PARADYGMATY I JĘZYKI PROGRAMOWANIA

Przegląd języków funkcyjnych (w10)

Treść

- Przegląd języków funkcyjnych
  - LISP i SCHEME (DrRacket)
    - podstawy
    - listy, reprezentacja list, operacje na listach
    - funkcje wbudowane
  - Haskell, ghc, ghci

LISP i SCHEME

Krótkie wprowadzenie

Historia

- Jest to pierwszy język funkcyjny
- John McCarthy, MIT, 1959
- Pierwsza implementacja: komputer IBM704
- Następne: SCHEME (Racket: http://racket-lang.org) – dialekt
  - Opis: Dybvig, 2003
  - Dostępność
  - Sussman, Steele, MIT, 1975
  - Interpreter REPL (read-evaluate-print loop); używany też w Ruby, Python

Charakterystyka ogólna

- Jednorodność (homogeniczność) danych i programu – wszystko jest listą; operacje na danych i programie wykonywane są wg. tego samego schematu, z użyciem tych samych narzędzi
- Język samodefiniujący się – semantykę Lispa można zdefiniować z pomocą interpretera Lispa
- pętla *read-eval-print* pozwala na interakcję między programem a użytkownikiem

Proste przykłady


```
(+ 3 4) ⇒ 7
7 ⇒ 7
(load "program")

((+ 3 4))
eval: 7 is not a procedure

(+ 3 4) ⇒ 7
((+ 3 4)) ⇒ błąd
```

Cytowanie:

```
(quote (+ 3 4)) ⇒ (+ 3 4)
```

Typowanie dynamiczne

```
(if (> a 0) (+ 2 3) (+ 2 "nic"))
```

Jeśli a jest dodatnie ⇒ 5

Jeśli a<=0 ⇒ błąd

- Predykaty

```
(boolean? a)      ;czy a jest Boolean?
(char? a)         ;czy a jest znakiem?
(string? a)       ;czy a jest napisem?
(symbol? a)
(number? a)
(pair? a)
(list? a)
```

Symbol odpowiada nazwie w innych językach.

- Funkcja lambda

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

Pierwszy argument funkcji lambda jest listą formalnych argumentów (tutaj `x`), a pozostałe tworzą ciało funkcji (tutaj `(* x x)`)

Obliczanie funkcji lambda:

```
((lambda (x) (* x x)) 7) ⇒ 49
```

Proste przykłady

- Nazwy można związać z wartościami poprzez zakresy zagnieżdżone używając konstrukcji `let`:

```
(let ((a 3)
      (b 4)
      (kwadrat
        (lambda
         (x) (* x x)))
      (plus +))
(sqrt (plus (kwadrat a)
(kwadrat b))) )     ⇒ 5
```

`let` posiada dwa argumenty: pierwszy zawiera pary nazwa-wartość (np. `(a 3)`), a drugi jest wartością wyrażenia (zielony). Zakresem wszystkich zmiennych jest tylko drugi argument `let`. Wiązania dotyczą nie tylko zmiennych w zwykłym sensie ale też funkcji, np. `plus, kwadrat`.

```
(let ((a 3)
  (let ((a 4)
        (b a))
    (+ a b))) ⇒ 7
```

Tutaj, `b` przyjmuje wartość zewnętrznego `a` (a więc 3).

Sposób w jaki nazwy stają się widoczne "wszystkie jednocześnie" na końcu listy deklaracji, nie pozwala na definicje rekurencyjne. Do tego celu służy `letrec`:

```
(letrec ((fact
          (lambda (n)
            (if (= n 1) 1
            (* n (fact (- n 1)))))))
   (fact 4)) ⇒ 24
```

Scheme jest językiem o zakresach statycznych. `let` i `letrec` nie zmieniają znaczeń globalnych. Globalne nazwy definiuje się używając `define` (efekty uboczne):

```
(letrec (p_prost
   (lambda (a b)
     (sqrt (+ (* a a) (* b b)))))

(p_prost 3 4) ⇒ 5
```

