Makra w języku Scheme

Maciek Godek

09 listopad 2014

Contents

W zeszłe wakacje, sprowokowany przez firra i Edka, opowiedziałem nieco o swoich doświadczeniach z językiem Scheme i programowaniem funkcyjnym. Ponieważ moje objaśnienia spotkały się z dość ciepłym przyjęciem, postanowiłem tym razem opowiedzieć co nieco o higienicznych makrach w języku Scheme.

Struktura tekstu jest następująca. W rozdziale pierwszym omawiam podstawowe wyrażenia z języka Scheme, a następnie w rozdziale drugim opisuję potrzebę zdefiniowania kilku nowych form specjalnych, oraz ogólną użyteczność możliwości definiowania tych form.

W rozdziale trzecim omawiam system makr w stylu Common Lispa (define-macro) i implementuję w nim wprowadzone we wcześniejszym rozdziale przykłady.

W rozdziale czwartym zamierzam omówić system makr higienicznych R5RS (syntax-rules) opracowany przez Kenta Dybviga i zaimplementować w nim wcześniej zaimplementowane formy, wskazując na wady i zalety tego systemu.

W rozdziale piątym chciałym podać różne sposoby obchodzenia ograniczeń systemu syntax-rules, mianowicie (1) ogólniejszy system makr “syntax-case”, (2) zaproponowaną przez Olega Kiselyova implementację maszyny abstrak-

cyjnej CK oraz (3) makra pisane "w stylu przekazywania kontynuacji" (continuation -passing-style)

W rozdziale szóstym chciałbym opowiedzieć o kilku zastosowaniach makr, które w innym przypadku byłyby trudne do uzyskania. W szczególności zamierzam zaprezentować pattern-matcher Wrighta-Shinna.

NINIEJSZY DOKUMENT STANOWI SZKIC ROBOCZY I PROSZĘ O NIEROZPOWSZECHNIANIE GO. NAJNOWSZA WERSJA POWINNA BYĆ DOSTĘPNA W REPOZYTORIUM

https://bitbucket.org/panicz/slayer/

W KATALOGU doc POD NAZWĄ "makra.pdf".

Tekst jest sformatowany zgodnie z wytycznymi trybu ORG dla emacsa. Fragmenty kodu/ewaluacji rozpoczynam ciągiem znaków ": ", ponieważ niektóre programy usuwają białe znaki na początku linii podczas wyświetlania. Symbol "===>" należy czytać jako "ewaluuje się do".

Po ukończeniu wszystkich rozdziałów wrzucę linkę do PDFa. Jeżeli ktoś byłby zainteresowany, mogę też gdzieś wrzucić wygenerowanego PDFa z tym, co napisałem do tej pory.

Będę wdzięczny za wszelkie pytania, uwagi i komentarze

1 JĘZYK SCHEME – KRÓTKIE OMÓWIENIE

1.1 EWALUACJA WYRAŻEŃ

Do głównych zalet Scheme'a należy to, że łączy on w sobie prostotę i semantyczną siłę rachunku lambda z prostotą i syntaktyczną siłą lispa. Dzięki tym dwóm cechom można z łatwością rozszerzać język o nowe wyrażenia, w naturalny sposób komponujące się ze starymi, a także zmieniać sposób interpretacji istniejących wyrażeń.

Składnia języka Scheme (podobnie jak wszystkich innych lispów) jest bardzo prosta i stanowi "w pełni onawiasowaną notację polską". Programy w lispie są złożone z nawiasów (przy czym każdy nawias otwierający musi mieć odpowiadający nawias zamykający), symboli (czyli dowolnych ciągów znaków oprócz nawiasów oraz znaków " ", ",", "'", ".", "#", podwójnych cudzysłowiów i białych znaków. Dodatkowy warunek jest taki, że – aby ciąg był uznany za symbol – nie może być interpretowany jako liczba) oraz liczb. (Standard Scheme'a definiuje również inne podstawowe jednostki leksykalne, m.in. stringi, znaki czy wektory, jednak ze względu na prostotę pominiemy sobie tutaj tę kwestię)

Przykładowe wyrażenie arytmetyczne w Schemie mogłoby mieć zatem postać:

```
(+ (* 3 4) (/ 20 5))
```

Reguła ewaluacji jest prosta: oczekujemy, że wartością pierwszego elementu listy będzie funkcja (np. dodawanie, mnożenie itd.), zaś aby poznać wartość całego wyrażenia, musimy zastosować funkcję do wartości argumentów. Jeżeli zatem symbole +, * i / są związane z konwencjonalnymi działaniami arytmetycznymi, powyższe wyrażenie możemy zgodnie z tą regułą najpierw przekształcić do

```
(+ 12 4)
```

a następnie do

```
16
```

1.2 FORMY SPECJALNE

1.2.1 FORMA "quote"

Ciekawą własnością lispa jest tzw. homoikoniczność: kod źródłowy programów stanowi listę symboli, liczb i ewentualnie innych list, zaś listy, symbole i liczby są typami danych, na których programy lispowe mogą operować. Aby powstrzymać ewaluator przed interpretowaniem symbolu, trzeba użyć operatora "quote". Wartością wyrażenia

```
(quote x)
```

będzie

```
x
```

Operator "quote" działa inaczej, niż użyte powyżej operatory arytmetyczne. Operatory arytmetyczne (domyślnie) stanowią zwykłe funkcje, zaś "quote" jest formą specjalną, która charakteryzuje się tym, że nie ewaluuje swoich argumentów. Lisp traktuje zapis 'x równoważnie z zapisem (quote x).

