Programowanie imperatywne vs funkcyjne

Programowanie imperatywne
- Program jest ciągiem instrukcji (rozkazów)
- Instrukcje działają na zmiennych, które są alokowane w pamięci komputera
- Celem języka jest efektywne i względnie niezawodne wykonywanie zadań

Programowanie funkcyjne
- Idea oparta na matematycznym pojęciu funkcji
- Niezwiązane z architekturą komputera
- Odzwierciedlające sposób myślenia człowieka

2

Funkcje

- Odwzorowanie f zbioru X (dziedziny) w inny zbiór (zbiór wartości) Y f: X → Y
- Funkcje f: R →R f(x) = $5x^2+bx+c$ f: N → N f(n) = n+(n-1)
- Funkcja matematyczna vs funkcja w imperatywnym języku programowania
  - funkcja. matematyczna – brak zmiennych (zamiast – argumenty funkcji), brak pamięci, brak stanu, definiuje wartość, czasem używa warunków $f(x) = \begin{cases} \cdot, 2x & x \le -1 \\ 1 & -1 \le x \le 1 \\ x^2 & x > 1 \end{cases}$

    często rekurencji *silnia(n) = n*silnia(n-1)*
  - Funkcja w programowaniu imperatywnym to raczej przepis na realizację działań prowadzących do wyniku, ciąg rozkazów działających na zmiennych i stałych

3

Funkcyjnie vs imperatywnie

Funkcyjnie (i rekurencyjnie)

```
(define (minimum lista)
  (cond
   ((null? (cdr lista)) (car lista))
   ((< (car lista) (minimum (cdr lista)))
(car lista))
    (else (minimum (cdr lista)))))

(define (suma lista)
   (if (null? lista) 0
   (+ (car lista) (suma (cdr lista)))
))

(map (λ(x) x*x) (2 5 8 9))
```

Imperatywnie (Iteracyjnie)

```
int function minimum(int c[]) {
   int min=c[0];
   for (i=1, i<=n, i++) {
     if (c[i] < min)
       min=c[i];}
   return min}

int function suma(int c[i]) {
   s =0;
   for (i=1, i<=n, i++) {
   s = suma+c[i] }
   return s;}

for (x in (2,5,8,9)) {
    print x*x ;
}
```

4

Lambda wyrażenia i funkcjonały

Lambda wyrażenia (A. Church, 1941)
- Opisują **nienazwane** funkcje
- Np. lambda-wyrażenie (λ(x) x * x * x) (2) daje wynik 8 i oznacza tyle co x* x * x dla argumentu 2, czyli λ(2) = 8
  - Zamiast np. szescian(x) = x*x*x, szescian(2) = 2*2*2 = 8
  w oryginalnym zapisie Churcha: $\lambda x.x^3 \quad (\lambda x.x^3)(2) = 2^3$

Funkcjonał – funkcja, której parametrem lub wynikiem są inne funkcje np..
- Złożenie funkcji f º g np. f(x) = sinx, g(x)=2x , f º g (x) = f(g(x)) = sin(2x)
- Operator typu *map* stosujący tę samą funkcję do listy argumentów: (*map* (λ(x) x*x) (2 5 8 9)) daje wynik (4 25 64 81)

  czyli map: map(funkcja(x), lista wartości x'ów)

5

Założenia języków funkcyjnych

- Celem jest opisywanie problemów w postaci funkcji w matematycznym sensie
- Nie jest interesujące, JAK przebiegają obliczenia, definiujemy CO ma być wyliczone
- Nie ma zmiennych (nie ma zarządzania pamięcią), są parametry (argumenty) funkcji
- Ta sama funkcja, dla tych samych parametrów daje zawsze ten sam wynik – (referential transparency = przezroczystość odwołań) co nie jest oczywiste w przypadku funkcji imperatywnych
- Nie ma pętli, jest rekurencja
- Jest zdefiniowany zestaw funkcji pierwotnych: operatory +, *, null? „funkcje standardowe" – trygonometryczne, pierwiastkowanie itp.
- Jest narzędzie do konstruowania funkcjonałów (np. map w Scheme)
- Są struktury do reprezentowania parametrów i wyników częściowych (let w Scheme)