Typy

- Dwa typy danych
  - atomy
  - listy
    - elementami list są pary
    - pierwszy element pary jest wskaźnikiem do atomu lub listy
    - drugi element pary jest wskaźnikiem do innego elementu lub wskaźnikiem do listy pustej
    - Elementy list są łączone za pomocą drugiego elementu pary
- Nie są to typy w sensie języków imperatywnych

Operacje podstawowe

- Dane literalne oznaczane są przez QUOTE (cytowania)
  - `(QUOTE A)` ⇒ `A`
  - `(QUOTE (A B C))` ⇒ `(A B C)`
  - W skrócie `QUOTE` = `'` (apostrof); np. `'(A B)`;
    Zapis taki (w językach LISP-podobnych jest konieczny z tego powodu, że kod i dane mają taką samą postać

- Operacje podstawowe na listach: `CAR, CDR, CONS`
  - `CAR` = *contents of the adress part of a register*;
  - `CDR` = *contents of decrement part of the register*
    – nazwy części rejestru komputera IBM 704 na którym pierwszy raz implementowano LISP

Operacje podstawowe

- Przykłady

```
(CAR '(A B C)) ⇒ A
(CAR '((A B) C D)) ⇒(A B)
(CAR 'A) ⇒ błąd, A nie jest listą
(CAR '()) ⇒ błąd

(CDR '(A B C)) ⇒(B C)
(CDR '(A)) ⇒()
(CDR '()) ⇒ błąd
```

Operacje podstawowe

- Złożenia operacji

```
(CAAR (...)) == (CAR (CAR (...))
(CADR (...)) == (CAR (CDR (...))
(CADDAR (...)) == (CAR (CDR (CDR (CAR (...))
```

- CONS

```
(CONS 'A '()) ⇒ (A)
(CONS 'A '(B C)) ⇒ (A B C)
(CONS '() '(A B)) ⇒ (() A B)
(CONS '(A B) '(C D)) ⇒ ((A B) C D)
(CONS (CAR lista) (CDR lista)) ⇒ lista
```

Operacje podstawowe

- Para niewłaściwa

```
(CONS 'A 'B) ⇒(A . B)
```

- LIST

```
(LIST 'X 'Y 'Z) ⇒  (X Y Z)
```

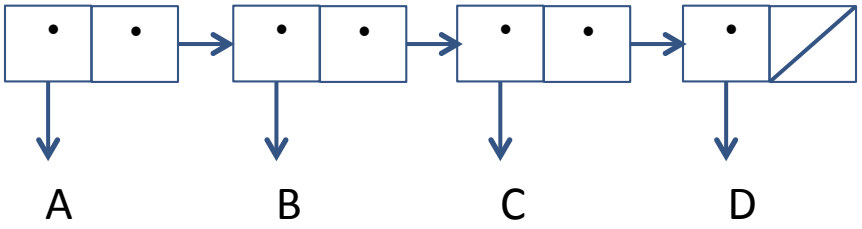
to samo

```
(CONS 'X (CONS 'Y (CONS 'Z '()))) ⇒  (X Y Z)
```

