Corso di Linguaggi di Programmazione

Lezione 3

Alberto Ceselli
alberto.ceselli@unimi.it

Università degli Studi di Milano

5 Marzo 2013

Università degli Studi di Milano

Storicamemente

- 1977: ACM Turing Award a John Backus per lo sviluppo del Fortran
- 1978, Seminario per l'Award: Backus afferma che i linguaggi funzionali sono meglio degli imperativi!
  - Piu' facili da capire, sia durante che dopo lo sviluppo
  - significato delle espressioni indipendenti dal contesto
  - propone anche un linguaggio puramente funzionale (FP).
- ... seguito da numerosi scienziati.

Imperativo VS Funzionale

Programmazione imperativa:

- Abbiamo una *macchina alla Von Neumann* da comandare
- questa ha uno *stato* (valore di registri e celle di memoria)
- realizziamo calcoli modificando lo stato della macchina
- forniamo istruzioni (o *comandi*) da interpretare in modo sequenziale secondo un flusso di controllo, e gestiamo questo flusso con costrutti di iterazione, selezione ecc.
- per modificare i valori nelle celle di memoria (e quindi lo stato della macchina), modifichiamo valori di variabili

Imperativo VS Funzionale

Programmazione funzionale:

- Consiste nel definire *funzioni* che producono quanto desiderato, a partire dai loro argomenti
  - la computazione avviene applicando le funzioni ai loro argomenti
- Le funzioni sono definite come quelle matematiche, basandosi più su *condizioni* e *ricorsione*, che su sequenze di istruzioni
- Una funzione inoltre non ha *effetti collaterali*, né assegnamenti
  - Es.: data la funzione $f(x) = x + 1$, valutare $f(x)$ non cambia nulla:
    - f non cambia il suo stato interno, $f(3)$ è sempre 4
    - f non modifica lo stato esterno, non apre un socket o salva un file su disco

Imperativo VS Funzionale

niente stato ...

- niente cicli: uso la *ricorsione* per sostituire *qualsiasi costrutto di iterazione*
- niente assegnamenti e niente variabili, niente gestione della memoria
- solo definizioni di funzioni ed applicazioni di funzioni a dati (niente "effetti collaterali").

Ricorsione (ripasso)

- Dal punto di vista formale un sottoprogramma è *ricorsivo* quando richiama se stesso (nel corpo del sottoprogramma presente una chiamata al sottoprogramma stesso).
- le relazioni di ricorrenza sono delle funzioni definite in maniera ricorsiva (es $n! = n \cdot (n-1)!$).
- Ogni definizione ricorsiva deve essere caratterizzata da:
  - Caso Base (condizione di terminazione): condizione per cui la relazione non è più funzione di se stessa (non "chiama" più se stessa);
  - Passo Induttivo (chiamata ricorsiva): la relazione è funzione di se stessa ("chiama" se stessa), relativamente ad un problema più "piccolo"; la soluzione al problema più piccolo viene combinata con altra informazione per produrre la soluzione al problema originale.

Ricorsione: nota bene

- N.B. senza caso base la funzione richiamerebbe sempre se stessa (loop infinito).
- N.B. i concetti di **ricorsione** ed **induzione matematica** sono strettamente legati.

Pensare ricorsivo ...

Esempi (alla lavagna):

- fattoriale
- numeri di Fibonacci
- somma e potenza

Programmazione funzionale e linguaggi funzionali

Perchè progettare linguaggi dedicati appositamente alla programmazione funzionale?

- mimare la definizione di funzioni matematiche il più possibile
- ... anche nella sintassi
- permettere di scrivere codice per ogni tipo di funzione
- ... senza restrizioni su forma, tipo e numero dei parametri (es. in molti linguaggi imperativi non è possibile definire funzioni che restituiscano altre *funzioni*)
- ... con funzioni che possano restituire una tupla di valori (anziché un singolo valore)

Linguaggi funzionali
Applicazioni

- Dominano nell'area dell'**intelligenza artificiale**
- Rappresentazione della conoscenza
- Trattamento di linguaggio naturale
- Utilizzati nei sistemi esperti
- Alcune applicazioni sviluppate con linguaggi funzionali:
  - Emacs, editor di testo, scritto in LISP
  - AutoCAD, programma per disegno tecnico, scritto in LISP, AutoLISP e VisualLISP
  - Macsyma, programma di matematica simbolica, scritto in LISP

Linguaggi funzionali
Applicazioni

Diversi applicativi hanno un interprete per linguaggi funzionali integrato:

- AutoCAD
- GIMP
- Audacity
- ...

