Common Lisp - funkcje i zmienne

Wojciech Jedynak

Instytut Informatyki Uniwersytetu Wrocławskiego

27 października 2010

Plan prezentacji

Plan prezentacji

Krótkie przypomnienie

Przykładowa definicja

```
(defun square (x)
  (* x x))
```

Przykładowe wywołanie

```
(square 17)   ;; 243
```

Krótkie przypomnienie

Przykładowa definicja

```
(defun square (x)
  (* x x))
```

Przykładowe wywołanie

```
(square 17)   ;; 243
```

Krótkie przypomnienie

Przykładowa definicja

```
(defun square (x)
  (* x x))
```

Przykładowe wywołanie

```
(square 17)   ;; 243
```

Plan prezentacji

Ogólna postać funkcji

Składnia

```
(defun nazwa (parametry*)
  doc-str?
  ciag-wyrazen?)
```

Przykłady

```
;; Pusta lista wyrazen, funkcja zawsze zwraca NIL
(defun pass ())

;;Troche bardziej realistyczny przyklad
(defun verbose-id (x)
  "This is a doc string"
  (format t "I got ~D~%." x)
  (save-the-world ())
  x)
```

Przykłady

```
;; Pusta lista wyrazen, funkcja zawsze zwraca NIL
(defun pass ())

;;Troche bardziej realistyczny przyklad
(defun verbose-id (x)
  "This is a doc string"
  (format t "I got ~D~%." x)
  (save-the-world ())
  x)
```

Sposoby podawania parametrów

Rodzaje argumentów

W Common Lisp mamy następujące opcje:

- Wymagane parametry pozycyjne
- Opcjonalne parametry pozycyjne
- Parametry jako lista
- Nazwane parametry (ang. keyword arguments)

Sposoby podawania parametrów

Rodzaje argumentów

W Common Lisp mamy następujące opcje:

- Wymagane parametry pozycyjne
- Opcjonalne parametry pozycyjne
- Parametry jako lista
- Nazwane parametry (ang. keyword arguments)

Sposoby podawania parametrów

Rodzaje argumentów

W Common Lisp mamy następujące opcje:

- Wymagane parametry pozycyjne
- Opcjonalne parametry pozycyjne
- Parametry jako lista
- Nazwane parametry (ang. keyword arguments)

Sposoby podawania parametrów

Rodzaje argumentów

W Common Lisp mamy następujące opcje:

- Wymagane parametry pozycyjne
- Opcjonalne parametry pozycyjne
- Parametry jako lista
- Nazwane parametry (ang. keyword arguments)

Sposoby podawania parametrów

Rodzaje argumentów

W Common Lisp mamy następujące opcje:

- Wymagane parametry pozycyjne
- Opcjonalne parametry pozycyjne
- Parametry jako lista
- Nazwane parametry (ang. keyword arguments)

Obowiązkowe parametry pozycyjne

- Podajemy listę zmiennych, pod które zostaną podstawione (związane) argumenty

Przykłady

```
(defun no-args ())

(defun foo (a b c))
```

- Podanie nieodpowiedniej liczby zmiennych kończy się zgłoszeniem wyjątku!
- A jeśli nie chcemy zawsze wszystkiego podawać?

Obowiązkowe parametry pozycyjne

- Podajemy listę zmiennych, pod które zostaną podstawione (związane) argumenty

Przykłady

```
(defun no-args ())

(defun foo (a b c))
```

- Podanie nieodpowiedniej liczby zmiennych kończy się zgłoszeniem wyjątku!
- A jeśli nie chcemy zawsze wszystkiego podawać?

Obowiązkowe parametry pozycyjne

- Podajemy listę zmiennych, pod które zostaną podstawione (związane) argumenty

Przykłady

```
(defun no-args ())

(defun foo (a b c))
```

- Podanie nieodpowiedniej liczby zmiennych kończy się zgłoszeniem wyjątku!
- A jeśli nie chcemy zawsze wszystkiego podawać?

Obowiązkowe parametry pozycyjne

- Podajemy listę zmiennych, pod które zostaną podstawione (związane) argumenty

Przykłady

```
(defun no-args ())

(defun foo (a b c))
```

- Podanie nieodpowiednej liczby zmiennych kończy się zgłoszeniem wyjątku!
- A jeśli nie chcemy zawsze wszystkiego podawać?

