Rekurencja

- Rekurencja jest kosztowna obliczeniowo
- Funkcja jest ogonowo rekurencyjna (tail recursive) (gdy jej rekurencyjne wywołanie jest ostatnią jej operacją w funkcji
- Funkcja ogonowo-rekurencyjna może być automatycznie przekształcona przez kompilator do postaci iteracyjnej
- W definicji języka Scheme zaleca się pisanie funkcji rekurencyjnych w postaci ogonowo rekurencyjnej

1

Funkcja rekurencyjna vs ogonowo rekurencyjna

- Klasyczne sformułowanie rekurencyjne funkcji silnia

```
(define (silnia n)
(if (<= n 0) 1
(* n (silnia (- n 1)))))    ;ostatnia operacja to mnożenie
```

- Sformułowanie z użyciem funkcji ogonowo-rekurencyjnej

```
( define  (silnia_ogonowa n pomsilnia)
(if (<= n 0) pomsilnia
(silnia_ogonowa (- n 1) (* n pomsilnia)))   ;ostatnia operacja to wywołanie funkcji
          ))
(define (silnia n)
          (silnia_ogonowa n 1))
```

2

Przebieg obliczeń

(silnia 4)=4* (silnia 3)=4*3*(silnia 2)=4*3*2(silnia 1)
=4*3*2*1*(silnia 0)= 4*3*2*1*1

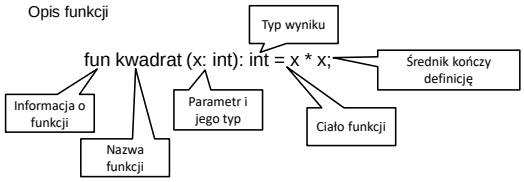
(silnia_ogonowa 4 1)=
(silnia_ogonowa 3 4*1)= (silnia_ogonowa 2 3 *4*1)=
(silnia_ogonowa 2 2*3*4*1) =(silnia_ogonowa 1 1*2*3*4)=
(silnia_ogonowa 1 1*2*3*4*1)

3

Język ML

- Stosuje zakresy statyczne
- Silnie typowany
- Zapis wyrażeń w tradycyjnej postaci (infiksowej - operator pomiędzy operandami)

Opis funkcji



4

Konstrukcje ML a

- Konstrukcje warunkowe: if then else

```
fun silnia(n: int): int = if n = 0 then 1 else n * silnia(n–1);
```

- Alternatywna konstrukcja z dopasowaniem do wzorca

```
fun silnia(0) = 1
| silnia(n: int): int = n * silnia(n–1);
```

znak | oddziela alternatywne definicje

5

ML – działania na listach

- Lista zapisywana w postaci [1 3 5 7]
- Lista pusta []
- Dodanie elementu do listy (odpowiednik CONS/append) :: (dwa dwukropki) np. 1 :: [3 5 7] daje [1 3 5 7]
- Elementy listy muszą być tego samego typu
- Odpowiedniki operatorów listowych CAR - hd (head) CDR – tl (tail)
- Operatora :: można używać w dopasowaniach do wzorca:

```
fun dlugosc([]) = 0
    | dlugosc(h :: t) = 1 + dlugosc(t);
```

- Np. Funkcja łącząca dwie listy:

```
fun dolacz([], lst) = lst
     | append(h :: t, lst) = h :: append(t, lst);
```

| - „znaczy albo”

6

ML – operacje na listach

- map

  fun szescian x = x * x * x;

  val lista_szesciabow = map szescian;

  val nowaLista = lista_szescianow [1, 3, 5];

Wynik: nowaLista=[1 27 125]

Z użyciem funkcji lambda

val newList = map (**fn** x => x * x * x, [1, 3, 5]);

- filter

  filter(**fn**(x) => x < 100, [25, 1, 711, 50, 100]);

  Uzyskany wynik: [25, 1, 50]

  Val jest odpowiednikiem define w Scheme

7

ML - Rozwijanie funkcji (currying)

- Polega na przekształcaniu funkcji wieloparametrowej na funkcję jednoparametrową : np.
  f(x, y) można przekształcić na (f(x))(y) - tu mamy funkcję (f(x)) działającą na parametrze y
- Przykład:
  fun dodaj a b = a + b;
  w istocie ML wykonuje funkcję (dodaj a) b
- fun dodaj5 b = 5 + b;
- W Scheme rozwijanie wyglądałoby tak:

