Wykład 13

Wprowadzenie do języka Scheme

Krótka informacja o języku Scheme

Podzbiór Scheme jako stosowany rachunek lambda

Kombinatory punktu stałego w języku Scheme

Typy danych

Porównywanie wartości

Wyjątki

Pary i listy

Podzbiór Scheme – formy podstawowe

Zmienne modyfikowalne, sekwencje

Krótka informacja o języku Scheme

Zdając sobie sprawę z możliwości wykorzystania rachunku lambda w programowaniu, John McCarthy zastosował lambda-notację w projektowaniu języka programowania Lisp (od LISt Processing). Lisp był chronologicznie drugim (po Fortranie) językiem programowania wyższego poziomu, ukazał się w 1958 roku. Jest on prekursorem języków funkcyjnych, takich jak Common Lisp, Scheme, Standard ML, OCaml, Haskell i Clojure.

W Lispie zarówno program, jak i dane są zapisywane w postaci **S-wyrażeń** (ang. S-expressions od symbolic expressions). Lisp wykorzystuje dynamiczne wyznaczanie zakresu zmiennych (ang. dynamic scoping). Wszystkie pozostałe wyżej wymienione języki stosują statyczne wyznaczanie zakresu zmiennych (ang. static scoping).

Scheme został zaprojektowany w latach 1970 w MIT. Jest językiem z typizacją dynamiczną (silną), stosuje ewaluację gorliwą i statyczne wyznaczanie zakresu. Parametry są przekazywane przez wartość.

Krótka informacja o języku Scheme

"Kanoniczną" pozycją dla języka Scheme jest:

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

Należy zwrócić uwagę, że w tej książce jest przedstawiony język Scheme opisany w:

Revised[6] Report on the Algorithmic Language Scheme http://www.r6rs.org/

Jest to faktyczny standard języka Scheme, ponieważ oficjalny standard

IEEE Standard for the Scheme Programming Language

został opublikowany w 1991 roku i jest przestarzały.

Istnieje wiele implementacji języka Scheme, patrz:
http://community.schemewiki.org/?scheme-faq-standards#implementations
W pracowniach jest zainstalowany Racket http://www.racket-lang.org/
Idiom Racket różni się od standardu r6rs, ale go implementuje. Należy w tym celu wybrać odpowiedni język, np. umieszczając na początku programu wiersz #!r6rs
(patrz: The Racket Guide, rozdział 23: Dialects of Racket and Scheme).

DrRacket -> Help -> Help Desk -> The Racket Guide

Krótka informacja o języku Scheme

Zbiór form syntaktycznych należących do rdzenia języka Scheme (ang. core syntactic forms) jest niewielki. Pozostałe formy mogą być zdefiniowane za pomocą form podstawowych jako rozszerzenia syntaktyczne. Po zdefiniowaniu mają one taki sam status jak formy podstawowe. Wszystkie dane (również funkcje) są wartościami pierwszej kategorii. Reprezentacja programów i danych jest taka sama (homoikoniczność), co w połączeniu z niewielkim rdzeniem ułatwia pisanie interpreterów i kompilatorów języka Scheme w języku Scheme (interpretery metacykliczne, ang. meta-circular interpreters).

Te cechy języka Scheme zostały wykorzystane w poniższej książce, która miała wielki wpływ na programy nauczania informatyki:

H.Abelson, G.J.Sussman, J.Sussman. *Struktura i interpretacja programów komputerowych.* WNT 2002 http://mitpress.mit.edu/sicp/

Krótka informacja o języku Scheme

Dane w języku Scheme są **modyfikowalne**!

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!.

W językach z typizacją dynamiczną każdy pogram poprawny składniowo jest kompilowany bez błędów. Poprawność typów jest sprawdzana w czasie wykonania przed wykonaniem odpowiednich operacji. W takich językach programista ma do dyspozycji predykaty sprawdzające, czy wyrażenie jest wymaganego typu. W językach z typizacją statyczną (np. Haskell, OCaml, SML) to nie jest potrzebne, bo poprawność typów jest gwarantowana przez kompilator. W wielu językach typizowanych statycznie są jednak pewne mechanizmy, pozwalające programiście na dynamiczne sprawdzanie typów, np. operator instanceof w Javie lub metoda isInstanceOf w Scali.

Uwaga. Istnieje już R^{7}RS, ale z niewielką liczbą implementacji.