Linguaggi funzionali

- Linguaggi funzionali principali:
  - *LISP*, *Common LISP*: core puramente funzionale + feature imperative
  - *Scheme*: piccolo, statically scoped
  - *ML*, *Haskell*: fortemente tipati, con una sintassi più convenzionale
- Caratteristiche comuni:
  - comprendono un insieme di funzioni predefinite e alcune forme funzionali per costruire funzioni più complesse
  - molti introducono funzioni per permettere anche la programmazione imperativa (e.g., `PROG` in LISP), le iterazioni e gli assegnamenti alle variabili (es. `SET` in Scheme)

Scheme - Storia

- Ideato da Sussman e Steele nel 1975, al MIT
- Dialetto di LISP (LISt Processing) [McCarthy 1960]
- Specifiche del linguaggio:
  - Standard IEEE (ufficiale)
  - "The Revised^6 Report on the Algorithmic Language Scheme" (R6RS) (standard de facto)
  - http://www.schemers.org/Documents/Standards

Perchè Scheme?

- È un linguaggio facile, adatto ai neofiti
- È un linguaggio che può essere facilmente immerso (embedded) ed esteso
  - La Free Software Foundation ha scelto i linguaggi C e Scheme come base per il sistema operativo GNU
  - Il numero degli applicativi e dei sistemi basati su Scheme cresce di mese in mese (e.g. Gimp)
- È un linguaggio potente e flessible con un semantica pulita
  - molti ricercatori lo usano da più di vent'anni per implementare e sperimentare le loro idee
- Offre supporto diretto a molti stili di programmazione
- Molte delle obiezioni mosse contro Scheme sono oggi superate
  - esistono compilatori molto efficienti (Stalin, Chicken)
  - esistono raffinati strumenti di analisi statica (MrSpider)

Scheme in una slide

- Scoping statico
  - per ogni occorrenza di una variabile vincolata è possibile determinare staticamente l'espressione che la definisce
  - questa caratteristica lo differenzia dal LISP
- Le funzioni sono oggetti di prima classe
  - le funzioni sono valori che possono essere assegnati a variabili, passati come argomenti ad altre funzioni o restituiti come valori da una funzione
- Le funzioni possono restituire valori multipli
- Pochi tipi predefiniti
- Case-insensitive

Materiale su Scheme

- Dove ottenere un **interprete**:
  - Dr. Racket (GUI + interprete) http://racket-lang.org/, ambiente che utilizzaremo a lezione, disponibile per molte piattaforme
  - Molti altri: MIT Scheme, Bigloo, Kawa, etc
  - Generalmente non del tutto compatibili!!
- **Manuali**:
  - *Teach Yourself Scheme in Fixnum Days*, di Dorai Sitaram http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
  - *How to Design Programs, An Introduction to Programming and Computing*, AA.VV. http://www.htdp.org/
  - *The Scheme Programming Language, 4th Edition*, http://www.scheme.com/tspl4d/
- Altro:
  - il portale di Scheme http://www.schemers.org/

Scheme - L'interprete

- Tutti gli interpreti Scheme sono basati sul seguente loop (Read-Eval-Print):
  1. **Leggi** un'espressione inserita dall'utente
  2. **Valuta** l'espressione per generare un risultato valido
  3. **Stampa a video** l'eventuale risultato
  4. Ripeti all'infinito.
- Scheme è tipicamente interattivo
  - reagisce immediatamente a quanto inserito
  - esisono anche compilatori, ma più di frequente interpreti

Esempio di utilizzo di DrRacket

Esempio di utilizzo di DrRacket

Come viene chiamata una funzione in Scheme?

