukcja `block`

Działa jak `progn` o zdefiniowanej nazwie, ale dodatkowo pozwala na natychmiastowe zakończenie wykonywania ciągu instrukcji i zwrócenie zadanej wartości.

```
(block nazwaBloku
  (format t "Start.")
  (return-from nazwaBloku 'Wyskakujemy)
  (format t "Koniec."))
```

```
Start.
WYSKAKUJEMY
```

```
(block nil
  (return 666))
```

```
666
```

Automatycznie tworzone bloki

Bloki o nazwie `nil` tworzą się automatycznie w ciałach niektórych instrukcji, na przykład z rodziny `do`, `loop`.

```
(dotimes (n 10) ; wykonuje pętlę zadaną liczbę razy
  (format t "~A " n)
  (if (> n 3) (return n)))
```

```
0 1 2 3 4
4
```

Automatycznie tworzone bloki c.d.

Zdefiniowanie funkcji automatycznie tworzy blok o takiej samej nazwie jak funkcja.

```
(defun fun(n)
  (+ (block blok
       (if (> n 0)
           (return-from fun 'escape)
           (return-from blok 60)))
     606))
```

```
> (fun 0)
666
> (fun 1)
ESCAPE
```

Instrukcje `return...` pozwalają na wychodzenie z bloków o różnym zagłębieniu.

Konstrukcja tagbody

Działa jak progn z tą różnicą, że pozwala na definiowanie etykiet i wykonywanie skoków.

```
(tagbody
  (setf x 0)
 top                    ; etykieta
  (setf x (+ x 1))
  (format t "~A " x)
  (if (< x 10) (go top))) ; skok
```

```
1 2 3 4 5 6 7 8 9 10
NIL
```

tagbody jest automatycznie zdefiniowane w ciałach większości operacji iterujących.

Konstrukcja `let`

Tworzy nowy kontekst w którym występują zadeklarowane i zainicjowane zmienne (w razie konieczności przykrywając już istniejące).

```
(let ((x 3) ; jako pierwszy argument pobiera
      (y 4)); listę par (zmienna wartość)
  (+ x y))
```

```
7
```

`let` został skonstruowany tak, aby zachowywać się jak wywołanie funkcji.

Powyższy przykład jest równoważny następującemu lambda wyrażeniu:

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

Konstrukcja `let*`

Jedną z niedogodności `let` jest niemożność definiowania zależnych od siebie zmiennych, co jest natomiast wykonalne w konstrukcji `let*`.

Poniższe wyrażenie zapisane jako `let` nie byłoby prawidłowe

```
(let* ((x 2)
       (y (+ x 3))
  (+ x y))
```

ponieważ byłoby równoważne następującemu wyrażeniu

```
((lambda (x y) (+ x y)) 2 (+ x 3))
```

Natomiast jako `let*` jest równoważne ciągowi zagnieżdżonych instrukcji `let`:

```
(let ((x 2))
  (let ((y (+ x 3)))
    (+ x y)))
```

Konstrukcja destructing-bind

To makro uogólniające `let` pobiera drzewo zmiennych do zadeklarowania i odpowiadające drzewo wartości które mają im być przypisane.

```
(destructuring-bind (w (x y) . z) '(1 (2 3) 4 5)
  (list w x y z))
```

```
(1 2 3 (4 5))
```

```
(destructuring-bind ((x (y)) (z))
                    '(((1 2) (3)) ((4 (5 6))))
  (list x y z))
```

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

Konstrukcja `flet`

Tworzy nowy kontekst w którym występuje zadeklarowana lokalna funkcja

```
(flet
  ((nazwa (x y) ; definicja funkcji
   (format t "Args: ~A,~A; " x y) (+ x y)))
  (format t "Main block; ")
  (+ (nazwa 1 2) (nazwa 3 4)))
```

```
Main block; Args: 1,2; Args: 3,4;
10
```

Funkcja zadeklarowana `flet` nie może wywoływać samej siebie

Konstrukcja `labels`

Nie posiada ograniczeń `flet`, pozwala więc na deklarowanie nazwanych lokalnie funkcji rekurencyjnych.

```
(labels
  ((nazwa (x) (format t "Arg: ~A; " x)
              (if (= x 1) 1 (+ x (nazwa (- x 1))))))
  (format t "Main block; ")
  (nazwa 4))
```

```
Main block; Arg: 4; Arg: 3; Arg: 2; Arg: 1;
10
```