Dla funkcji (define (dodaj a b) (+ a b); jej rozwijanie wygladałoby tak:
(define (dodaj b) (lambda (a) (+ b a)))

Podobne konstrukcje istnieją w językach Haskell, Caml, F#, Scala

8

Język Haskell

Nazwa na cześć **Haskella** Brooksa Curry (ur. 1900 – matematyk i logik z doktoratem z Heidelbergu pod kierunkiem Davida Hilberta)

- Silnie typowany
- Zakresy statyczne
- Stosuje tzw leniwe obliczanie (lazy evaluation)
- Czysto funkcyjny – żadnych efektów ubocznych
- Definicje funkcji mają postać (dopasowanie do wzorca)

```
silnia 0 = 1
silnia 1 = 1
silnia n = n * silnia (n–1)
```

9

Język Haskell 2

- Gdy zachodzi potrzeba opisania warunków, stosuje się notację:

```
silnia n
| n == 0  = 1
| n == 1  = 1
| n > 1  = n *  silnia(n-1)
```

- Stosuje się również odpowiednik „w przeciwnym przypadku” (otherwise)

```
fun n
| n< 10  = 0
| n > 100  =2
|otherwise  =1
```

10

Język Haskell – działania na listach

- Oznaczenie listy: [1 2 3 4 5]
- Lista pusta []
- Odpowiednik CONS - : (dwukropek) np. 1 : [2 3 4 5] daje [1 2 3 4 5]
- Operator łączenia list: ++ - np. [1 2] ++ [3 4 5] daje [1 2 3 4 5]
- Operator # liczy długość listy np. #[1 2 3 4 5] daje 5
- Operator .. Umożliwia zwarte podawanie zakresów
  np. [1 .. 5] to [1 2 3 4 5] ; [3 6 ..10] to [3 6 7 8 9 10]
- Operatora : można używać w definicjach z dopasowaniem do wzorca

```
iloczyn [] = 1
iloczyn (a : x) = a * iloczyn x  (a oznacza głowę listy, x ogon )
```

11

Haskell - operacje listotwórcze

- Poniższe operacje dotyczą list reprezentujących zbiory
- Ogólna postać: [wyrażenie | kwalifikator] np.
  [n * n | n ← [1..20]] - kwadraty liczb od 1 do 20

W Scheme: (map (lambda(n) (* n n)) '(1 2 3 4 5))

- Inny przykład

```
dzielniki n = [ i | i ← [1..n `div` 2], n `mod` i == 0]
```

funkcja dzielniki na argumencie n: utwórz listę elementów i przebiegających wartości od 1 do n div 2 (dzielenie całkowitoliczbowe) takich, że n jest podzielne (mod) przez i

- Szybkie sortowanie (https://www.youtube.com/watch?v=82XxdhRCMbI)

```
sort [] = []
sort (h:t) = sort [b | b←t, b<=h] ++ [h] ++ sort [b | b←t, b>h]
```

12

Quick Sort Haskell

[3 5 2 6 1 4] = [3: [5 2 6 1 4]]

[2 1] 3 [5 6 4]

[2 1] [5 6 4]
1 2 3 4 5 6

13

W Haskelu możliwa jest nieskończoność?

- Dopuszczalne są definicje postaci
  dodatnie = [0..]
  parzyste = [2 4 ..]
  kwadraty = [n*n | n ← [0..]]
- Oczywiście należy z takimi strukturami postępować ostrożnie, aby nie doprowadzić do nieskończonych obliczeń Struktury takie mogą być wygodne np. w tzw *leniwych obliczeniach*
- Możliwe konstrukcje
  nieskończoność :: Integer
  nieskończonosc = nieskończonosc + 1
- Jednak takiej operacji ? nieskończonosc raczej nie należy używać (? To prompt Haskella, jak > w Scheme)
- Ale funkcja
  mnóż :: (Integer, Integer) → Integer
  mnóż (x, y) = if x == 0 || y == 0 then 0 else x * y // || = OR

  Nie zawsze musi skończyć się fatalnie, nawet, gdy np.
  y =nieskonczonosc+1, o ile x =0

14

Obliczanie leniwe (lazy evaluation)