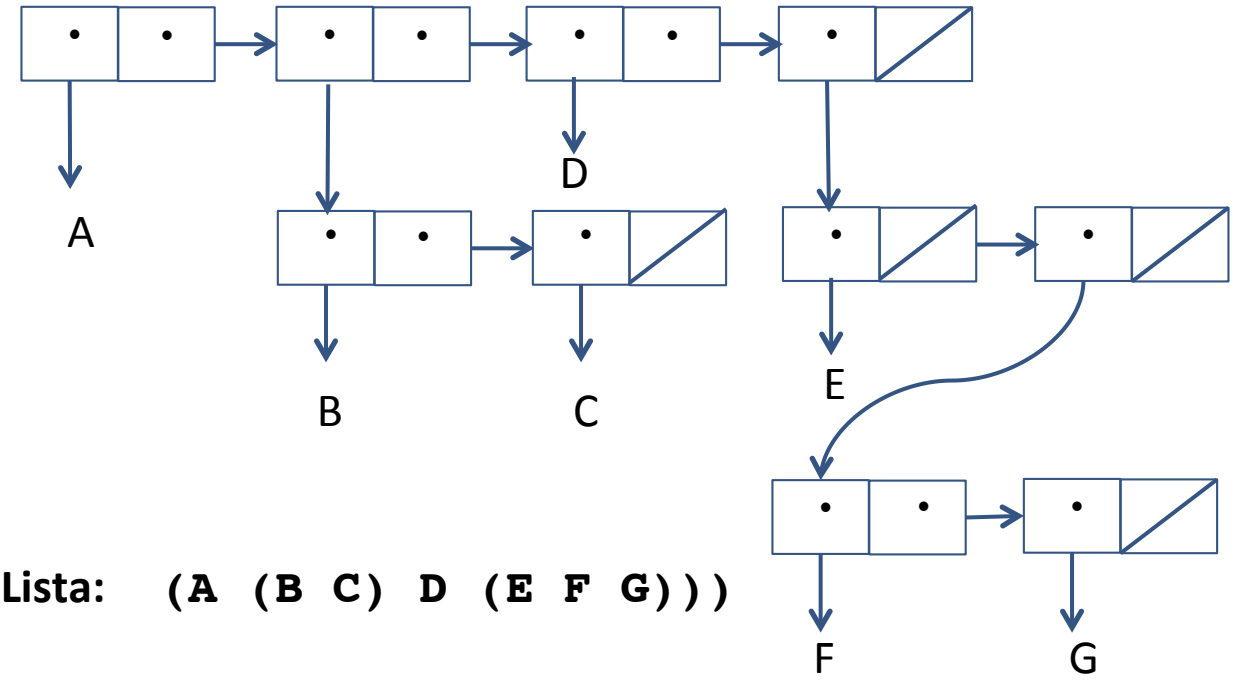
- Predykaty

```
EQ?, EQV?, NULL?, LIST?,...
```

Reprezentacja list



Lista (A B C D)



Lista: (A (B C) D (E F G)))

Funkcje

- Notacja polska (Jan Łukasiewicz, 1920)
  - `(+ 2 3),  (* 1 3 5 7)`
- funkcja lambda
  - `(nazwa (lambda (arg1...argn) wyr.)`
- `EVAL` – funkcja obliczania *wszystkiego;* interpreter LISPA
  - Zakresy dynamiczne od początku;
    obecnie (np. SCHEME):
    `stat | dyn | (oba do wyboru)`

Predykaty, sterowanie

- `=, <>, >, <, >=, <=, EVEN?, ODD?, ZERO?, ...`
- `#T, #F` – wartości logiczne
- `NOT, AND, OR`
- Sterowanie – schemat:

```
(IF predykat wyrażenie_then  wyrażenie_else)
```

- Przykład (`DEFUN` – LISP; `DEFINE` – SCHEME)

```
(DEFUN (silnia n)
       (IF  (<= n 1)
         1
       (* n (silnia (- n 1)))
)
```

Sterowanie

- `COND`

```
(cond
   ((< 3 2) 1)
   ((< 4 3) 2)
   (else 3))      ⇒3
```

- Argumentami `cond` są pary, które obliczane są w kolejności wystąpienia. Wartością całego wyrażenia jest wartość drugiego elementu tej pary, która rozwija się do `#t` (true). Symbolu `else` można użyć tylko w ostatniej parze (oznacza wówczas #t).