Aby uzyskać listę symboli (a b c), moglibyśmy użyć funkcji "list":

```
(list 'a 'b 'c)
===>
(a b c)
```

Operator "quote" działa jednak nie tylko na symbole, ale również na całe wyrażenia, więc powyższy wynik moglibyśmy uzyskać również pisząc po prostu

```
'(a b c)
===>
(a b c)
```

1.2.2 FORMA "lambda"

Oprócz operatora "quote", w języku Scheme występuje 5 innych podstawowych form specjalnych. Jedną z nich jest forma "lambda", która ma następującą postać:

```
(lambda (argumenty ...) ciało + ...)
```

Forma "lambda" jest bardzo potężna i w zasadzie wzięta sama w sobie stanowi kompletny język programowania, pozwalający na wyrażenie instrukcji warunkowych "if-then-else" oraz liczb naturalnych. Osobom zainteresowanym tym zagadnieniem polecam kurs programowania funkcyjnego autorstwa Johna Harrisona:

http://www.cl.cam.ac.uk/teaching/Lectures/funprog-jrh-1996/index.html

Mówiąc najkrócej, wyrażenie lambda tworzy nową funkcję. Najprostszy przykład mógłby wyglądać następująco:

```
(lambda (x y)(+ x (* x y)))
```

W wyniku dostajemy funkcję dwuargumentową, która dodaje swój pierwszy argument do iloczynu obu argumentów. Gdybyśmy chcieli użyć tej funkcji, moglibyśmy napisać:

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

Aplikację wyrażenia lambda można pojmować w taki sposób, że zastępujemy całe wyrażenie lambda jego ciałem, w którym symbole użyte jako argumenty zastępujemy wartościami tych argumentów. W powyższym przypadku byłoby to:

```
(+ 3 (* 3 4))
```

Choć może to wyglądać niepozornie, mając do dyspozycji tę regułę, możemy wyrazić dowolną funkcję rekurencyjną (za sprawą tzw. kombinatorów punktu stałego, w szczególności – kombinatora Y).

Dodatkowo Scheme pozwala nam zdefiniować funkcję mogącą przyjmować dowolnie wiele argumentów, jeżeli zamiast listy argumentów podamy pojedynczy symbol. Wówczas z owym symbolem zostaje w ciele funkcji związana lista wszystkich przekazanych argumentów:

```
((lambda args args) 1 2 3)
===>
(1 2 3)
```

1.2.3 FORMA "define"

Kolejną formą specjalną jest forma "define", która pozwala nam nazywać różne obiekty. Na przykład moglibyśmy zdefiniować podnoszenie do kwadratu:

```
(define square (lambda (x) (* x x)))
```

Operacja ta wiąże funkcję, która mnoży swój jedyny argument przez siebie, z symbolem "square". Zapis

```
(square 5)
```

jest równoważny zapisowi

```
((lambda (x) (* x x)) 5)
```

co na mocy wcześniejszej reguły można przekształcić do

```
(* 5 5)
```

Formy "define" można również użyć do wprowadzenia definicji w zasięgu wyrażenia "lambda", np.

```
(define x 5) ; zasięg zewnętrzny
x
===> 5

((lambda () (define x 20) x))
====> 20
```

```
x
===> 5 ; wartość w zasięgu zewnętrznym nie uległa zmianie
```

Warto wiedzieć, że formalnie nie uznaje się definicji za wyrażenia, i na przykład zapis

```
(lambda () (define x 20))
```

jest błędny, ponieważ ciało wyrażenia lambda musi zawierać przynajmniej jedno wyrażenie.

1.2.4 FORMA "if"

Kolejną formą specjalną, która zasługuje na uwagę, jest forma warunkowa "if", o następującej składni:

```
(if warunek wyrażenie-gdy-warunek-spełniony)
```

lub

```
(if warunek
    wyrażenie-gdy-warunek-spełniony
    wyrażenie-w-przeciwnym-razie)
```

Forma "if" działa w taki sposób, że najpierw dokonuje ewaluacji warunku, i jeżeli wartość warunku jest różna od wartości fałszu (zapisywanej w Schemie jako "#f" albo – wg. najnowszego standardu – jako "#false"), to wartością całego wyrażenia staje się wartość **wyrażenia-gdy-warunek-spełniony**, zaś w przeciwnym razie wartość wyrażenia albo jest nieokreślona (dla wariantu pierwszego), albo jest wartością **wyrażenia-w-przeciwnym-razie**.

Forma "if" pozwala na definiowanie złożonych funkcji. Klasycznym przykładem jest funkcja "silnia":

```
(define ! (lambda (n) (if (= n 0) 1 (* n (! (- n 1))))))
```

Zgodnie z regułami ewaluacji, wyrażenie

```
(! 5)
```

można zinterpretować następująco:

```
((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 5)
===>
(if (= 5 0) 1 (* 5 (! (- 5 1))))
===>
(* 5 (! 4))
===>
(* 5 ((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 4))
===>
(* 5 (if (= 4 0) 1 (* 4 (! (- 4 1)))))
===>
(* 5 (* 4 (! 3)))
===>
(* 5 (* 4 ((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 3)))
===>
(* 5 (* 4 (if (= 3 0) 1 (* 3 (! (- 3 1))))))
===>
(* 5 (* 4 (* 3 (! 2))))
===>
(* 5 (* 4 (* 3 ((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 2))))
===>
(* 5 (* 4 (* 3 (if (= 2 0) 1 (* 2 (! (- 2 1)))))))
===>
(* 5 (* 4 (* 3 (* 2 (! 1)))))
===>
(* 5 (* 4 (* 3 (* 2 ((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 1)))))
===>
(* 5 (* 4 (* 3 (* 2 (if (= 1 0) 1 (* 1 (! (- 1 1))))))))
===>
(* 5 (* 4 (* 3 (* 2 (* 1 (! 0))))))
===>
(* 5 (* 4 (* 3 (* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (! (- n 1))))) 0))))))
===>
(* 5 (* 4 (* 3 (* 2 (* 1 (if (= 0 0) 1 (* 0 (! (- 0 1)))))))))
===>
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
===> ... ===>
120
```