- Parametry funkcji są obliczane tylko wtedy, gdy jest to potrzebne np. jeżeli funkcja ma 2 parametry, a w konkretnym wywołaniu jeden z nich nie jest potrzebny, to nie będzie obliczany
- Przykład: sprawdzanie występowania na liście
  elem1 b [] = False
  elem1 b (a : x) = (a == b) || elem1 b x (|| oznacza OR)

  elem2 n (m : x)
  | m < n = elem2 n x
  | m == n = True
  | otherwise = False
- Wykorzystamy funkcję do sprawdzenia, czy liczba 16 jest kwadratem jakiejś liczby naturalnej
- elem1 16 kwadraty (zdefiniowane na poprzedniaj stronie) - jeśli parametr (16) nie będzie idealnie taki jak na wygenerowanej liście – nastąpi przepełnienie stosu
- elem2 16 kwadraty – nawet jeśli nie trafimy dokładnie w element listy (nawet jeśli nie będzie m == n, to dla pewnego m m>n i zadziała otherwise

15

Język F#

- Rozwinięty na bazie Ocaml następcy ML
- W zasadzie funkcyjny, zawiera jednak cechy imperatywne i obiektowe
- ma bogate IDE i bogatą bibliotekę funkcji, współpracuje z językami platformy .NET
- używane typy danych: listy, n-tki, rekordy, tablice o stałej i zmiennej długości
- wspiera sekwencje np. **let** x = seq {1..4};; **let** seq1 = seq {1..2..7}
- Pozwala definiować sekwencje iteracyjnie
  let cubes = seq {for i in 1..4 -> (i, i * i * i)};;

16

Język F# definicje funkcji

- Jako wyrażenia lambda: (fun a b → a/b)
- Jako nazwane funkcje:
  let f =
    let pi = 3.14
    let dwaPi = 2 * pi

    dwaPI;; (wywołanie interpretacji)

  let rec factorial x = // gdy funkcja jest rekurencyjna należy
    if x <= 1 **then** 1 // jej nazwę poprzedzić frazą **rec**
    else n* factorial(n – 1)

17

Funkcjonały |> i >> w F#

- |> - potok przekazuje wartość lewego operandu do prawego
  let lista = [1; 2; 3; 4; 5]
  let parzysteRazyPiec = lista
    |> List.filter (**fun** n → % 2 = 0) // lista przekazana do filtra parzystości
    |> List.map (**fun** n → 5 *n) // lista parzystycjh przekazana do „spięciokrotnienia"

  wynik: [10; 20]
- >> złożenie funkcji : (f >> g) x jest równoważne g(f(x)

18

Dlaczego F# jest interesujący

- zbudowany w oparciu o wcześniejsze języki funkcyjne
- wspiera praktycznie wszystkie metodologie programowania będące dziś w powszechnym użyciu
- jest pierwszym językiem funkcyjnym zorientowanym na współpracę z innymi szeroko używanymi językami
- posiada bogate IDE i bibliotekę funkcji

- Wsparcie dla programowania funkcyjnego w innych językach: JavaScript, Phyton, Ruby, Java, C#

19

Scala

- Łączy cechy języków funkcyjnych i obiektowych
- Działa na maszynie wirtualnej Javy i .NET
- Statycznie typowany
- Obsługuje funkcje lambda, zagnieżdżanie funkcji, rozwijanie (currying), operacje na listach
- Stosuje leniwe obliczanie (lazy evaluation)
- Przykładowy program (quick sort = szybkie sortowanie)

```
def qsort(list : List[Int]): List[Int] = list match {
    case Nil => Nil
    case pivot :: tail => {
      val (smaller, rest) = tail partition (_ < pivot)
      qsort(smaller) ::: pivot :: qsort(rest)
  } }
```

20

Podsumowanie
najważniejszych elementów paradygmatu

- Języki oparte na idei funkcji matematycznej – program jest zbiorem funkcji
- Oddzielenie struktury programu od architektury komputera (von Neumanna)
- Brak stanów, brak/minimalizacja efektów ubocznych
- Rekurencja zamiast iteracji
- Operacje listowe: map, filter, fold? (aplikowanie wskazanej funkcji do każdego elementu listy)
- Funkcja może być parametrem funkcji
- Funkcja może być wartością funkcji (zwracana przez funkcję)
- Złożenia funkcji stosowane w naturalny sposób
- Zwartość kodu

21

Języki imperatywne vs funkcyjne

imperatywne
- efektywne wykonanie
- Złożona semantyka
- Złożona składnia
- współbieżność nadzorowana przez programistę

funkcyjne
- prosta semantyka
- prosta składnia
- mniej efektywne wykonanie
- Łatwość zrównoleglania automatycznego

22