Iteracje

- Iteracje zapewnia specjalna postać **do** i funkcja **for-each**:

```
(define iter-fibo (lambda (n)

    ; drukuje n liczb Fibonacciego

    (do ((i 0 (+ i 1)) ; pocz. 0, nast. +1
     (a 0 b)           ; pocz. 0, nast. b
     (b 1 (+ a b)))    ; pocz. 1; suma a, b
    ((= i n) b)        ; test konca; wartosc
  (display b)          ; wnętrze pętli
  (display " "))))     ; wnętrze pętli
```

Przykład definicji funkcji

- LISP

```
(DEFUN (przestepny? rok)
   (COND
      ((ZERO? (MODULO rok 400)) #T)
      ((ZERO? (MODULO rok 100)) #F)
      (ELSE (ZERO? (MODULO rok 4)))
))
```

Kolejność obliczania

- tryb *aplikacyjny*
  - argumenty są obliczane przed przekazaniem do funkcji lub z funkcji
- tryb *normalny*
  - przekazywane są nieobliczone argumenty
- Przykład (Scott):

```
(define double (lambda (x) (+ x x)))
```

  - tryb aplikacyjny

```
   (double (* 3 4))
⇒ (double 12)
⇒ (+ 12 12)
⇒ 24
```

  - tryb normalny

```
   (double (* 3 4))
⇒ (+ (* 3 4) (* 3 4))
⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
```

  - W trybie normalnym wykonuje się dwa razy obliczanie `(* 3 4)`
  - W innych przypadkach może być odwrotnie (patrz: następna strona)

Kolejność obliczania

- Przykład (Scott)

```
(define switch (lambda (x a b c)
   (cond ((< x 0) a)
         ((= x 0) b)
         ((> x 0) c))))
```

  - tryb aplikacyjny

```
  (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
        ((= -1 0) 5)
        ((> -1 0) 7))
⇒ (cond (#t 3)
        ((= -1 0) 5)
        ((> -1 0) 7))
⇒ 3
```

  - tryb normalny

```
  (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4)))
⇒ (cond (#t (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4)))
⇒ (+ 1 2)
⇒ 3
```

  - W trybie normalnym nie są obliczane wyrażenia `(+ 2 3)`, `(+ 3 4)`
  - W obu trybach istnieją wyjątki. W takich specjalnych wyrażeniach jak `cond` funkcje arytmetyczne i logiczne (`+` lub `<`) są obliczone nawet w przypadku gdy obliczenia sa w trybie normalnym.

Czystość, a leniwe wartościowanie

- Tryb obliczeń może mieć wpływ nie tylko na szybkość obliczeń, ale też na poprawność programów – program, błędny z punktu widzenia trybu aplikacyjnego (semantyczne błędy w czasie obliczania) może być poprawny w przypadku normalnego trybu obliczeń (tutaj nie wszystko jest obliczane)

- *Funkcja czysta* – wymagane jest określenie wszystkich jej argumentów
- *Funkcja nieczysta* – przeciwnie
- *Język czysty* – wszystkie funkcje są czyste
- *Język nieczysty* – nie wszystkie funkcje są czyste
- Zdania języka czystego mogą być wartościowane w trybie aplikacyjnym
- Zdania języka nieczystego nie mogą być obliczane w trybie aplikacyjnym
- Scheme i ML są w tym sensie *czyste*
- Haskel nie jest językiem czystym
- *Leniwe wartościowanie* pozwala na tryb normalny obliczeń – nie oblicza się wyrażeń aktualnie niepotrzebnych

Funkcje rekurencyjne

- Przykład (SCHEME)

```
(DEFINE (member elm lista)
   (COND
      ((NULL? lista) #f)
      ((EQ elm (CAR lista)) #t)
       (ELSE (member elm (CDR lista)))
))
```

- Zadanie. Napisać program porównywania prostych list (bez zagnieżdżeń)

DrRacket

- DrRacket:
  - http://racket-lang.org

Język Haskell

Wstęp

Historia, kompilatory

- Lata 80
- Główne trendy:
  - leniwe wartościowanie
  - funkcyjność
- 1987 – konferencja *Functional Programming Languages and Computer Architecture*;
  1990 – Haskell, poprawiony w 1998; Haskell Prime
- Implementacja ghc (Glasgow Haskell Compiler)
  ghci (Glasgow Haskell Compiler interactive)