1.2.5 FORMA "set!"

Forma "set!" powoduje przypisanie nowej wartości do istniejącej (tzn. związanej) zmiennej. Zmienna może być związana albo przy pomocy formy "define", albo jako parametr formalny w formie "lambda"

```
(define x 5)
x
===> 5

((lambda (x) (set! x (+ x 1)) x) 7) ; zmiana wartości zmiennej lokalnej
===> 8

x
===> 5

((lambda () (set! x (+ x 1)))) ; zmiana wartości zmiennej zewnętrznej
x
===> 6
```

1.2.6 FORMA "begin"

Formy "begin" używamy do grupowania wyrażeń, np. wewnątrz którejś gałęzi formy "if". Warto jednak zwrócić uwagę, że zapis

```
(begin definicje/wyrażenia ...)
```

nie jest równoważny zapisowi

```
((lambda () definicje/wyrażenia ...))
```

ponieważ forma "begin" nie wprowadza nowego zasięgu. W szczególności forma "begin" może zawierać wyłącznie definicje, i np. zapis

```
(begin (define symbol-1 wartość-1) (define symbol-2 wartość-2))
```

jest równoważny zapisowi:

```
(define symbol-1 wartość-1)
(define symbol-2 wartość-2)
```

1.3 FUNKCJE WBUDOWANE

Lista funkcji wbudowanych nie jest szczególnie istotna. W dotychczasowych przykładach używałem funkcji +,*,-,/,= oraz list.

1.3.1 FUNKCJA "apply"

Jest jednak kilka funkcji, które zasługują na szczególną uwagę. Pierwszą z nich jest "apply". Zapis

```
(funkcja argumenty ...)
```

jest równoważny zapisowi

```
(apply funkcja (list argumenty ...)).
```

Na przykład zamiast

```
(+ 1 2)
```

można napisać

```
(apply + '(1 2))
```

Ponadto funkcja "apply" może być użyta z dodatkowymi argumentami pomiędzy argumentem funkcyjnym a listą. Wówczas elementy pomiędzy pierwszym a ostatnim argumentem zostaną dopisane na początku ostatniego argumentu. W związku z tym następujące wywołania sa równowazne:

```
(+ 1 2 3)
(apply + '(1 2 3))
(apply + 1 '(2 3))
(apply + 1 2 '(3))
(apply + 1 2 3 '())
```

1.3.2 FUNKCJE "values" i "call-with-values"

Pewną specyficzną cechą Scheme'a jest to, że zdefiniowane w nim funkcje mogą zwracać więcej niż jedną wartość. Do zwracania wielu wartości służy funkcja "values", która pobiera dowolnie wiele argumentów:

```
(values 1 2 3)
===> 1
===> 2
===> 3
```

Do przechwycenia tych wartości służy funkcja call-with-values, która pobiera dwie funkcje – producenta (0-argumentowego) i konsumenta (n-argumentowego), np.

```
(call-with-values (lambda()(values 1 2 3)) (lambda(a b c)(+ a b c)))
```

1.3.3 FUNKCJE "cons", "car" i "cdr"

Do tej pory w kilku przykładach użyłem dowolnie-wielo-argumentowej funkcji "list", której wartością jest lista podanych do niej argumentów. Warto jednak wiedzieć nieco więcej o architektonice list. Po pierwsze, istnieje w Schemie specjanly symbol na oznaczenie listy pustej, a jest on zapisywany tak:

```
'()
```

Po drugie, listy skonstruowane są z par. Do tworzenia par służy funkcja cons:

```
(cons 'a 'b)
===> (a . b)
```

Po trzecie, funkcje car i cdr służą do pobrania odpowiednio pierwszego i drugiego elementu pary:

```
(car (cons 'a 'b))
===> a

(cdr (cons 'a 'b))
===> b
```

Elementy (car x) i (cdr x) będziemy od tej pory nazywać odpowiednio głową i ogonem listy.

Po czwarte, zapis

```
(a . (b . (c . ())))
```

jest równoważny zapisowi

```
(a b c)
```

zaś zapis

```
(a . (b . (c . d)))
```

zapisowi

```
(a b c . d)
```

dla dowolnych dopuszczalnych wartości a, b, c, d). Jeżeli drugi element ostatniej pary listy nie jest listą pustą, to taką listę nazywamy "listą kropkowaną" ("dotted list").

Stąd też (list 1 2 3) jest równoważne (cons 1 (cons 2 (cons 3 '()))).