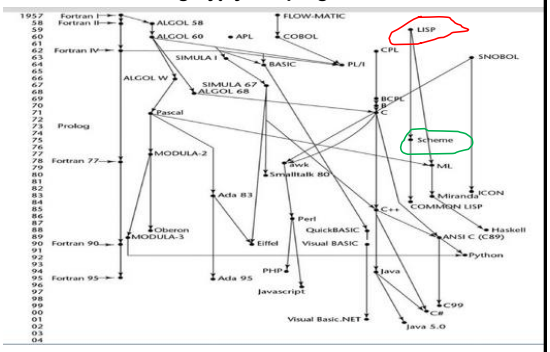
6

Pierwszy język funkcyjny Lisp

- Lisp **Lis**t **P**rocessing , 1958, język przetwarzania list
- Podstawowe typy danych w oryginalnym Lispie: atomy i listy np. (A B (C D) E) – elementy tej listy to atomy A, B, podlista (C D) oraz atom E, listę pustą oznacza się: ()
- W pierwszych wersjach brak typów
- **Wszystko jest listą**: (A B C) może być interpretowane jako lista danych, albo jako funkcja o nazwie A działająca na argumentach B C
- W zapisie wyrażeń (list) stosuje się **Notację Polską** (Łukasiewicz, Cambridge Polish notation) - prefiksową (operacja arg1 arg2 arg3 …) np. (* (+ a b) (- c d)) = (a+b)*(c-d) (fun a b) oznacza funkcję o nazwie fun na argumentach a b
- Funkcja **EVAL** funkcja, która może policzyć każdą inną funkcję, Implementacja funkcji EVAL może służyć (i służyła) za interpreter Lispu.

7

Genealogia języków programowania



8

Wprowadzenie do języka Scheme

- powstał w połowie lat 70 jako jaśniejsza, nowocześniejsza i prostsza wersja Lispu
- używa wyłącznie zakresów statycznych
- Funkcje mogą być:
  - wyrażeniami wyliczającymi jakąś wartość,
  - przypisywane jako wartości zmiennym
  - przekazywane jako parametry
  - zwracane jako wartości przez inne funkcje

Interpreter Scheme

- W trybie interakcyjnym jest nieskończoną pętlą typu
  czytaj - oblicz - wypisz (oryginalnie: **R**ead **E**valuate **P**rint **L**oop -- REPL) (podobnego trybu używają Phyton i Ruby)
- Wyrażenia są interpretowane przez tzw funkcję EVAL
- Napisy nie sa interpretowane

9

Definiowanie funkcji w Scheme

- Program w Scheme jest zbiorem definicji funkcji
- Do definiowania służy polecenie **define.**
- Pozwala definiować wartości (wiązać wartości z nazwami):
  (define e 2.71536) (define alfa 3) (define literaa "A")
  uwaga: e, alfa, literaa NIE SA zmiennymi w imperatywnym sensie
- Pozwala definiować funkcje (DEFINE (kwadrat x) (* x x))
  aplikacja (wywołanie) funkcji dla wartości 3: (kwadrat 3) - wynik 9
- Parametry do funkcji są przekazywane przez wartość
- Obsługa wejścia/wyjścia: zwykle wyjście z interpretera – wynik działania funkcji EVAL
  ponadto (display wyrażenie) – funkcja imperatywna,
  inna funkcja imperatywna: (newline)

10

Program i wykonanie w Scheme (DrRacket)

```
#lang racket
(define (fun x) (* (- x 1) (+ x 1)))
(define (wielo x) (- (* x x x) 8))

(define (calkowanie f dol gora N)
   (let ((przedzial (/ (- gora dol) N)))
     (define (iks m)
      (if (<= m 0) dol
          (+ przedzial (iks (- m 1)))))
     (define (ygrek m)
       (f (iks m)))
     (define (sumay m)
       (if (<= m 0) (ygrek 0)
       (+ (ygrek m) (sumay (- m 1)))))
      (* (sumay N) przedzial)

     ))

(define dol -1)
(define gora 1)
(define N 80)
(define przedzial (/ (- gora dol) N))

(define (iks m)
  (if (<= m 0) dol
     (+ przedzial (iks (- m 1)))))
```

```
Welcome to DrRacket, version 7.3 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (calkowanie fun -1 1 100)
```

11

Podstawowe funkcje i wyrażenia lambda

- podstawowe funkcje arytmetyczne: +, -, *, /, abs, sqrt, min, max i inne
  np. (+ 1 2 3) wynik 6, (sqrt 9) wynik 3 (min 4 2 7 3) wynik 2
- wyrażenia lambda np. (lambda (x) (* x x)) - x jest nazywane zmienną związaną
- lambda wyrażenia mogą być stosowane z parametrami aktualnymi:
- ((lambda (x) (* x x)) 7) wynik - 49
- lambda wyrażenia mogą mieć dowolną liczbę parametrów:
  ((lambda (a b c) (+ (* a b) (* b c))) 1 2 3) wynik 8

12

Funkcja define

- Pozwala definiować wartości (wiązać wartości z nazwami):
  (define e 2.71536) (define alfa 3) (define literaa „A")
  uwaga: e, alfa, literaa NIE SA zmiennymi w imperatywnym sensie
- Pozwala definiować funkcje
  (DEFINE (kwadrat x) (* x x))
  aplikacja (wywołanie) funkcji dla wartości 3:
  (kwadrat 3)
  wynik 9

13

Podstawowe elementy języka

Operatory

- Porównań arytmetycznych
  >, <, >=, <=, <> (> 7 5) wynik true
  EVEN?, ODD?, ZERO?, NEGATIVE? (even? 13) wynik false
  Prawda #t Fałsz #f
  not
- Decyzje „instrukcja warunkowa"
  (if warunek wyrazenie_gdy_prawda wyrażenie_gdy_fałsz)