Test

Co powinno być w linijce ze znakiem zapytania?

```
(defun fun (x y)
    (let ((y x) (x y))
    (tagbody
     bot
      (block blok
        (if (= y 0) (return-from blok))
        (if (= x 2) (setf y 0))
        (return-from fun (list x y)))
      (setf y x)
      ?
      (go bot))))
```

```
> (fun 0 2)
(2 0)
```

A. `(setf y 0)` **B.** `(return-from fun (list x y))`
C. (format nil "Litwo Ojczyzno moja...")

Test

Co powinno być w linijce ze znakiem zapytania?

```
(defun fun (x y)
    (let ((y x) (x y))
    (tagbody
     bot
      (block blok
        (if (= y 0) (return-from blok))
        (if (= x 2) (setf y 0))
        (return-from fun (list x y)))
      (setf y x)
      ?
      (go bot))))
```

```
> (fun 0 2)
(2 0)
```

A. `(setf y 0)` **B.** `(return-from fun (list x y))`

C. (format nil "Litwo Ojczyzno moja...")

Instrukcje warunkowe

Konstrukcja `if`

Sprawdza warunek podany w pierwszym argumencie, po czym w zależności od tego czy jest prawdą czy nie, wykonuje odpowiednio swój drugi lub trzeci argument. Trzeci argument `if` jest opcjonalny, domyślnie wynosi `nil`.

```
(if (> 3 2)   ; warunek
    (+ 1 2)   ; then
    (+ 1 1))  ; else
```

Konstrukcja `if`

Sprawdza warunek podany w pierwszym argumencie, po czym w zależności od tego czy jest prawdą czy nie, wykonuje odpowiednio swój drugi lub trzeci argument. Trzeci argument `if` jest opcjonalny, domyślnie wynosi `nil`.

```
(if (> 3 2)   ; warunek
    (+ 1 2)   ; then
    (+ 1 1))  ; else
```

`if` pozwala na umieszczenie jedynie po jednej instrukcji w argumentach then/else. Jeśli chcemy napisać w tych miejscach coś większego trzeba explicite wykorzystać np. `progn`.

```
(if (= x 0)
    (progn
      (format t "then" )
      (+ x 2))
    (format nil "else" )
```

Konstrukcje `when`, `unless`

Problem ten rozwiązują częściowo instrukcje `when` i `unless`. Ich pierwszym argumentem jest warunek, który w przypadku `when` musi być prawdziwy, zaś dla `unless` fałszywy, aby program wykonywał kolejno wszystkie instrukcje podane jako dalsze argumenty.

```
(when (= x 0)
  (format t "then" )
  (+ x 2))
```

```
(unless (= x 0)
  (format t "else" )
  (+ x 1))
```

Konstrukcja cond

Odpowiada zagnieżdżonym instrukcjom `if`. Sprawdza po kolei podane warunki, aż któryś nie okaże się prawdą, jeśli tak wywołuje kolejne instrukcje przypisane do tego warunku, zwracając wartość ostatniej z nich. Jeśli żaden warunek nie będzie spełniony, zwracany jest `nil`

```
(cond ((< x 0) (format t "-x") -1)
      ((= x 0) (format t "0")  (+ -1 1))
      (t       (format t "+x") 1))
```

W przypadku gdy po warunku nie ma żadnych instrukcji, zwracana jest wartość samego warunku.

```
> (cond (666))
666
```

Konstrukcja case

Pozwala na dopasowanie do serii wartości. W jednej klauzuli występuje porównanie (używające `eql`) do pojedynczej wartości lub listy wartości. W przypadku powodzenia wykonywane są instrukcje przypisane do tych wartości.
Zawsze wykonywaną klauzulę można oznaczyć jako `t` lub `otherwise`. W przypadku braku dopasowania, lub gdy po dopasowaniu nie ma instrukcji, zwracany jest `nil`.

```
(case x
  ((-3 -2 -1) 'ujemna)
  (0          'zero)
  (otherwise  (if (> x 100) 'duza 'mala)))
```

```
> (case 666 (666))
nil
```

Test

Jaki będzie efekt wywołania programów?