1.3.4 PREDYKATY PORÓWNUJĄCE: "=", "<", "<=", ">", ">=", "eq?"

Oprócz użytej w jednym z poprzednich przykładów funkcji "=", sprawdzającej równość numeryczną, Scheme definiuje kilka innych użytecznych predykatów do operowania na liczbach, m.in. "<" (mniejszy od), który zwraca prawdę wtw jej argumenty (liczbowe) stanowią ciąg rosnący, np.

```
(< 1 3 7)
===> #t ; gdzie #t to specjalna wartość odpowiadająca prawdzie logicznej

(< 3 7 1)
===> #f
```

Pozostałe funkcje porównujące ("<=", ">", ">=") działają tak, jak można się po nich spodziewać. Na uwagę zasługuje predykat eq?, który sprawdza, czy dwa obiekty są tym samym obiektem – na przykład, czy dwa symbole są tym samym symbolem. W przypadku list działanie tej funkcji może być zaskakujące:

```
(define a '(1 2 3))
(define b '(1 2 3))
(eq? a b)
===> #f
```

Wynika to stąd, że listy a i b stanowią osobne obiekty w pamięci. (W pewnych przypadkach niektóre kompilatory mogłyby zoptymalizować powyższy kod w taki sposób, że a i b byłyby tym samym obiektem, i wówczas użycie funkcji eq? może zwrócić #t).

Lista pusta jest zawsze tożsama ze sobą, tj.

```
(eq? '() '())
===> #t
```

1.3.5 UWAGI O FUNKCJI "call-with-current-continuation" ("call/cc")

Ostatnią osobliwą funkcją jest call-with-current-continuation, zazwyczaj skracana jako call/cc. Z pewnych względów omówię ją dopiero później, gdy będę miał możliwość zademonstrowania jej działania na konkretnym przykładzie.

2 POTRZEBA NOWYCH FORM SPECJALNYCH

2.1 UWAGI WSTĘPNE

Dotychczas omówiłem zaledwie 6 podstawowych form specjalnych potrzebnych do tego, żeby budować złożone programy. Brakuje tutaj jednak wielu rodzajów wyrażeń, które znamy z innych języków programowania, takich jak możliwość wprowadzania zmiennych pomocniczych w ciele funkcji, albo logicznych operatorów koniunkcji i alternatywy (negację moglibyśmy bowiem zdefiniować jako funkcję: (define not (lambda (x) (if x #f #t)))).

W tym celu musielibyśmy dysponować jeszcze przynajmniej jedną formą specjalną, pozwalającą nam na definiowanie nowych form specjalnych. Nie powinno to być trudne, ponieważ – jak sobie powiedzieliśmy na początku – programy w lispie są listami symboli, więc wystarczyłoby zmodyfikować reguły ewaluacji w taki sposób, żeby wszystkie nowe formy specjalne były przed wykonaniem przekształcane do postaci zawierającej wyłącznie funkcje i pierwotne formy specjalne.

Taka procedura przekształcająca jakieś wyrażenie (listę) w inne wyrażenie (listę) nazywana jest **makrem**. Zanim poznamy szczegóły dotyczące tego, jak można definiować makra, spróbujemy się zastanowić, jak byśmy mogli chcieć używać danego wyrażenia, oraz jak mogłaby wyglądać nasza forma wynikowa. Jedyne ograniczenie jest takie, że nasza składnia musi być zgodna ze składnią lispa.

UWAGA! WSZYSTKIE MAKRA PRZEDSTAWIONE W TYM ROZDZIALE SĄ CZĘŚCIĄ JĘZYKA SCHEME I NIE TRZEBA ICH DEFINIOWAĆ.

GDYBY CZYTELNIK ZECHCIAŁ ZDEFINIOWAĆ SWOJE WERSJE PONIŻSZYCH FORM SPECJALNYCH, ZALECAM NADANIE IM INNYCH NAZW.

2.2 ZMIENNE LOKALNE – FORMA "let"

Czasem wygodnie jest przechować daną wartość w zmiennej pomocniczej.

```
(let ((x^2 (* x x))
      (y^2 (* y y)))
  (/ (- x^2 y^2) (+ x^2 y^2)))
```

W tym wypadku użyliśmy zmiennych lokalnych o nazwach "x^2" i "y^2" do zapamiętania wyniku obliczenia działań (* x x) i (* y y) (przy założeniu, że zmienne "x" i "y" są związane w kontekście wyrażenia). W przeciwnym razie musielibyśmy dwukrotnie wyliczyć wynik mnożenia, zaś główne wyrażenie stałoby się rozwleklejsze:

```
(/ (- (* x x) (* y y)) (+ (* x x) (* y y)))
```

Zauważmy, że pożądany efekt możemy uzyskać przy pomocy formy "lambda":

```
((lambda(x^2 y^2)(/ (- x^2 y^2) (+ x^2 y^2))) x y)
```

Dlatego chcielibyśmy, żeby formy postaci

```
(let ((nazwa wartość) ...) ciało + ...)
```

były przed ewaluacją transformowane do postaci

```
((lambda (nazwa ...) ciało + ...) wartość ...)
```

2.3 LOGICZNE SPÓJNIKI "and" i "or"

Moglibyśmy zdefiniować operatory logiczne "and" i "or" jako funkcje:

```
(define and2 (lambda (p q) (if p q #f)))
(define or2 (lambda (p q) (if p p q)))
```

Z taką definicją są jest jednak pewien problem: mianowicie, nasz operator ewaluuje wsztstkie swoje argumenty, choć teoretycznie jeżeli pierwszy argument ma wartość fałszu, to moglibyśmy zaniechać ewaluacji kolejnych argumentów (taka metoda ewaluacji nosi nazwę "short-circit evaluation" albo "McCarthy evaluation", i implementują je właściwie wszystkie współcześnie używane języki programowania).