  (if (> a b) a - b) (if (> n 0) n 0)
  wiele warunków (case)

```
(cond
  (( warunek operacja)
    (warunek operacja )
……
  (else operacja))
```

```
(define przestepny? rok)
    (cond
      ((= (mod rok 400) 0) #t)
       ((= (mod rok 100) 0) #f)
       (else (= (mod rok 4) 0) #t)
    ))
```

14

Operacje na listach

funkcja	przykład	Wynik
Zapis listy	'(1 3 5 8) '("a" "b" "c" "d") dla znaków a, b c użyj cudzysłowów (list 1 3 6 4)	
list	(list 1 3 6 4) (list "a" "b" "c" "d") – lista z danych elem	'(1 2 3 4) '("a" "b" "c" "d")
car/first	(first '(1 2 5 3 5))car (first)	1 - pierwszy element listy
cdr/rest	(rest '(1 5 3 6)) (rest '((1 2 3) 5 (3 4))	'(5 3 6) ogon listy '(5 (3 4))
cons/	(cons 2 '(3 4)) (cons "a" "b")) (cons "a" '("b")	'(2 3 4) dołóż do listy ("a"."b") ("a" "b")
append	(append '(1 2) '(3 4 5))	'(1 2 3 4 5) składa listy

15

Przykłady działań z listami

- Drugi element listy : (car (cdr '(1 2 3 4)) 2
- Czy element a jest na liście:

```
(define (jestnaliscie a lista)
  (cond
    [(null? lista) #f]
    [(eq? a (car lista)) #t]
    [else (jestnaliscie a (cdr lista))]
    ))
```

- Czy dwie listy są jednakowe (nawet gdy nie są listami)

```
(define (rowne lista1 lista2)
 (cond
   [(not (list? lista1)) (eq? lista1 lista2)]
   [(not (list? lista2)) #f]
   [(null? lista1) (null? lista2)]
   [(null? lista2) #f]
   [(rowne (car lista1) (car lista2)) (rowne (cdr lista1) (cdr lista2))]
   (else #f)
   ))
```

16

Funkcja let

- Pozwala związać nazwy z wyrażeniami i obliczenie wartości wyrażeń
  (LET ((nazwa1 wyrażenie1)
  (nazwa2 wyrażenie2) … |
  (nazwan wyrażenien)) ciało)

- Pozwala definiować wartości pomocnicze, nie można jednak raz nadanych wartości zmieniać

```
(define (row_kwadrat a b c)
  (let (
     (pierw_delta_przez_2a (/ (sqrt (- (* b b) (* 4 a c))) (* 2 a)))
     (minus_b_przez_2a (/ (- 0 b) (* 2 a)))
      )
   (list (+ minus_b_przez_2a pierw_delta_przez_2a) (- minus_b_przez_2a
pierw_delta_przez_2a))
   ))
```

17

Funkcja Let

- pozwala nie powtarzać wielokrotnie tych samych wyliczeń, wyliczone wartości nie są dostepne poza funkcją (zakres statyczny, lokalny), nie mogą być również zmieniane

18

Funkcja jako parametr funkcji

- Funkcja generalmap ma parametr fun, który może być funkcją, drugim parametrem jest lista wartości.

```
(define (generalmap fun lst)
  (cond
    ((null? lst) '())
    (else (cons (fun (car lst)) (generalmap fun (cdr lst))))))

(define (kwadrat x) (* x x))
(define (pol x) (/ x 2))
```

- Wynik

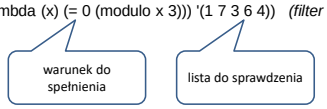
```
> (generalmap kwadrat '(1 2 3 4))
'(1 4 9 16)

; równoważnie, z użyciem funkcji map i lambda-wyrażeń
(define (mapkwadrat lst)
  (map (lambda(x) (* x x)) lst))

(define (mappol lst)
  (map (lambda(x) (/ x 2)) lst))
```

19

Aplikacja funkcji (wyrażenia lambda) do list - funkcjonały map i filter

- We wszystkich językach funkcyjnych są zdefiniowane funkcje **map aplikujące podaną funkcję po kolei do parametrów podanych na liście**
- W szczególności aplikowana funkcja może być zdefiniowana wewnątrz map (użycie funkcji lambda):
- (map (**lambda (liczba) (/ liczba 2)**) '(2 4 6 8 10)) wynik:'(1 2 3 4 5)

- (filter (lambda (x) (= 0 (modulo x 3))) '(1 7 3 6 4)) *(filter warunek lista)*



20

Funkcje tworzace kod

- Program i dane w Scheme mają taką samą postać – są listami dzięki czemu można „dynamicznie" budować kod

Przykład: Policzyć sumę elementów podanej listy można tak:

```
(define (sumator lst)
    (cond
      ((null? lst) 0)
      (else (+ (car lst) (sumator (cdr lst))))
    )
)
```

albo tak

```
(define (sumator1 lst)
    (cond
      ((null? lst) 0)
      (else (eval (cons ' + lst)))
    )
)
```

Czyli używamy funkcji eval z parametrem w postaci listy z dodanym symbolem + na początku lst :('(2 5 4) (cons '+ lst) (+ 2 5 4) i na tym eval

21