- `(nome-funzione lista-argomenti)`
  - sintassi prefissa
  - Es. `(+ 1 2 3 4)`
  - le funzioni possono venire composte: `(+ (* 3 2) 1)`
- **Valutazione di una funzione:**
  1. recupera il primo simbolo dopo la parentesi
  2. cerca la funzione corrispondente
  3. valuta gli argomenti
     - NB: l'ordine di valutazione è volutamente non specificato
  4. applica la funzione ai valori risultanti

Scheme - Elementi di base

- **Atomi**: numeri e simboli (associati a valori)
- **Liste**: sequenze, anche non omogenee, di dati
- **Funzioni**: oggetti di prima classe
- **Forme speciali**: assomigliano ad una funzione, ma l'interprete le valuta in modo diverso
- **Operazioni di I/O**

Tipi di dato (elementari)

- **Numerici**:
  - interi (Es. 15)
  - reali (Es. 1.5)
  - razionali (Es. $1/5$)
  - complessi (Es. $1 + 5i$)
- **Simboli**:
  - utilizzati come identificatori di variabili (Es. *xyz*)
- **Booleani**:
  - $\#f$
  - $\#t$ (ogni valore diverso da $\#f$ viene considerato vero)
- **Valutazione di un tipo numerico o booleano:**
  - Restituisce il valore numerico o booleano
- **Valutazione di un simbolo:**
  - Restituisce il valore associato, altrimenti errore
  - Per non ricevere messaggio d'errore, "commentare" il simbolo non associato usando la forma speciale `quote`

Funzioni aritmetiche

fun	# arg	**valore di ritorno**
+	0 o più	somma degli argomenti
−	1 o più	differenza degli argomenti (da sx a dx)
*	0 o più	prodotto degli argomenti
/	1 o più	quoziente degli argomenti (da sx a dx)
`max`	1 o più	massimo
`min`	1 o più	minimo
`truncate`	*num*	la parte intera di *num*
`sqrt`	*num*	la radice quadrata di *num*
`abs`	*num*	il valore assoluto di *num*
`expt`	*num pwr*	*num* alla *exp*
`quotient`	*num*-1 *num*-2	il quoziente di *num*-1 diviso *num*-2
`remainder`	*num*-1 *num*-2	il resto di *num*-1 diviso *num*-2

- + e * possono avere un qualsiasi numero di argomenti, nel caso non ce ne siano viene restituita l'identità (0 e 1)
- nel caso / abbia un solo argomento *n*, viene restituito 1/*n*

Predicati Tipi Primitivi

fun	# arg	**restituisce #t sse**
`not`	*arg*	*arg* è #f
$<$	*num-1* ... *num-n*	*num-1* $<$ *num-2* ... $<$ *num-n*
$>$	*num-1* ... *num-n*	*num-1* $>$ *num-2* ... $>$ *num-n*
$=$	*num-1* ... *num-n*	*num-1* $=$ *num-2* ... $=$ *num-n*
$<=$	*num-1* ... *num-n*	*num-1* $<=$ *num-2* ... $<=$ *num-n*
$>=$	*num-1* ... *num-n*	*num-1* $>=$ *num-2* ... $>=$ *num-n*
`zero?`	*num*	*num* $= 0$
`positive?`	*num*	*num* > 0
`negative?`	*num*	*num* < 0
`even?`	*num*	*num* pari
`odd?`	*num*	*num* dispari
`number?`	*num*	*num* è un numero
`real?`	*num*	*num* è un reale
`integer?`	*num*	*num* è un intero
`symbol?`	*num*	*num* è un simbolo

Come associare un valore ad un simbolo?

- `(define identificatore valore)`
  - al simbolo `identificatore` viene associato `valore`
  - analogo alla definizione di una costante
  - Es. `(define pi 3.14)`

Programmazione Funzionale
Vantaggi, svantaggi e luoghi comuni

- Permettono uno *sviluppo molto rapido* di prototipi
- La sintassi e la semantica sono più semplici
  - più facile scrivere programmi corretti in assenza di side-effect
- Sono *meno efficienti* dei linguaggi imperativi?
  - i linguaggi funzionali generalmente basati su un interprete, ma esistono anche compilatori
  - i linguaggi imperativi costituiscono un modello molto preciso dell'HW attuale (è piu' facile scrivere compilatori ottimizzanti)
- Sono adatti a progetti software di grandi dimensioni?

Esercizi:

Esercizi:

- Installare un interprete Scheme (es. DrRacket), e provare a valutare semplici espressioni numeriche.
- Scrivere un programma C che calcoli il massimo comun divisore tra due numeri, sia in modo iterativo che in modo ricorsivo (ripassate l'algoritmo di Euclide ;)).