Moglibyśmy zatem życzyć sobie, żeby kod

```
(or p q)
```

był przed wykonaniem transformowany do postaci

```
(if p p q)
```

jednak wówczas problemem byłoby to, że wyrażenie "p" – o ile jego wartością nie byłby fałsz logiczny – byłoby ewaluowane dwukrotnie. Moglibyśmy temu zapobiec transformując naszą alternatywę do postaci

```
(if p #t q)
```

albo, pragnąc zachować wartość wyniku ewaluacji, przekształcić ją następująco:

```
(let ((T p)) (if T T q))
```

gdzie T jest symbolem niewystępującym w q. Moglibyśmy też uogólnić operator "or" w taki sposób, żeby mógł przyjmować dowolnie wiele argumentów, i żeby wyrażenia postaci

```
(or p q r ...)
```

były transformowane do postaci

```
(let ((T p)) (if T T (or q r ...)))
```

Analogicznie moglibyśmy oczekiwać, żeby wyrażenia postaci

```
(and p q)
```

były przed ewaluacją przekształcane do postaci

```
(if p q #f)
```

i ogólniej, interpretować wyrażenia

```
(and p q r ...)
```

jako

```
(if p q (and r ...))
```

2.4 FORMA SPECJALNA "cond"

Większość popularnych języków programowania posiada specjalną konstrukcję dla wyboru sekwencyjnego, np.

```
if warunek { działania ... }
else if inny_warunek { inne_działania ... }
...
else { ostateczne_działania ... }
```

W Schemie moglibyśmy użyć kaskady instrukcji "if":

```
(if warunek
    (begin działania ...)
    (if inny_warunek
        (begin inne_działania ...)
        ... (begin ostateczne_działania ...) ...))
```

Taki kaskadowy kod osiąga jednak dość szybko duży poziom zagłębienia i staje się nieprzyjemny do czytania. Definiując Lispa, John McCarthy wprowadził formę "cond" o następującej składni:

```
(cond (warunek działania ...)
      (inny_warunek inne_działania ...)
      ...
      (else ostateczne_działania ...))
```

"else" jest w kontekście formy "cond" symbolem specjalnym. Powyższy kod mógłby być przekształcony do postaci rekurencyjnej:

```
(if warunek
    (begin działania ...)
    (cond (inny_warunek inne_działania ...)
          ...
         (else ostateczne_działania ...)))
```

zaś formę z jednym warunkiem (przypadek brzegowy), np.

```
(cond (ostatni_warunek środki_ostateczne ...))
```

moglibyśmy interpretować równoważnie z

```
(if ostatni_warunek (begin środki_ostateczne ...))
```

2.5 PĘTLA "while" oraz funkcja "call/cc"

W sekcji objaśniającej działanie formy specjalnej "if" podałem przykład funkcji obliczającej silnię. Podany przeze mnie kod był czysto funkcyjny, dzięki czemu można było dokonać analizy jego działania za pomocą prostych podstawień.

Osoby przyzwyzajone do stylu imperatywnego mogłyby zapewne chciec zdefiniować silnię następująco:

```
(define !!
  (lambda (n)
    (let ((result 1))
      (while (> n 0)
        (set! result (* result n))
        (set! n (- n 1)))
      result)))
```

Użyliśmy tutaj pętli "while", której zasada działania powinna być znana większości programistów. Moglibyśmy ją wyrazić przy pomocy wprowadzonych dotychczas środków. Mianowicie chcielibyśmy, żeby kod

```
(while warunek działania ...)
```

był transformowany do postaci

```
(begin
  (define LOOP
    (lambda ()
      (if warunek
        (begin działania ... (LOOP)))))
  (LOOP))
```

W tym kontekście warto omówić działanie "call/cc". Otóż jest to specjalna procedura, która pobiera jeden argument, będący lambda-wyrażeniem od jednego argumentu. Tym argumentem jest tzw. kontunuacja – procedura (mogąca przyjmować dowolnie wiele argumentów), której wywołanie spowoduje przerwanie wykonywania bieżącej funkcji i przekazanie sterowania do wyrażenia następującego bezpośrednio po wywołaniu call/cc. Owo wyjaśnienie może brzmieć niezrozumiale, dlatego warto posłużyć się przykładem:

```
(call/cc
  (lambda (return)
    (display "a")
    (return)
    (display "b")))
(display "c")
```

Powyższe wywołanie wypisze ciąg znaków "ac" (wypisanie "b" zostanie pominięte). Procedurę call/cc możemy zastosować do zaimplementowania instrukcji "break", znanej z takich języków jak C czy Python. Na przykład, moglibyśmy transformować powyższą pętlę "while" do postaci

```
(call/cc
  (lambda (break)
    (define LOOP
      (lambda ()
        (if warunek
          (begin działania ... (LOOP)))))
    (LOOP)))
```

Kontynuacje (czyli pojęcie, pod które podpadają takie konstrukcje z języków imperatywnych, jak "exit" w kontekście programu, "return", "yield" czy "goto" w kontekście funkcji, "continue" i "break" w kontekście pętli oraz "throw" i "catch" w kontekście wyjątków) mają w języku Scheme status obiektów, które mogą być przekazywane, zachowywane i wielokrotnie wywoływane. Spora ogólność kontynuacji była przedmiotem krytyki z powodu pewnych trudności implementacji, dużego zużycia zasobów i niskiej wydajności, co skłoniło teoretyków języków programowania do wprowadzenia różnych wariantów kontynuacji o ograniczonym zasięgu. Osoby zaintersowane szczegółową dyskusją odsyłam do strony

http://okmij.org/ftp/continuations/against-callcc.html

Tymczasem dodanie do pętli "while" instrukcji "continue" pozostawiam jako ćwieczenie dla ciekawskich.