Charakterystyka języka

- wysokopoziomowy
- funkcyjny
- silnie typowany; typy wnioskowane
- leniwy
- modularny; hermetyzacja w obrębie modułów
- kontraktowy (kontrakty są określone przez klasy typów)
- bez efektów ubocznych; funkcja może zwrócić efekt uboczny, który może być następnie wykorzystany
- ochrona stanów - monady
- dość trudny!

Proste początki


```
bash-3.2$ ghci
GHCi, version 7.4.1: http://www.haskell.org/
ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 2+3
5
Prelude> 5
5
Prelude> (+) 2 4
6
```

Proste początki


```
Prelude> 'x':"y"
"xy"
Prelude> pi
3.141592653589793
Prelude> :t pi
pi :: Floating a => a
Prelude> :t "co to jest?"
"co to jest?" :: [Char]
Prelude> sin 1.2
0.9320390859672263
Prelude> sin (1.2)
0.9320390859672263
Prelude> 2^200
1606938044258990275541962092341162602522202993782792835301376
```

Proste początki


```
Prelude> (sin) 2
0.9092974268256817
Prelude> let dwarazy x = 2*x
Prelude> dwarazy pi
6.283185307179586
Prelude> dwarazy 2
4
Prelude> dwarazy "x"
<interactive>:7:1:
    No instance for (Num [Char])
      arising from a use of `dwarazy'
    Possible fix: add an instance declaration for (Num [Char])
    In the expression: dwarazy "x"
    In an equation for `it': it = dwarazy "x"
Prelude> :t dwarazy
dwarazy :: Num a => a -> a
Prelude>
```

Pliki programów

- Rozszerzenie: **hs (*.hs)**
- Ładowanie:

```
ghci hello.hs     lub w ghci    :load hello.hs
```

```
-- PLIK hello.hs
-- Taken from 'exercise-1-1.hs'
module Main where
c = putStrLn "C!"

combine before after =
  do before
     putStrLn "In the middle"
     after

main = do combine c c
          let b = combine (putStrLn "Hello!") (putStrLn "Bye!")
          let d = combine (b) (combine c c)
          putStrLn "So long!"
```

Pliki programów


```
bash-3.2$ ghci hello.hs
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
C!
In the middle
C!
So long!
*Main>
Leaving GHCi.
```

Haskell w sieci

- http://www.haskell.org
- http://book.realworldhaskell.org
- http://tryhaskell.org
- http://learnyouahaskell.com

Przykład funkcji podwajania

- Imperatywny

```
function doubleIt( n ){

// Double and store
    var result = (n*2);

// Return result
    return( result );
}
```

- Haskell
  - `let double0 n = n+n`
  - `double1 n =`
    `result`
    `where`
    `result = n * 2`
  - `double2 n =`
    `let`
    `result = n * 2`
    `in`
    `result`

Przykład rekurencji (7lang)

- Plik `fib.hs`

```
fib :: Integer -> Integer
fib 0 = 1
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

- Wydajniejsza implementacja; krotki; Plik `fibKr.hs`

```
fibKr :: (Integer, Integer, Integer) -> (Integer, Integer, Integer)
fibKr (x, y, 0) = (x, y, 0)
fibKr (x, y, licznik) = fibKr (y, x+y, licznik - 1)

fibWynik :: (Integer, Integer, Integer) -> Integer
fibWynik (x, y, z) = x

fib :: Integer -> Integer
fib x = fibWynik (fibKr (0, 1, x))
```

Przykład rekurencji


```
bash-3.2$ ghci fibKr.hs
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( fibKr.hs, interpreted )
Ok, modules loaded: Main.

