Progetto interprete Scheme

Roberto Lucchi

Linguaggi di Programmazione 2005/06

Chi sono e come contattarmi

- Roberto Lucchi
  - E-mail: lucchi@cs.unibo.it
  - Home page: http://www.cs.unibo.it/~lucchi/
  - Ricevimento: mercoledi, ore 14-15, studio 16

Progetto interprete scheme

- Obbligatorio per superare l'esame
- Sviluppato in linguaggio Java
  - Esistono diversi tool in laboratorio
    - Usate preferibilmente NetBeans
  - In gruppi formati da 2/3 persone
- Quando si consegna?
  - una sola data per sessione

Modalità consegna progetto

- Per mail:
  - all'indirizzo lucchi@cs.unibo.it con oggetto "LP – consegna progetto"
  - Il testo deve contenere la descrizione del gruppo (nome, cognome e matricola degli elementi del gruppo)
  - In allegato il file .zip (altri formati non verranno accettati) del progetto con:
    - Documentazione (file doc.html)
    - Sorgenti progetto
    - File di esempio

Documentazione (doc.html)

- Deve contenere 3 sezioni ben distinte
  - l'elenco dei componenti del gruppo che hanno partecipato al progetto, con indirizzo di posta elettronica (del corso di laurea)
  - una breve descrizione dell'architettura del vostro progetto, identificando i gruppi di classi che formano moduli logici ben definiti
  - note sulla vostra implementazione (tecniche particolari utilizzate, problematiche incontrate nell'uso del paradigma object-oriented e/o del linguaggio Java, ecc.).

Il progetto

- Consiste nell'implementare, in Java, un interprete per un linguaggio funzionale, in particolare MiniScheme che è un sottolinguaggio derivato da Scheme
- Parte opzionale (obbligatoria per chi ha 12 C.F.)
  - Implementare un sistema di type inference per tale linguaggio

Valutazione del progetto

- Valutato in laboratorio con tutti gli elementi del gruppo
- Si basa sull'effettivo funzionamento dell'interprete e sulla sua struttura, verranno anche considerate
  - la chiarezza del codice
  - la corretta applicazione dei meccanismi fondamentali dei linguaggi object-oriented (incapsulamento, polimorfismo ed ereditarietà)
  - l'uso di classi della libreria di Java

Il linguaggio MiniScheme

Non terminali

$$
\begin{array}{lcl}
\langle program \rangle & ::= & \langle define \rangle_1 \ \cdots \ \langle define \rangle_n \\
 & & n \geq 1 \\
\langle define \rangle & ::= & \texttt{(define } id \ \langle expr \rangle \texttt{)} \\
 & | & \texttt{(define (}id_1 \ \cdots \ id_n\texttt{)} \ \langle expr \rangle \texttt{)} \\
 & & n \geq 1 \\
\langle const \rangle & ::= & \texttt{\#t} \mid \texttt{\#f} \\
 & | & int \\
 & | & string
\end{array}
$$

Il linguaggio MiniScheme

$$
\begin{array}{lcll}
\langle expr \rangle & ::= & \langle const \rangle & \\
 & | & id & \\
 & | & \texttt{(and } \langle expr \rangle_1 \cdots \langle expr \rangle_n \texttt{)} & n \geq 0 \\
 & | & \texttt{(or } \langle expr \rangle_1 \cdots \langle expr \rangle_n \texttt{)} & n \geq 0 \\
 & | & \texttt{(cond } \langle branch \rangle_1 \cdots \langle branch \rangle_n \texttt{)} & n \geq 1 \\
 & | & \texttt{(cond } \langle branch \rangle_1 \cdots \langle branch \rangle_n \texttt{ (else } \langle expr \rangle \texttt{))} & n \geq 0 \\
 & | & \texttt{(local (} \langle define \rangle_1 \cdots \langle define \rangle_n \texttt{) } \langle expr \rangle \texttt{)} & n \geq 1 \\
 & | & \texttt{(lambda (} id_1 \cdots id_n \texttt{) } \langle expr \rangle \texttt{)} & n \geq 0 \\
 & | & \texttt{(} \langle expr \rangle_1 \cdots \langle expr \rangle_n \texttt{)} & n \geq 1 \\
\langle branch \rangle & ::= & \texttt{(} \langle expr \rangle \ \langle expr \rangle \texttt{)} &
\end{array}
$$