2.6 INNE FORMY SPECJALNE

Podana powyżej lista form specjalnych jest podzbiorem form specjalnych definiowanych przez standard Scheme'a. Oprócz nich definiuje się również m.in. formy "let*", "letrec" czy "do". Moim celem nie jest jednak omówienie wszystkich form obecnych w języku Scheme, ale zarysowanie, w jaki sposób

obecność makr zwiększa siłę wyrazu języka, oraz zademonstrowanie różnych podejść do tego, w jaki sposób można owe formy specjalne definiować.

Pod koniec tekstu podam kilka mniej trywialnych zastosowań dla makr, a tymczasem przejdę do omówienia możliwych (praktycznych) implementacji.

3 MAKRA PROCEDURALNE (define-macro)

Niektóre wersje Scheme'a obsługują proste markra proceduralne w stylu Common Lispa. Pomysł jest taki, że traktujemy wyrażenie wejściowe jako listę, i piszemy procedurę, która przekształca tę listę do innej listy. Do zdefiniowania takiej formy specjalnej używamy specjalnej formy define-macro o następującej postaci:

```
(define-macro (nazwa-makra argumenty ...)
  ciało + ...)
```

gdzie "ciało + ..." powinno zwracać wyrażenie, które ma zostać ewaluowane (lub dalej przekształcone). Przykłady powinny nieco tę kwestię rozjaśnić.

3.1 DEFINICJA FORMY "let"

Gwoli przypomnienia, powiedzieliśmy sobie w poprzednim rozdziale, że forma "let" powinna być przed ewaluacją transformowana z postaci

```
(let ((nazwa wartość) ...) ciało + ...)
```

do postaci

```
((lambda (nazwa ...) ciało + ...) wartość ...)
```

3.1.1 PRELIMINARIA – ZWYKŁA FUNKCJA TRANSFORMUJĄCA LISTĘ

Zanim przystąpimy do zdefiniowania makra, spróbujmy napisać zwykłą funkcję, która przetransformuje listę

```
(let ((nazwa-1 wartość-1) (nazwa-2 wartość-2)) ciało-1 ciało-2)
```

do postaci

```
((lambda (nazwa-1 nazwa-2) ciało-1 ciało-2) wartość-1 wartość-2)
```

Nazwijmy pierwszą z powyższych list (tę zawierającą symbol "let") X, a drugę (tę zaczynającą się od listy zaczynającej się od symbolu "lambda") – Y.

- DEKONSTRUKCJA LISTY WEJŚCIOWEJ

  Zauważmy, że:

```
(car X) = let
(cdr X) = (((nazwa-1 wartość-1) (nazwa-2 wartość-2)) ciało-1 ciało-2)
(car (cdr X)) = ((nazwa-1 wartość-1) (nazwa-2 wartość-2))
(cdr (cdr X)) = (ciało-1 ciało-2)
```

  Dla ogólnego przypadku możemy zatem zdefiniować sobie:

```
(define macro-body (lambda (macro) (cdr macro)))
(define let-bindings (lambda (macro-body) (car macro-body)))
(define let-body (lambda (macro-body) (cdr macro-body)))
```

  Jest jasne, że lista wiązań zwracana przez "let-bindings" ma postać

```
((nazwa wartość) ...)
```

  Chcielibyśmy mieć możliwość oddzielenia od siebie nazw i wartości, tzn. dysponować funkcjami

```
(define bindings-names (lambda (let-bindings) ??))
(define bindings-values (lambda (let-bindings) ??))
```

  o takich własnościach, że

```
(bindings-names '((nazwa-1 wartość-1) (nazwa-2 wartość-2)))
```

  da w wyniku

```
(nazwa-1 nazwa-2)
```

  zaś

```
(bindings-values '((nazwa-1 wartość-1) (nazwa-2 wartość-2)))
```

zwróci

```
(wartość-1 wartość-2)
```

Wiemy też, że skoro każde pojedyncze wiązanie ma postać

```
(nazwa wartość)
```

to możemy zdefiniować

```
(define binding-name (lambda (binding) (car binding)))
(define binding-value (lambda (binding) (car (cdr binding))))
```

Funkcje "bindings-names" i "bindings-values" możemy zdefiniować rekurencyjnie. Jeżeli lista wiązań jest pusta, to oczywiście listy nazw [wartości] też będą puste. W przeciwnym razie zwracamy parę, której głową jest nazwa [wartość] z głowy listy, a ogonem – lista pozostałych nazw [wartości]:

```
(define bindings-names
  (lambda (let-bindings)
    (if (eq? let-bindings '())
      '()
      (cons (binding-name (car let-bindings))
            (bindings-names (cdr let-bindings))))))

(define bindings-values
  (lambda (let-bindings)
    (if (eq? let-bindings '())
      '()
      (cons (binding-value (car let-bindings))
            (bindings-values (cdr let-bindings))))))
```

- KONSTRUKCJA FORMY WYJŚCIOWEJ

Teraz, kiedy jesteśmy już w stanie wyekstrahować poszczególne elementy – listy nazw i wartości oraz wrażenia z ciała funkcji – pozostaje nam do rozwiązania jeszcze jeden problem. Pożądana przez nas lista wyjściowa ma mieć postać

```
((lambda (nazwa-1 nazwa-2) ciało-1 ciało-2) wartość-1 wartość-2)
```

