Wykład 14

Język Scheme, cd.

Forma quote

Pary i listy

Pary niemodyfikowalne i modyfikowalne (Racket)

Współdzielenie i tożsamość

Przykład: stos

Typy danych, definiowane przez użytkownika

Dopasowanie do wzorca

Biblioteki i moduły

Jak każdy język programowania, Scheme ma wiele form syntaktycznych i bogate biblioteki. W tym wykładzie zostaną przedstawione wybrane formy, które umożliwią pisanie nieco bardziej zaawansowanych programów i porównanie stylów programowania w języku funkcyjnym z typizacją dynamiczną i statyczną.

Wszystkie omówione tu przykłady są w pliku w14.rkt. Wykorzystują one idiom Racket. Odnośniki do książki Dybviga umożliwiają porównanie form Racketa ze standardem r6rs.

R.K.Dybvig. *The Scheme Programming Language, fourth edition.*
The MIT Press 2009 http://www.scheme.com/tspl4/

Komentarze wierszowe.	Scheme: ;	OCaml: brak	Haskell: --
Komentarze blokowe.	Scheme: #| .. |#	OCaml: (* … *)	Haskell: {- … -}

Podzbiór Scheme – formy podstawowe

<*program*> → <*form*>*

<*form*> → <*definition*> | <*expression*>

<*definition*> → < *variable definition*> | (begin <*definition*>*)

< *variable definition*> → (define <*variable*> <*expression*>)

<*expression*> → <*constant*>
| <*variable*>
| (quote <*datum*>)
| (lambda <*formals*> <*expression*> <*expression*>*)
| <*application*>
| (if <*expression*> <*expression*> <*expression*>)
| (set! <*variable*> <*expression*>)

<*constant*> → <*boolean*> | <*number*> | <*character*> | <*string*> | <*symbol*>

<*formals*> → <*variable*> | (<*variable*>*)
| (<*variable*> <*variable*>* . <*variable*>)

<*application*> → (<*expression*> <*expression*>*)

Forma quote

Wyrażenie (quote <*datum*>) nie jest aplikacją procedury, gdyż nie powoduje ewaluacji argumentu. Jest to odrębna forma składniowa, która pozwala traktować swój argument jako dane. Użycie formy quote w stosunku do takich obiektów jak liczby, wartości logiczne, znaki i napisy nie jest konieczne, ponieważ ich wartościami są same te obiekty.

```
(quote (1 2 3 4))              => '(1 2 3 4)
(quote (+ 3 4))                => '(+ 3 4)
(quote ("to" "jest" "lista"))  => '("to" "jest" "lista")
```

Ze względu na częste używanie formy quote w programach, wprowadzono poprzedzający wyrażenie pojedynczy znak apostrofu ' , który można stosować jako jej skrócony zapis.

```
'(+ 3 4)                       => '(+ 3 4)
'"napis"                       => "napis"
"napis"                        => "napis"
```

Pary i listy

Jak widzieliśmy, w języku Scheme (i Lisp) podstawowym typem strukturalnym jest para. Para jest tworzona za pomocą konstruktora (cons v_1 v_2).

```
> (cons 1 2)   ; lista niewłaściwa
'(1 . 2)   ; to jest odpowiedź w systemie Racket, np. w Chez Scheme nie ma apostrofu
```

Para jest reprezentowana zewnętrznie w notacji „kropkowej” (ang. dotted pair notation) – poprzez ujęcie jej składowych w nawiasy i oddzielenie kropką, otoczoną spacjami.

Struktury listowe są zbudowane z zagnieżdżonych par.

Lista właściwa (ang. proper list) jest zdefiniowana rekurencyjnie jako lista pusta '() (stanowiąca wyróżniony typ danych, to nie jest para) lub para, której drugi składnik jest listą właściwą. W przeciwnym razie ciąg par tworzy *listę niewłaściwą* (ang. improper list, dotted list). Listy niepuste można tworzyć za pomocą konstruktorów cons (właściwe i niewłaściwe) oraz list (tylko właściwe).

Pary niemodyfikowalne i modyfikowalne (Racket)

The Racket Reference, ch. 4.9-10

Dybvig, ch. 6.3 (standard r6rs)

Jedna z istotnych różnic między r6rs, a idiomem Racket polega na tym, że procedura cons w r6rs tworzy pary, które można modyfikować za pomocą procedur set-car! i set-cdr!, natomiast w Racket cons tworzy parę niemodyfikowalną. Do utworzenia pary modyfikowalnej w Racket należy użyć procedury mcons, a do jej modyfikowania pary procedur set-mcar! i set-mcdr!.