Opcjonalne parametry pozycyjne

- Przed pierwszym parametrem który ma być opcjonalny umieszczamy frazę &optional
- Możemy podać wartość domyślną, a także sprawdzić, czy dany argument został podany przy wywołaniu

Przykład

```
(defun test (&optional a (b 1) (c 6 c-p))
  (list a b (list c c-p)))

CL-USER> (test)
(NIL 1 (6 NIL))
CL-USER> (test 1 2 3)
(1 2 (3 T))
```

Opcjonalne parametry pozycyjne

- Przed pierwszym parametrem który ma być opcjonalny umieszczamy frazę &optional
- Możemy podać wartość domyślną, a także sprawdzić, czy dany argument został podany przy wywołaniu

Przykład

```
(defun test (&optional a (b 1) (c 6 c-p))
  (list a b (list c c-p)))

CL-USER> (test)
(NIL 1 (6 NIL))
CL-USER> (test 1 2 3)
(1 2 (3 T))
```

Parametry w postaci listy

- Po ostatnim parametrze, który ma być nazwanym argumentem umieszczamy frazę &args i nazwę zmiennej z którą ma zostać związana lista pozostałych argumentów

Przykład

```
(defun test (a b &rest params)
  (list a b params))

CL-USER> (test 1 2)
(1 2 NIL)
CL-USER> (test 1 6 7 8 9)
(1 6 (7 8 9))
```

Parametry w postaci listy

- Po ostatnim parametrze, który ma być nazwanym argumentem umieszczamy frazę &args i nazwę zmiennej z którą ma zostać związana lista pozostałych argumentów

Przykład

```
(defun test (a b &rest params)
  (list a b params))

CL-USER> (test 1 2)
(1 2 NIL)
CL-USER> (test 1 6 7 8 9)
(1 6 (7 8 9))
```

Nazwane parametry

- Przed pierwszym parametrem który ma być (opcjonalnym) nazwanym parametrem umieszczamy frazę &key
- Możemy podać wartość domyślną, a także sprawdzić, czy dany argument został podany przy wywołaniu

Przykład

```
(defun test (&key a (b 2) (c 3 c-p) d)
  (list a b (list c c-p) d))

CL-USER> (test)
(NIL 2 (3 NIL) NIL)
CL-USER> (test :a 1 :c 6)
(1 2 (6 T) NIL)
```

Nazwane parametry

- Przed pierwszym parametrem który ma być (opcjonalnym) nazwanym parametrem umieszczamy frazę &key
- Możemy podać wartość domyślną, a także sprawdzić, czy dany argument został podany przy wywołaniu

Przykład

```
(defun test (&key a (b 2) (c 3 c-p) d)
  (list a b (list c c-p) d))

CL-USER> (test)
(NIL 2 (3 NIL) NIL)
CL-USER> (test :a 1 :c 6)
(1 2 (6 T) NIL)
```

Używanie wielu wariantów jednocześnie

- Łączenie w ramach jednej definicji argumentów opcjonalnych i nazwanych bądź listowych i nazwanych prowadzi w niektórych przypadkach do dziwnych zachowań.

Używanie wielu wariantów jednocześnie cd.

```
(defun optional-key (&optional a b &key c)
  (list a b c))

(defun rest-key (&rest params &key a b c)
  (list a b c rest))
```

- Jak należy wywołać te funkcje, żeby doszło do czegoś dziwnego?

Plan prezentacji

Funkcje jako dane

- Chcielibyśmy traktować funkcje na równi z danymi, tj przekazywać je jako parametry, zwracać jako wyniki itp.
- Mamy jednak problem: umiemy wywołać konkretną funkcję, ale nie umiemy tego zrobić dla zmiennej (ani nawet utworzyć takiej zmiennej). [Podejście z Haskella/MLa nie działa]
- Na ratunek przybywają function, funcall i apply

Funkcje jako dane

- Chcielibyśmy traktować funkcje na równi z danymi, tj przekazywać je jako parametry, zwracać jako wyniki itp.
- Mamy jednak problem: umiemy wywołać konkretną funkcję, ale nie umiemy tego zrobić dla zmiennej (ani nawet utworzyć takiej zmiennej). [Podejście z Haskella/MLa nie działa]
- Na ratunek przybywają function, funcall i apply

Funkcje jako dane

- Chcielibyśmy traktować funkcje na równi z danymi, tj przekazywać je jako parametry, zwracać jako wyniki itp.
- Mamy jednak problem: umiemy wywołać konkretną funkcję, ale nie umiemy tego zrobić dla zmiennej (ani nawet utworzyć takiej zmiennej). [Podejście z Haskella/MLa nie działa]
- Na ratunek przybywają function, funcall i apply

Tworzenie obiektów funkcyjnych przez function

- function jest operatorem specjalnym, który zwraca obiekt funkcyjny na podstawie nazwy funkcji
- Zamiast (function foo) możemy użyć skrótu #'foo

```
CL-USER> (defun pass ())
PASS
CL-USER> pass
EVAL: variable PASS has no value
CL-USER> (function pass)
#<FUNCTION PASS NIL (DECLARE (SYSTEM::IN-DEFUN ...
```