Konwencje nazewnicze w języku Scheme

- W ukształtowanej historycznie terminologii języka Scheme (i innych języków z rodziny Lisp) funkcje noszą nazwę procedur, a dane to obiekty (nie należy ich mylić z obiektami w sensie programowania obiektowego).
- Nazwy predykatów kończą się znakiem zapytanie, np. zero?, eq?.
- predykaty sprawdzające typy, np. boolean?, pair?, list?, składają się z nazwy typu i znaku zapytania.
- Nazwy procedur, przeprowadzających koercję typów, wyglądają tak: *typ1->typ2*.

```
> (symbol->string 'identyfikator)
"identyfikator"
> (string->symbol "identyfikator")
'identyfikator
```

- Nazwy procedur i form syntaktycznych powodujących efekty obliczeniowe, np. set!, set-car!, kończą się wykrzyknikiem.

W idiomie Racket zamiast słowa lambda można używać greckiej litery λ (kombinacja klawiszy ctrl + \ ).

Podzbiór Scheme jako stosowany rachunek lambda

<*program*> → <*form*>*

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

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

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

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

<*formals*> → (<*variable*>)

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

<*boolean*> → #f | #t

<*character*> → #\ <any character>

<*datum*> → <*constant*> | <*pair*> | <*list*>

Przykłady

```
(define factorial
  (lambda (n)
    (if (zero? n)
      1
      (* n (factorial (sub1 n))))))
 (factorial 5)  => 120

(((lambda (m) (lambda (n) (+ m n))) 2) 3)  => 5

(define ***                    ; w identyfikatorach mogą występować rozmaite symbole
    (lambda (n) (* n (* n  n))))
(*** 2) => 8

(quote symbol)  => 'symbol

'symbol  => 'symbol              ; apostrof  to lukier syntaktyczny dla quote
```

W przykładach po => jest podawany wynik ewaluacji wyrażenia.

Wyrażenie let jest rozszerzeniem syntaktycznym (por. z let-polimorfizmem, wykład 12)
(let ([<*variable*> <*expression*>]) <*expression*$_1$>) jest równoważne

((lambda <*variable*> <*expression*$_1$>) <*expression*>)

Nawiasów kwadratowych można używać dla polepszenia czytelności.

Kombinatory punktu stałego w języku Scheme

Y $\equiv \lambda f.\ W\ W$, gdzie $W \equiv \lambda x.\ f\ (x\ x)$

Niestety, tak zdefiniowany kombinator punktu stałego będzie działał poprawnie tylko w językach z ewaluacją leniwą. Musimy zmodyfikować *W* wykorzystując η-ekspansję: $W \equiv \lambda x.\ \lambda y.\ f\ (x\ x)\ y$. Taki kombinator będzie poprawnie obliczał punkty stałe funkcji, ale zwykle to wystarczy (porównaj z kombinatorem punktu stałego w OCamlu z wykładu 11).

```
(define Y1
  (lambda (f)
     (let ([W (lambda (x)(lambda (y) ((f (x x)) y)))])
        (W W))))


(define fact1
  (Y1 (lambda (g) (lambda (n) (if (zero? n) 1 (* n (g (sub1 n))))))

(fact1 5) => 120
```

Kombinatory punktu stałego w języku Scheme

Y $\equiv \lambda f.\ W\ W$, gdzie $W \equiv \lambda x.\ f\ (x\ x)$

Można zastosować η-ekspansję w innym miejscu: $W \equiv \lambda x.\ f\ \ \lambda y.\ (x\ x)\ y$ (porównaj z drugim kombinatorem punktu stałego w OCamlu z wykładu 11).

```
(define Y2
  (lambda (f)
     (let ([W (lambda (x)( f (lambda (y) ((x x) y))))])
        (W W))))


(define fact2
  (Y2 (lambda (g) (lambda (n) (if (zero? n) 1 (* n (g (sub1 n)))))

(fact2 5) => 120
```

Typy danych

Atomowe typy danych

- Liczby: 5, 88.6, 23e5
- Znaki: #\a, #\newline
- Symbole: ’halo , ’ala

Obiekty złożone

- Napisy: "hi there", "ala"
- Pary

Konstruktory: cons
Akcesory: car, cdr
Predykaty: pair?

- Listy

Konstruktory: '(), cons, list
Akcesory: car, cdr
Predykaty: null?, list?