Przykłady są w pliku w14.rkt.

Współdzielenie i tożsamość danych

Współdzielenie danych wygląda analogicznie, jak w innych językach funkcyjnych (por. wykład 2, str. 26-29; wykład 6, str. 25-26).
W języku OCaml każde użycie konstruktora wartości z argumentem tworzy nową wartość. W języku Scheme każde użycie cons tworzy nową parę. Symbole są współdzielone; w języku Scheme dla każdej zadanej nazwy istnieje dokładnie jeden niepowtarzalny symbol o tej nazwie. Ponieważ Scheme nie daje możliwości modyfikowania symboli, takie współdzielenie jest bezpieczne i niewykrywalne.

```
(eq? 'a 'a)                              ; => #t
(define xs1 (list 'a 'b))
xs1                                      ; => '(a b)
(define ys1 xs1)
ys1                                      ; => '(a b)
(eq? xs1 ys1)                            ; => #t
(define ys2 (cons (car xs1) (cdr xs1)))
(eq? xs1 ys2)                            ; => #f
(equal? xs1 ys2)                         ; => #t
(define xs2 (list 'c 'd))
(define zs (append xs1 xs2))
(eq? xs2 (cdr (cdr zs)))                  ; => #t
```

Przykład: stos

W celu ilustracji omówionych wyżej form językowych został zdefiniowany prosty stos. Lokalna zmienna ls jest związana z listą, reprezentującą stos. Poniższa funkcja jest nieco zmodyfikowanym przykładem z książki Dybviga (ch. 2.9, p.52). Porównaj ze stosem modyfikowalnym na liście w OCamlu, wykład 7, str. 19.

```
(define make-stack
  (lambda ()
    (let ([ls '()])
      (lambda (msg . args)
        (cond
          [(eqv? msg 'empty?) (null? ls)]
          [(eqv? msg 'push!) (set! ls (cons (car args) ls))]
          [(eqv? msg 'top) (if (null? ls) (error 'empty_stack) (car ls))]
          [(eqv? msg 'pop!) (if (null? ls) (set! ls '()) (set! ls (cdr ls)))]
          [else (error 'unrecognized_stack_operation)])))))
```

Typy danych, definiowane przez użytkownika

The Racket Guide, ch. 5 struktury
Dybvig, ch. 9 rekordy

Strukturę (rekord) definiuje się za pomocą formy struct (define-record-type w r6rs). Forma struct pozwala na korzystanie z wielu opcji, zarówno dla całej struktury, jak i dla poszczególnych pól. Domyślnie struktury są nieprzezroczyste (ang. opaque), a ich pola są niemodyfikowalne.

Oto przykład zdefiniowania typu point dla struktury przezroczystej (ang. transparent), w której pierwsze pole x jest niemodyfikowalne, a drugie pole y jest modyfikowalne:

```
(struct point (x (y #:mutable)) #:transparent)
```

Zostają wtedy zdefiniowane:

(point *x y*)	konstruktor
(point? *obj*)	predykat
(point-x *p*)	akcesor dla pola x
(point-y *p*)	akcesor dla pola y
(set-point-y! *p y*)	mutator dla pola y

Typy danych, definiowane przez użytkownika

Przykłady są w pliku w14.rkt.
Są tam powtórzone przykłady dla rekordów w OCamlu, w szczególności jest zdefiniowana lista cykliczna. Pokazano też, że porównanie strukturalne list cyklicznych daje poprawną odpowiedź (w OCamlu takie porównanie powoduje zapętlenie).

Dopasowanie do wzorca

The Racket Guide, ch. 12

Przykład.

```
(define (fpat v)
  (match v
    [(cons x y) (cons y x)]
    [(point x y) (point y x)]
    [x x]))

(fpat '(2 . 3))         => '(3 . 2)
(fpat (point 2  3))   => (point 3 2)
(fpat 'default)         => 'default
```

Biblioteki i moduły

The Racket Guide, ch. 6	moduły
Dybvig, ch. 10	(standard r6rs)

Biblioteka w idiomie Racket to moduł, przeznaczony do wykorzystywania przez wiele programów. W standardzie r6rs mówi się po prostu o bibliotekach (a nie modułach).

Kilka przykładów definicji stosu jako modułów i porównanie ich z odpowiednimi modułami w OCamlu można znaleźć w folderze w14-programy.