Esempio

- Il programma che segue definisce l'algoritmo di inversione degli elementi di una lista

```
(define (reverse l)
    (local ((define (aux l1 l2)
                    (cond ((null? l2) l1)
                    (else (aux (cons (car l2) l1)
                               (cdr l2))))))
          (aux (list) l)))

(define (main args)
(reverse (list 1 2 3 4 5 6 7 8 9 10)))
```

- Output: (10 9 8 7 6 5 4 3 2 1)

Interpretazione

- Ai fini dell'implementazione del progetto, si assume che ogni programma MiniScheme debba definire una funzione **main** che accetta una lista di stringhe corrispondenti agli argomenti passati dalla linea di comando. Come in un programma C, tale funzione è quella valutata per prima dall'interprete.

Interpretazione di un linguaggio funzionale

- Valutazione delle espressioni
  - Da espressione *e* a valore *v*, cioè un valore non più riducibile
    - Per esempio i numeri interi 1 e -3, i booleani #f e #t non sono ulteriormente riducibili (in altre parole, sono valutabili emmediatamente)
  - Esempio: l'espressione (+ 1 2) è riducibile attraverso un calcolo che possiamo rappresentare schematicamente
    - (+ 1 2) → 3

Espressioni

- Espressioni più complesse possono richiedere più passi di calcolo prima di ottenere un valore non più riducibile
  - (+ (* 2 3) 4) → (+ 6 4) → 10
    - Indicato anche con (+ (* 2 3) 4) →* 10
  - Si noti che prima di poter calcolare la somma è necessario che tutti gli operandi del + siano diventati dei valori (nell'esempio, 6 e 4). Solo a quel punto l'operatore può calcolare il risultato

Valutazione di espressioni

- Linguaggio funzionale
  - meccanismo di calcolo essenziale la creazione e l'applicazione di funzioni
  - le funzioni contengono semplici espressioni

Espressioni booleane

- Forma and
  - E = (and E1 · · · En)
  - Il valore di E dipende dal valore delle sottoespressoni E1, …, En
  - Se tutte le sottoespressioni si riducono a #t allora anche E si riduce a #t
    - In modo più formale esprimiamo questo con la regola:

$$[\textsc{AndT}]\ \frac{E_i \to \texttt{\#t}, \forall i \in \{1,\ldots,n\}}{(\texttt{and}\ E_1\ \cdots\ E_n) \to \texttt{\#t}}$$

Espressioni booleane

- Se esiste almeno una sottoespressione che riduce a #f allora anche E si riduce a #f
  - In modo più formale esprimiamo questo con la regola:

$$[\textsc{AndF}]\ \frac{E_i \to \texttt{\#t}, \forall i \in \{1,\ldots,j-1\} \quad E_j \to \texttt{\#f}, j \in \{1,\ldots,n\}}{(\texttt{and}\ E_1\ \cdots\ E_n) \to \texttt{\#f}}$$

  - Nota: le Ei vengono valutate in sequenza, quindi non appena viene trovata una Ej che si riduce a #f la E riduce a #f

Espressioni booleane

- Duali sono le regole per risolvere espressioni or

$$[\textsc{OrF}]\ \frac{E_i \to \texttt{\#f}, \forall i \in \{1,\ldots,n\}}{(\texttt{or}\ E_1\ \cdots\ E_n) \to \texttt{\#f}}$$

$$[\textsc{OrT}]\ \frac{\begin{array}{l}E_i \to \texttt{\#f}, \forall i \in \{1,\ldots,j-1\}\\ E_j \to \texttt{\#t}, j \in \{1,\ldots,n\}\end{array}}{(\texttt{or}\ E_1\ \cdots\ E_n) \to \texttt{\#t}}$$

Costrutto cond

- Il costrutto condizionale cond attiva la parte di codice relativa alla prima (nell'ordine in cui appaiono) condizione booleana (guardia) che riduce a #t
- L'idea è che (cond (T1 E1) · · · (Tn En)) si deve ridurre a v se Ti è la prima espressione booleana che riduce a #t e Ei riduce a v