Test

Jaki będzie efekt wywołania programów?

```
(when (null '(nil))
  (format t "then" )
  (return 'escape)
  (+ 1 2))
```

A. "then" ESCAPE **B.** "then" 3 **C.** **error** **D.** nil

Test

Jaki będzie efekt wywołania programów?

```
(when (null '(nil))
  (format t "then" )
  (return 'escape)
  (+ 1 2))
```

A. "then" ESCAPE **B.** "then" 3 **C.** **error** **D.** **nil**

Test

Jaki będzie efekt wywołania programów?

```
(when (null '(nil))
  (format t "then" )
  (return 'escape)
  (+ 1 2))
```

A. "then" ESCAPE **B.** "then" 3 **C.** **error** **D.** **nil**

```
(if (integerp (+ -1/3 8/6))
    (lambda (x) (+ x 1) (+ 1 1)))
    (+ 2 2)
```

A. 2 **B.** 3 **C.** 4 **D.** nil

Test

Jaki będzie efekt wywołania programów?

```
(when (null '(nil))
  (format t "then" )
  (return 'escape)
  (+ 1 2))
```

A. "then" ESCAPE **B.** "then" 3 **C.** **error** **D.** **nil**

```
(if (integerp (+ -1/3 8/6))
    (lambda (x) (+ x 1) (+ 1 1)))
    (+ 2 2)
```

A. 2 **B.** 3 **C.** **4** **D.** nil

Instrukcje iterujące

Konstrukcja `loop`

Najprostrza z możliwych pętli, kolejno wywołuje instrukcje przekazane jej jako argumenty, a gdy zrobi ostatnią przechodzi z powrotem do pierwszej. Istnieje możliwość opuszczenia pętli `loop` za pomocą instrukcji `return`, ponieważ automatycznie tworzy ona blok `nil`.

```
(setf x 0)
(loop
  (format t "obrot ~A~%" x)
  (setf x (+ x 1))
  (if (= x 4) (return 'starczy)))
```

```
obrot 0
obrot 1
obrot 2
obrot 3
STARCZY
```

Rozszerzenia `loop`

Słowa kluczowe dla `loop` to: `across`, `and`, `below`, `collecting`, `counting`, `finally`, `for`, `from`, `summing`, `then`, `to`.

Przykłady:

```
(loop for i from 1 to 10 collecting i)
```

```
(1 2 3 4 5 6 7 8 9 10)
```

```
(loop for x from 1 to 10 summing (* x x))
```

```
385
```

```
(loop for x across "aXXbcYbcaXZcZ"
      counting (find x "XYZ"))
```

```
6
```

Konstrukcja do

Pierwszym argumentem jest lista specyfikacji dla zmiennych. Elementy tej listy są listami postaci (*zmienna wartość_początkowa aktualizacja*), lub skróconymi (bez ostatniego, lub dwóch ostatnich elementów).
Pętla tworzy daną zmienną, przypisuje jej początkową wartość, po czym po każdej iteracji wywołuje instrukcję odpowiedzialną za jej aktualizację.
Kolejnym argumentem jest lista zawierające wyrażenie (którego prawdziwość jest sprawdzana po zastosowaniu aktualizacji), jeśli jest ono prawdziwe pętla zwaraca zadaną wartość.
Wszystkie kolejne argumenty stanowią ciało pętli.

```
(do ((zmienna1 wartość_początkowa1 aktualizacja1)
     ...
     (zmiannaN wartość_początkowaN aktualizacjaN))
    (warunek zwracana_wartość)
    (instrukcja1)
    ...
    (instrukcjaN))
```

Pętla do z zależnymi zmiennymi iterowanymi

Gdy pętla wykorzystuje więcej niż jedną zmienną iterowaną i zmienne te są od siebie zależne (podczas aktualizacji jednej z nich brana jest pod uwagę wartość innej), to podczas aktualizacji wszystkich zmiennych brane są pod uwagę wartości sprzed zmian.

```
(let ((x 'a))
  (do ((x 1 (+ x 1))
       (y x x))
      ((> x 5))
    (format t "(~A ~A) " x y)))
```

```
(1 A) (2 1) (3 2) (4 3) (5 4)
NIL
```

Konstrukcja do*

Pętla do* różni się od do tym, czym let* od let - czyli dynamiczną aktualizacją zmiennych.

```
(let ((x 'a)) ; nieużywana zmienna
  (do* ((x 1 (+ x 1))
        (y x x))
       ((> x 5))
    (format t "(~A ~A) " x y)))
```

```
(1 1) (2 2) (3 3) (4 4) (5 5)
NIL
```