Z tego powodu nie możemy napisać

```
(list (list 'lambda <lista-nazw> <ciało-funkcji>) <lista-wartości>)
```

ponieważ wówczas lista wynikowa miałaby postać

```
((lambda (nazwa-1 nazwa-2) (ciało-1 ciało-2)) (wartość-1 wartość-2))
```

Możemy jednak użyć funkcji "apply" w połączeniu z funkcją "list", żeby "spłaszczyć" ostatnią listę:

```
(apply list (apply list 'lambda <lista-nazw> <ciało-funkcji>) <lista-wartości>)
```

Dzięki temu mamy wszystko, co niezbędne do tego, żeby przetransformować listę X do postaci Y:

```
(define transform-let
  (lambda (macro)
    (apply list
      (apply list
        'lambda
        (bindings-names (let-bindings (macro-body macro)))
        (let-body (macro-body macro)))
      (bindings-values (let-bindings (macro-body macro))))))
```

3.1.2 DEFINICJA MAKRA

Zaopatrzeni w tę wiedzę, możemy jej użyć do zdefiniowania makra:

```
(define-macro (let . macro-body)
  (apply list
    (apply list
      'lambda
      (bindings-names (let-bindings macro-body))
      (let-body macro-body))
    (bindings-values (let-bindings macro-body))))
```

Nagłówek funkcji stanowi listę kropkowaną – to dlatego, że nasze makro może przyjąć dowolnie wiele argumentów. Makro różni się od funkcji transform-let tym, że w funkcji musieliśmy pominąć samą nazwę makra ("let"), natomiast procesor makr robi to za nas.

Widać też, że definiując makro, nie używamy formy lambda. Przy okazji warto dodać, że funkcje można definiować analogicznie – zamiast pisać

```
(define funkcja (lambda (argumenty ...) ciało + ...))
```

możemy również napisać

```
(define (funkcja argumenty ...) ciało + ...)
```

i wówczas przed ewaluacją ta druga forma zostanie przetransformowana do tej pierwszej.

Definicja

```
(define my-list (lambda x x))
```

jest przy tym równoważna definicji

```
(define (my-list . x) x)
```

Oczywiście, moglibyśmy również zdefiniować makro przyjmujące stałą ilość argumentów, nie używając listy kropkowanej:

```
(define-macro (two-argument-macro a b)
  (list 'quote (list 'argument-a: a 'argument-b: b)))
```

- SPECJALNA SKŁADNIA DO BUDOWANIA LIST – FORMA "quasiquote"

  Trzeba przyznać, że postać otrzymanej przez nas funkcji przekształcającej formę "let" do wyrażenia "lambda" jest skomplikowana i trudna do czytania. Z tego powodu hakerzy lispa wymyślili specjalną notację ułatwiającą budowanie złożonych struktur listowych.

  Po pierwsze, przyjmijmy, że – tak jak zapisy "(quote X)" i "'X" są równoważne, zapisom

  ```
  (quasiquote X)
  (unquote X)
  (unquote-splicing X)
  ```

odpowiadają skróty

```
`X
,X
,@X
```

"quasiquote" jest zaś pewną formą specjalną (którą można zdefiniować np. przy pomocy "define-macro"), w obrębie której symbole "unquote" i "unquote-splicing" mają dodatkowo specjalne znaczenie. Na przykład, zapis

```
`(z ,z ,@z)
```

jest równoważny zapisowi

```
(apply list 'z z z)
```

i jeśli np. wartością symbolu "z" będzie lista (1 2 3), to wyrażenie otrzyma wartość:

```
(let ((z '(1 2 3)))
  `(z ,z ,@z))
===>
(z (1 2 3) 1 2 3)
```

Element "unquote-splicing" może się pojawić na dowolnej pozycji, w związku z czym wartością wyrażenia

```
(let ((z '(1 2 3)))
  `(,z ,@z z))
```

będzie

```
((1 2 3) 1 2 3 z)
```

Takiego efektu nie moglibyśmy uzyskać przy pomocy funkcji "apply" (tak naprawdę pierwsze użycie operatora "quasiquote" też tego nie robi. W praktyce do łączenia list służy funkcja "append", której definiowania postanowiłem uniknąć, żeby nieco uprościć swój i tak już chyba nadmiernie skomplikowany wywód)

Dysponując makrem "quasiquote", możemy zdefiniować nasze makro następująco:

```
(define-macro (let . macro-body)
  `((lambda ,(bindings-names (let-bindings macro-body))
      ,@(let-body macro-body))
    ,@(bindings-values (let-bindings macro-body))))
```

Zapis ten jest już dużo krótszy i czytelniejszy od poprzedniego, choć do zrozumienia tego makra wymagana jest znajomość zdefiniowanych wcześniej funkcji "let-bindings", "let-body", "binding-names" i "binding-values" – analiza kodu, choć stosunkowo prosta, jest mimo wszystko nietrywialna.

3.2 DEFINICJA SPÓJNIKÓW LOGICZNYCH "or" i "and"

Po tym, co już zostało powiedziane, definicja spójników logicznych powinna być stosunkowo prosta. Zaczniemy od definicji koniunkcji:

```
(define-macro (and first . rest)
  (if (eq? rest '())
    first
    `(if ,first (and ,@rest) #f)))
```

Jedyna nowość, jaka się tu pojawia, dotyczy tego, że lista argumentów składa się z dwóch symboli, przy czym pierwszy (first) zostaje związany z pojedynczym elementem, zaś drugi, pojawiający się po kropce (rest) – z listą pozostałych argumentów.