Porównywanie wartości

W języku Scheme (i Lisp) wartość jest referencją do obiektu (te pojęcia są często utożsamiane). Istnieją trzy generyczne predykaty porównujące dowolne wartości i zwracające jedną z wartości logicznych #t lub #f : eq?, eqv?, equal?

eq? definiuje równość tożsamości (por. == w OCamlu, wykład 6, str. 9-11), equal? definiuje równość strukturalną (por. = w OCamlu), ale działa poprawnie również dla struktur cyklicznych, eqv? sprawdza równoważność pośrednią. Szczegóły są w podręcznikach i dokumentacji.

Jeśli znamy typy argumentów, to lepiej używać predykatów dla tych typów, np. = dla wartości numerycznych, string=? dla napisów czy symbol=? dla symboli.

Przykłady są w pliku w13.rkt.

Wyjątki

The Racket Guide, ch. 10
Dybvig, ch. 11 (standard r6rs)

Zostanie tu podany tylko jeden sposób zgłaszania i obsługi wyjątków w systemie Racket.

Wyjątki można zgłaszać przez wywołanie funkcji error, np. (error 'ujemny_argument). Taki wyjątek jest opakowany w strukturę o nazwie exn:fail.

Wyjątki można przechwycić i obsłużyć za pomocą formy with-handlers.

Przykłady są w pliku w13.rkt.

Pary i listy

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).

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*>*)

Lista argumentów funkcji

Jeszcze raz zostały podane formy podstawowe języka Scheme (patrz Dybvig, ch. 3.1). W porównaniu z poprzednim podzbiorem form zmieniły się definicje dwóch kategorii syntaktycznych:

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

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

W definicji funkcji można podać listę (być może pustą) parametrów (<*variable*>*), co po zaaplikowaniu funkcji wymaga podania odpowiedniej liczby argumentów <*expression*>*.

```
(define f2
    (λ(m n)(+ m n)))
; (f2 2 3) => 5
```

Jeśli parametrem formalnym jest zmienna <*variable*> (a nie lista), to po zaaplikowaniu funkcji jest on wiązany z listą, utworzoną z argumentów.

```
(define f3
  (λ p  (+ (car p) (cadr p))))
; (f3 2 3) => 5
```

Lista argumentów funkcji

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

Trzecia klauzula (*<variable>* *<variable>** . *<variable>*) kategorii syntaktycznej *<formals>* stanowi połączenie dwóch poprzednich klauzul. Jeśli lista parametrów formalnych ma postać (var_1 … var_n . var_r), to zmienne var_1 … var_n są wiązane z wartościami pierwszych *n* argumentów, natomiast zmienna var_r jest wiązana z listą, utworzoną z pozostałych argumentów.

```
((lambda (a b c . r) (list a b c r)) 1 2 3 4 5 6)    =>   '(1 2 3 (4 5 6))

(define  f4
  (lambda (m . r) (+ m (car r))))
(f4  2 3)    =>  5
```

Zmienne modyfikowalne, sekwencje

Do pierwszego podzbioru form została dodana klauzula

| (set! <*variable*> <*expression*>)

Procedura set! zmienia wartość zmiennej.

```
(define v 5)
v    ; => 5
(set! v 10)
v    ; => 10
```

W związku z obecnością efektów obliczeniowych w języku Scheme można używać sekwencji (por. OCaml). Jest to uwzględnione w klauzuli dla abstrakcji funkcyjnej.

| (lambda <*formals*> <*expression*> <*expression*>*)

Po zaaplikowaniu do argumentów wartością funkcji będzie wartość ostatniego wyrażenia w sekwencji. Forma syntaktyczna dla sekwencji

(begin <*expression*> <*expression*>*)

jest zdefiniowana jako aplikacja ((lambda () <*expression*> <*expression*>*))

```
(begin (set! v 12) 'wynik)      ;  => 'wynik
v      ;  => 12
((lambda () (+ 1 2) 'wynik))    ;  => 'wynik
```

W domu należy przeczytać rozdział 2 z książki:

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

i/lub:

The Racket Guide, rozdział 2: Racket Essentials

DrRacket -> Help -> Help Desk -> The Racket Guide, rozdział 2: Racket Essentials

W celu uniknięcia problemów w początkowym etapie nauki, proszę w środowisku Racket używać idiomu Racket (`#lang racket` w pierwszym wierszu).