*Main> fib 100
354224848179261915075
*Main> fib 1234
347746739180370201052517440604335969788684934927843710657352239304121649686845967975636459392453053377493026875020744760145842401792378749321113719919618588095724485583919541019961884523908359133457357334538791778480910430756107407761555218113998374287548487
*Main>
```

Ciąg Fibonacciego inaczej


```
  1, 1, 2, 3,  5,  8, 13,  21, ...
+ 1, 2, 3, 5,  8, 13, 21, ...
-----------------------------------
= 2, 3, 5, 8, 13, 21, 34, ...
```

```
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

Definicja **`fibs`** pozwala wyliczać elementy ciągu w czasie liniowym i dodatkowo zapamiętywać raz wyliczony element.

Kompozycja funkcji

- Lista składa się z głowy **(head)** i ogona **(tail)**
- Definicja funkcji wybierającej drugi element listy:

```
let drugi = head . tail
```

Jest to złożenie funkcji **head** i **tail** – kompozycja

- Wyrażenie

```
drugi = head . tail
```

jest równoważnikiem

```
drugi lista = head (tail lista)
```

Kompozycja funkcji

-- Liczby Fibonacciego; krotki --

```
-- Pojedyncza para --
fibNastPara :: (Integer, Integer) -> (Integer, Integer)
fibNastPara (x, y) = (y, x+y)

-- nastepna para --
fibNtaPara :: Integer -> (Integer, Integer)
fibNtaPara 1 = (1, 1)
fibNtaPara n = fibNastPara (fibNtaPara (n-1))


-- pobieramy pierwszy element n-tej krotki --
fib :: Integer -> Integer
fib = fst . fibNtaPara

-- fst.fibNtaPara jest to odpowiednik wyrazenia fst = head (fibNtaPara fst) --
```

Przeglądanie list

- Podział listy na głowę i ogon:

```
*Main> let (g:o) = [1, 2, 3, 4, 5]
*Main> g
1
*Main> o
[2, 3, 4, 5]
*Main>
```

- Dodawanie elementów (listy są homogeniczne)

```
1:[2, 3] ⇒[1, 2, 3]
[1]:[[2, 3], [4]] ⇒ [[1],[2, 3], [4]]
```

- Zakresy

```
[1..4] ⇒ [1, 2, 3, 4]
[10,8..4] ⇒ [10,8,6,4]
take 5 [0,2..] => [0, 2, 4, 6, 8]
```

- Listy składane

```
[x * 3 | x <- [1,2,3]] ⇒ [2, 3, 4]
```

- Dopasowywanie wzorca

```
[(y, x) | (x, y) <- [(1, 2), (2, 3), (3, 1)]
⇒ [(2, 1), (3, 2), (1, 3)]
```

Funkcje wyższego rzędu

- Funkcje anonimowe

```
(\x -> x) „Cokolwiek.” ⇒ Cokolwiek
(\x -> x ++ “ to jest”) “Cokolwiek to jest”
```

- Funkcje `map, where` (plik `map.hs`)

```
module Main where

   kwadratWszystkich lista = map kwadrat lista

      where kwadrat x = x * x
```

Wykonanie:

```
*Main> kwadratWszystkich [2, pi, 2, 3]
[4.0,9.869604401089358,4.0,9.0]
```

Funkcje wyższego rzędu

- Funkcje `filter, foldl, foldr`
  - `filter odd [1..10]`
    `⇒ [1, 3, 5, 7, 9]`
  - Składanie list w lewo i w prawo
    `foldl (+) [1..4] ⇒ 10`
- zwijanie i rozwijanie (currying, uncurrying) funkcji wieloargumentowych
- Leniwe obliczanie
- Zadanie. Napisać funkcję leniwego obliczania elementów ciągu Fibonacciego.

Monady

- wejście/wyjście
- liczby losowe

Literatura

- Scott: Programming language pragmatics
- Bylina + Bylina
- Tate: 7 języków
- Haskell w sieci

za tydzień ... !?