W analogiczny sposób możemy definiować funkcje, przy czym zapis

```
(define (funkcja arg-1 ... arg-n . args) ciało + ...)
```

jest równoważny zapisowi

```
(define funkcja (lambda (arg1 ... arg-n . args) ciało + ...))
```

Spójnik "or" mógłby być zdefiniowany równie prosto, gdybyśmy nie chcieli zachować wartości prawdziwego wyrażenia:

```
(define-macro (or first . rest)
  (if (eq? rest '())
    first
    `(if ,first #t (or ,@rest))))
```

Jeżeli jednak chcemy tę wartość zachować, to – zgodnie ze wcześniejszymi rozważaniami – musimy wprowadzić nowy identyfikator, tzn. chcemy, żeby formy o postaci

```
(or p q r ...)
```

były transformowane do postaci

```
(let ((T p)) (if T T (or q r ...)))
```

dla T niewystępującego w q, r, ...

Jednym ze sposobów byłoby ustalenie pewnego T i zabronienie używania go w formie "or". Takie rozwiązanie byłoby jednak dość nieeleganckie. Innym sposobem byłoby przeanalizowanie zawartości form q, r, ... pod kątem użytych symboli. Klasycznym rozwiązaniem jest wygenerowanie identyfikatora przy pomocy funkcji "gensym".

```
(define-macro (or first . rest)
  (if (eq? rest '())
    first
    (let ((T (gensym)))
      `(let ((,T ,first))
         (if ,T ,T (or ,@rest))))))
```

Wprawdzie operator "gensym" nie gwarantuje, że wygenerowany symbol nie był dotychczas użyty w kodzie źródłowym, ale gwarantuje, że każde kolejne jego wywołanie wygeneruje nowy symbol, zaś wygenerowane przezeń symbole są na tyle dziwne, że prawdopodobieństwo użycia ich w praktycznym kodzie jest nikłe.

3.3 DEFINICJA FORMY "cond"

Przypomnijmy, że (pomijając warunki brzegowe) chcieliśmy transformować kod

```
(cond (warunek działania ...)
      (inny_warunek inne_działania ...)
      ...
      (else ostateczne_działania ...))
```

do postaci

```
(if warunek
    (begin działania ...)
    (cond (inny_warunek inne_działania ...)
          ...
          (else ostateczne_działania ...)))
```

Każda pojedyncza gałąź makra “cond” ma postać

```
(warunek działania ...)
```

Możemy zatem zdefiniować funkcje destrukturyzujące warunki, tak jak robiliśmy to przy definicji makra “let”:

```
(define (cond-condition cond-branch) (car cond-branch))
(define (cond-actions cond-branch) (cdr cond-branch))
```

Dzięki temu możemy łatwo napisać definicję naszej formy “cond”

```
(define-macro (cond first-branch . remaining-branches)
  (let ((first-condition (cond-condition first-branch))
        (first-actions (cond-actions first-branch)))
    `(if ,(or (eq? first-condition 'else) first-condition)
         (begin ,@first-actions)
         ,@(if (eq? remaining-branches '())
               '()
               `((cond ,@remaining-branches))))))
```

3.4 DEFINICJA FORMY "while"

Definicja formy “while” nie powinna już wymagać dodatkowych objaśnień:

```
(define-macro (while condition . actions)
  (let ((LOOP (gensym)))
    `(call/cc
       (lambda (break)
         (define ,LOOP
            (lambda ()
              (if ,condition
                  (begin ,@actions (,LOOP)))))
        (,LOOP)))))
```

4 MAKRA OPARTE NA PRZEKSZTAŁCANIU WZORCÓW (syntax-rules)

W poprzednim rozdziale udało się uzyskać dość zgrabną metodologię pisania makr, które są stosunkowo proste w analizie.

5 MAKRA HYBRYDOWE (syntax-case)

6 WARIACJE NA TEMAT MAKR – MASZYNA ABSTRAKCYJNA CK

7 STUDIUM PRZYPADKU – PATTERN MATCHER

8 NOTATKI: Makra w schemie:

1. podstawowe operatory języka scheme: lambda, define,

quote, if, set!, begin, call/cc, apply, cons, car, cdr (define-syntax/define-macro/let-syntax/letrec-syntax)

1. potrzeba makr – przykłady:

let, let*, letrec, cond, and, or, while, and-let*

1. pierwsze rozwiązanie – makra jako procedury

transformujące kod. operatory quasiquote/unquote/unquote-splicing
problemy: higiena, nieczytelność, nieograniczony zasięg definicji
rozwiązanie: gensym problem: komplikacja

1. drugie rozwiązanie – makra jako transformacje

wzorców (syntax-rules). formy let-syntax i letrec-syntax
problem: mała siła wyrazu, czasem potrzebne jest "złamanie" higieny (np. instrukcja "break" w pętli "while")

1. trzecie rozwiązanie – makra jako hybrydy kodu

proceduralnego z kodem transformującym wzorce
problem: rzeczy robią się skomplikowane. Zamiast prostych list mamy dziwne "syntax-objecty"
sposoby "złamania higieny" są niekompozycjonalne

1. próba rozwiązania problemu kompozycjnoalności – makra oparte o maszynę abstrakcyjną CK
2. inne zastosowania: pattern-matcher wrighta-shinna, publish,

e.g.

1. kultura czytania kodu. readscheme