Esencja do

Przykład wykorzystania drugiej iterowanej zmiennej jako akumulatora, z równoczesnym przerzuceniem obliczeń z ciała pętli do fazy aktualizacji zmiennej.

```
(defun silnia (n)
  (do ((j n (- j 1))
       (f 1 (* j f)))
      ((= j 0) f)))
```

Konstrukcja `dolist`

Pętla przypisuje do zadanej zmiennej kolejne elementy listy. Trzeci element pierwszego argumentu zostanie zwrócony jako wartość pętli - domyślnie wynosi `nil`

```
(dolist (x '(a b c d) 'done)
  (format t "~A " x))
```

```
A B C D
DONE
```

Konstrukcja `dotimes`

Pętla o podobnej konstrukcji jak `dolist`. Wykonuje zadane n iteracji, przypisując zmiennej kolejne wartości od 0 do $n-1$.

```
(dotimes (x 5 (+ x 1))
  (format t "~A " x))
```

```
0 1 2 3 4
6
```

Konstrukcja mapc

Pobiera funkcję oraz tyle list ile wynosi arność tej funkcji. W i-tym kroku wywołuje ją dla i-tych elementów list. Koniec pętli następuje, gdy dojdzie do końca którejś z podanych list.
Zawsze zwraca swój drugi argument.

```
(mapc #'(lambda (x y z)
                (format t "(~A~A~A) " x y z))
      '(k r g b more)
      '(o a a a)
      '(t k z l))
```

```
(KOT) (RAK) (GAZ) (BAL)
(K R G B MORE)
```

Konstrukcja mapcar

Działa podobnie jak mapc, ale w trakcie działania z rezultatów zwracanych przez zadaną funkcję buduje nową listę, którą zwraca jako rezultat.

```
(mapcar #'(lambda (x y z)
                  (format nil "~A~A~A" x y z))
        '(k r g b more)
        '(o a a a)
        '(t k z l))
```

```
("KOT" "RAK" "GAZ" "BAL")
```

Zagadka

Jaki będzie efekt działania poniższej instrukcji?

```
(dotimes (n 3)
  (dotimes (k 5 'val)
    (format t "~A " k)
    (if (> k 2) (return n)))
  (format t "~%"))
```

Zagadka

Jaki będzie efekt działania poniższej instrukcji?

```
(dotimes (n 3)
  (dotimes (k 5 'val)
    (format t "~A " k)
    (if (> k 2) (return n)))
  (format t "~%"))
```

```
0 1 2 3
0 1 2 3
0 1 2 3
NIL
```

Wyjątki

Konstrukcje throw/catch

Za pomocą instrukcji `throw` rzuca się wyjątek o podanej nazwie i wartości, natomiast `catch` łapie wyjątki jeśli się odpowiednio nazywają i zwraca ich wartość.
Jeśli wyjątek trafi na `catch` nieodpowiedniego typu to wykonanie przenosi się poziom wyżej. Jeżeli w ogóle nie zostanie złapany – powoduje błąd.

```
(defun sub ()
  (throw 'uaaa 666))

(defun super ()
  (catch 'uaaa
    (sub)
  (format t "To␣się nie pokaże" )))
```

```
> (super)
666
```

Konstrukcja error

Umożliwia zasygnalizowanie błędu wykonania i wywołuje system obsługi błędów Lispa.

```
(error "Marchewka")
```

```
Marchewka
   [Condition of type SIMPLE-ERROR]

Restarts:
 0: [RETRY] Retry SLIME REPL evaluation request.
 1: [ABORT] Return to SLIME's top level.
 2: [ABORT-BREAK] Reset this thread
 3: [ABORT] Kill this thread

Backtrace:
  0: (CCL::CALL-CHECK-REGS ERROR "Marchewka")
  1: (CCL::CHEAP-EVAL (ERROR "Marchewka"))
--more--
```

Konstrukcja unwind-protect

Pozwala na zadeklarowanie bloku kodu który będzie wykonywany niezależnie od powstałych w nim wyjątków.
Instrukcja bierze dowolną liczbę argumentów i zwraca wartość pierwszego z nich.

```
(setf x 1)
(catch 'uaaa
  (unwind-protect
    (throw 'uaaa 666)
    (setf x 2)))
```

```
666
> x
2
```

Podsumowanie

- Grupowanie instrukcji
- Instrukcje warunkowe
- Instrukcje iterujące
- Wyjątki

Bibliografia


Paul Graham, *ANSI Common Lisp*, Prentice Hall 1996


Peter Seibel, *Practical Common Lisp*, Apress 2005

Koniec