Tworzenie obiektów funkcyjnych przez function

- function jest operatorem specjalnym, który zwraca obiekt funkcyjny na podstawie nazwy funkcji
- Zamiast (function foo) możemy użyć skrótu #'foo

```
CL-USER> (defun pass ())
PASS
CL-USER> pass
EVAL: variable PASS has no value
CL-USER> (function pass)
#<FUNCTION PASS NIL (DECLARE (SYSTEM::IN-DEFUN ...
```

Tworzenie obiektów funkcyjnych przez function

- function jest operatorem specjalnym, który zwraca obiekt funkcyjny na podstawie nazwy funkcji
- Zamiast (function foo) możemy użyć skrótu #'foo

```
CL-USER> (defun pass ())
PASS
CL-USER> pass
EVAL: variable PASS has no value
CL-USER> (function pass)
#<FUNCTION PASS NIL (DECLARE (SYSTEM::IN-DEFUN ...
```

Funkcje wyższego rzędu

Przekazanie funkcji jako argumentu

```
(defun succ (x) (+ x 1))

(mapcar #'succ '(1 2 3 4 5))
```

funcall oraz apply

- funcall i apply służą do wywoływania funkcji zadanych poprzez obiekt funkcyjny
- funcall używamy, gdy znamy arność funkcji
- apply przydaje się, gdy argumenty mamy w postaci listy

Przykłady

```
CL-USER> (funcall #'+ 1 2 3)
6
CL-USER> (apply #'+ '(1 2 3 4 5))
15
```

funcall oraz apply

- funcall i apply służą do wywoływania funkcji zadanych poprzez obiekt funkcyjny
- funcall używamy, gdy znamy arność funkcji
- apply przydaje się, gdy argumenty mamy w postaci listy

Przykłady

```
CL-USER> (funcall #'+ 1 2 3)
6
CL-USER> (apply #'+ '(1 2 3 4 5))
15
```

Plan prezentacji

Funkcje anonimowe i operator lambda

- czasami potrzebujemy "jednorazowej" funkcji, którą np. podamy jako parametr do funkcjonału typu mapcar
- lambda wyrażenie może użyte wszędzie tam, gdzie można użyć zwykłej nazwanej funkcji

Przykłady

```
((lambda (x) (+ x 1)) 1)
(mapcar #'(lambda (x) (* x x)) '(1 2 3))
```

Funkcje anonimowe i operator lambda

- czasami potrzebujemy "jednorazowej" funkcji, którą np. podamy jako parametr do funkcjonału typu mapcar
- lambda wyrażenie może użyte wszędzie tam, gdzie można użyć zwykłej nazwanej funkcji

Przykłady

```
((lambda (x) (+ x 1)) 1)
(mapcar #'(lambda (x) (* x x)) '(1 2 3))
```

Funkcje anonimowe i operator lambda

- czasami potrzebujemy "jednorazowej" funkcji, którą np. podamy jako parametr do funkcjonału typu mapcar
- lambda wyrażenie może użyte wszędzie tam, gdzie można użyć zwykłej nazwanej funkcji

Przykłady

```
((lambda (x) (+ x 1)) 1)
(mapcar #'(lambda (x) (* x x)) '(1 2 3))
```

Plan prezentacji

Tworzenie zmiennych leksykalnych

Mamy dwa sposoby tworzenia zmiennych leksykalnych (lokalnych):

- możemy użyć setf jako jednego z wyrażeń
- możemy też wprowadzić nowe zmienne (bądź zasłonić stare) poprzez let (bądź let*)

Przykłady

```
(defun sum (n)
  (setf count 0)
  (dotimes (x n)
    (incf count x))
  count)
```

Tworzenie zmiennych leksykalnych

Mamy dwa sposoby tworzenia zmiennych leksykalnych (lokalnych):

- możemy użyć setf jako jednego z wyrażeń
- możemy też wprowadzić nowe zmienne (bądź zasłonić stare) poprzez let (bądź let*)

Przykłady

```
(defun sum (n)
  (setf count 0)
  (dotimes (x n)
    (incf count x))
  count)
```

Tworzenie zmiennych leksykalnych

Mamy dwa sposoby tworzenia zmiennych leksykalnych (lokalnych):

- możemy użyć setf jako jednego z wyrażeń
- możemy też wprowadzić nowe zmienne (bądź zasłonić stare) poprzez let (bądź let*)

Przykłady

```
(defun sum (n)
  (setf count 0)
  (dotimes (x n)
    (incf count x))
  count)
```

Tworzenie zmiennych leksykalnych

Mamy dwa sposoby tworzenia zmiennych leksykalnych (lokalnych):

- możemy użyć setf jako jednego z wyrażeń
- możemy też wprowadzić nowe zmienne (bądź zasłonić stare) poprzez let (bądź let*)

Przykłady

```
(defun sum (n)
  (setf count 0)
  (dotimes (x n)
    (incf count x))
  count)
```

Domknięcia

- A co jeśli zwrócimy funkcję, która odwołuje się do zmiennej lokalnej?

