Corso di Linguaggi di Programmazione

Lezione 6

Alberto Ceselli
alberto.ceselli@unimi.it

Università degli Studi di Milano

12 Marzo 2013

Funzioni che restituiscono tuple di valori

Le funzioni Python possono *restituire un numero arbitrario di valori*

```
def computexy()
...
return x,y

>> a,b = computexy()
```

Funzioni come applicazione parziale di argomenti

Utilizzando `lambda`-expressions, possiamo definire funzioni che sono *specializzazioni* di altre funzioni. Esempio: consideriamo la funzione "+" (somma)

```
> (+ 2 3)
5
```

la funzione incremento può essere definita come

```
(define inc
( lambda (x) (+ x 1) )
)
```

Esempio: funzione replace generica (alla lavagna).

Function currying

Molti linguaggi funzionali (non Scheme) supportano la definizione di funzioni tramite *currying* implicito, ovvero:

- chiamo una funzione specificando un sottoinsieme dei suoi parametri
- questa chiamata mi restituisce una *funzione* che attende come argomenti gli argomenti non specificati
- Esempio: costruire la funzione *RADDOPPIA* partendo dalla funzione *prodotto di due numeri*

```
> (* 2 3)
6
> (* 2)
==> e' la funzione RADDOPPIA
> ( (* 2) 3 )
6
```

(N.B. l'esempio è fittizio, il codice non funziona realmente in interpreti Scheme)

Function currying in Python

```
>>> from functional import curry
>>> computation = lambda a,b,c,d: (a + b**2 + c**3 + d**4)
>>> computation(1,2,3,5)
657
>>> fillZero  = curry(computation)
>>> fillOne   = fillZero(1)
>>> print fillOne(2,3,5)
657
>>> fillTwo   = fillOne(2)
>>> fillThree = fillTwo(3)
>>> answer    = fillThree(5)
>>> answer
657
```

Tail Recursion (1) - Esempio

- Calcolo del fattoriale - versione non *tail recursion*:

```
int  fattoriale (int  n) {
    if (n == 0) return  1;
    return n*fattoriale(n-1);
}
```

  - questa definizione deve eseguire un'operazione matematica (una moltiplicazione) al ritorno di ogni chiamata ricorsiva

- Calcolo del fattoriale - versione *tail recursion*:

```
int  fattoriale_aus (int  n, int result) {
    if (n == 1) return result;
    return fattoriale_aus(n-1, n*result);
}
int fattoriale (int n) {
    return fattoriale_aus(n,1);
}
```

  - si utilizza un accumulatore per i risultati intermedi

Tail recursion (2)

- TL è caso particolare di ricorsione che è possibile ottimizzare e portare allo stesso livello di efficienza dell'iterazione
  - la soluzione è *sintatticamente ricorsiva*, ma il *processo computazionale è iterativo*
- Si chiama **tail recursion** (ricorsione di coda)
  - l'invocazione alla funzione stessa compare sempre e solo come ultima operazione eseguita
  - non ci sono operazioni pendenti che devono venire eseguite al ritorno da una chiamata ad una funzione ricorsiva
- Si invocano le funzioni ricorsivamente
  - ognuna calcola qualcosa
  - alla terminazione il risultato è pronto e tutte ritornano

Accumulatori e Tail Recursion

Funzione potenza con accumulatori (esempi allegati)

Input/Output

- **Output**:

fun	**args**	**valore stampato**
`newline`	-	linea vuota
`display`	*expr*	il valore di *expr* senza apici
`write`	*expr*	il valore di *expr*

- **Input**
  - `read`: legge l'informazione dallo standard input e ritona tale informazione.

even-odd-split

Soluzione del problema “even–odd–split” (esempi allegati)

merge sort

Ordinare una lista tramite merge sort (esempi allegati)

dizionari implementati con liste

Fibonacci con accumulatori, implementati con liste (esempi allegati)

dizionari implementati con funzioni

Fibonacci con accumulatori, implementati come funzioni (esempi allegati)

Esercizi:

Esercizio:

- Realizzare un programma Scheme `derive` che riceva in ingresso un'espressione algebrica contenente *solo operazioni di somma e prodotto* ed il nome di *una variabile*, e restituisca l'espressione algebrica corrispondente alla derivata prima del polinomio rispetto a quella variabile. Esempio:

```
> (derive '(+ x 4) 'x)
(+ 1 0)
> (derive '(* a x) 'x)
(+ (* 0 x) (* a 1))
> (derive '(+ (* a x) (* x 2) ) 'x)
(+ (+ (* 0 x) (* a 1)) (+ (* 1 2) (* x 0)))
> (derive '(+ (* a x) (* x 2) ) 'a)
(+ (+ (* 1 x) (* a 0)) (+ (* 0 2) (* x 0)))
```

Esercizi:

Estensioni facoltative:

- Estendere `derive`, in modo da gestire anche la presenza nell'espressione delle seguenti operazioni
  - sottrazioni
  - divisioni
  - elevamento a potenza