Przykład

```
(setf incr
  (let ((count 0))
    #'(lambda () (incf count))))
```

Domknięcia

- A co jeśli zwrócimy funkcję, która odwołuje się do zmiennej lokalnej?

Przykład

```
(setf incr
  (let ((count 0))
    #'(lambda () (incf count))))
```

Plan prezentacji

Zmienne dynamiczne w działaniu

```
(defvar *x* 10)

(defun foo ()
  (format t "X: ~d~%" *x*))

(defun bar ()
  (foo)
  (let ((*x* 20)) (foo))
  (foo))

CL-USER> (bar)
X: 10 X: 20 X: 10
```

Zmienne dynamiczne

- Wszystkie zmienne na poziomie globalnym (najwyższym) są zmiennymi dynamicznymi
- Zmiana wartości zmiennej dynamicznej ma efekt na wszystkie wywołania funkcji w zasięgu tej zmiany - nawet jeśli tamte funkcji były zadeklarowane przed zmianą
- No to czemu nie mówimy na to zmienne globalne?
- Bo to coś lepszego!

Zmienne dynamiczne

- Wszystkie zmienne na poziomie globalnym (najwyższym) są zmiennymi dynamicznymi
- Zmiana wartości zmiennej dynamicznej ma efekt na wszystkie wywołania funkcji w zasięgu tej zmiany - nawet jeśli tamte funkcji były zadeklarowane przed zmianą
- No to czemu nie mówimy na to zmienne globalne?
- Bo to coś lepszego!

Zmienne dynamiczne

- Wszystkie zmienne na poziomie globalnym (najwyższym) są zmiennymi dynamicznymi
- Zmiana wartości zmiennej dynamicznej ma efekt na wszystkie wywołania funkcji w zasięgu tej zmiany - nawet jeśli tamte funkcji były zadeklarowane przed zmianą
- No to czemu nie mówimy na to zmienne globalne?
- Bo to coś lepszego!

Zmienne dynamiczne

- Wszystkie zmienne na poziomie globalnym (najwyższym) są zmiennymi dynamicznymi
- Zmiana wartości zmiennej dynamicznej ma efekt na wszystkie wywołania funkcji w zasięgu tej zmiany - nawet jeśli tamte funkcji były zadeklarowane przed zmianą
- No to czemu nie mówimy na to zmienne globalne?
- Bo to coś lepszego!

Zmienne dynamiczne

- Zmienne globalne są wygodne, ale łatwo nabałaganić (i zapomnieć posprzątać)
- W Common Lisp możemy powiedzieć "zmień tę wartość dla następnego wywołania (i podwywołań będących jego konsekwencją) i *tylko* dla niego"

Zmienne dynamiczne

- Zmienne globalne są wygodne, ale łatwo nabałaganić (i zapomnieć posprzątać)
- W Common Lisp możemy powiedzieć "zmień tę wartość dla następnego wywołania (i podwywołań będących jego konsekwencją) i *tylko* dla niego"

Zmienne dynamiczne w działaniu

```
(defvar *y* 10)

(defun foo ()
  (format t "Y: ~d~%" *y*)
  (incf *y*)
  (format t "Y: ~d~%" *y*))

(defun bar ()
  (foo)
  (let ((*y* 20)) (foo))
  (foo))

CL-USER> (bar)
Y: 10 Y: 11 Y: 20 Y: 21 Y: 11 Y: 12
```

Zmienne dynamiczne

- Jak utworzyć zmienną dynamiczną? defvar lub defparameter
- Jak zmienić wartość zmiennej dynamicznej? setf lub let
- Jak odróżnić w kodzie zmienne dynamiczne od lokalnych? Konwencja nazewnicza - zmienne dynamiczne otaczamy gwiazdkami

Zmienne dynamiczne

- Jak utworzyć zmienną dynamiczną? defvar lub defparameter
- Jak zmienić wartość zmiennej dynamicznej? setf lub let
- Jak odróżnić w kodzie zmienne dynamiczne od lokalnych? Konwencja nazewnicza - zmienne dynamiczne otaczamy gwiazdkami

Zmienne dynamiczne

- Jak utworzyć zmienną dynamiczną? defvar lub defparameter
- Jak zmienić wartość zmiennej dynamicznej? setf lub let
- Jak odróżnić w kodzie zmienne dynamiczne od lokalnych? Konwencja nazewnicza - zmienne dynamiczne otaczamy gwiazdkami

Podsumowanie

(the end)