Costrutto cond

- Regole formali:

$$[\textsc{Cond1}]\ \frac{\begin{array}{l}T_i \to \texttt{\#f}, \forall i \in \{1,\ldots,j-1\}\\ T_j \to \texttt{\#t}, j \in \{1,\ldots,n\}\\ E_j \to v\end{array}}{(\texttt{cond}\ (T_1\ E_1) \cdots (T_n\ E_n)) \to v}$$

$$[\textsc{Cond2}]\ \frac{\begin{array}{l}T_i \to \texttt{\#f}, \forall i \in \{1,\ldots,j-1\}\\ T_j \to \texttt{\#t}, j \in \{1,\ldots,n\}\\ E_j \to v\end{array}}{\begin{array}{l}(\texttt{cond}\ (T_1\ E_1)\ \cdots\ (T_n\ E_n)\\ \quad (\texttt{else}\ E_{n+1})) \to v\end{array}}$$

Costrutto cond

- Se nessuna guardia vale #t ed è presente l'else, si applica la seguente regola:

$$[\textsc{Cond3}]\ \frac{T_i \to \texttt{\#f}, \forall i \in \{1,\ldots,n\} \quad E_{n+1} \to v}{\begin{array}{l}(\texttt{cond}\ (T_1\ E_1)\ \cdots\ (T_n\ E_n)\\ \quad (\texttt{else}\ E_{n+1})) \to v\end{array}}$$

- Nota: non c'è nessuna regola che spiega il comportamento del sistema quando l'else non è definito e nessuna guardia risulta vera. Questo significa che in fase di interpretazione si arriva ad uno stato di errore

L'ambiente

- Finora ci siamo limitati ad esaminare le regole di valutazione di espressioni *chiuse*, ovvero di espressioni che non contengono riferimenti a nomi di variabile
- Supponiamo dunque di dover valutare l'espressione

  (and x y)

  che dipende ovviamente da x e y

Definizione dell'ambiente

- L'ambiente definisce il valore di ogni variabile. Per tenere traccia del valore associato ai nomi, definiamo l'ambiente

$$\mathcal{E} : nome \mapsto valore$$

- Dal momento che, in alcune situazioni, le associazioni nome-valore nell'ambiente possono cambiare, annotiamo ogni regola con l'ambiente a cui facciamo riferimento.

Valutazione delle variabili

- La seguente regola dice che, se nell'ambiente E il nome x è associato al valore v, allora possiamo ridurre x a v:

$$[\textsc{Var}]\ \frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \to v}$$

- Le regole viste per le espressioni booleane e il costrutto cond vengono adattate conseguentemente con l'aggiunta dell'ambiente, ma la loro struttura rimane invariata.

Aggiornamento dell'ambiente

- In MiniScheme ci sono tre modi per introdurre nuove associazioni nome-valore
  - definire dei nomi globali (define al top-level)
  - definire dei nomi locali (define all'interno di local)
  - applicare una funzione (in questo caso il valore degli argomenti viene associato ai nomi degli stessi argomenti)

Ambiente: nomi locali

Valuto Ei nell'ambiente di origine

Valuto E nell'ambiente aggiornato

$$[\textsc{Ldef1}]\ \frac{\begin{array}{c}\mathcal{E} \vdash E_i \to v_i, \forall i \in \{1,\ldots,n\} \\ \mathcal{E}[x_1/v_1]\cdots[x_n/v_n] \vdash E \to v\end{array}}{\begin{array}{l}\mathcal{E} \vdash \texttt{(local ((define } x_1\ E_1\texttt{)} \\ \qquad\vdots \\ \texttt{(define } x_n\ E_n\texttt{)) } E\texttt{)} \to v\end{array}}$$

$$\mathcal{E}[x/v](y) = \begin{cases} v, & x = y \\ \mathcal{E}(y), & x \neq y \end{cases}$$

Ambiente: nomi globali

- Le definizioni globali non producono un valore, si limitano a produrre un nuovo ambiente in cui il legame tra il nome definito ed il suo valore è stabilito

$$[\textsc{Gdef1}]\ \frac{\mathcal{E} \vdash E \to v}{\mathcal{E} \vdash \texttt{(define } x\ E\texttt{)} \Rightarrow \mathcal{E}[x/v]}$$

Funzioni come valori

- In un linguaggio funzionale, le funzioni possono essere passate come argomenti di altre funzioni e possono essere ritornate come il risultato di un'altra funzione. Ad esempio, in Scheme (e in MiniScheme) è possibile scrivere

  ((cond (T *) (else +)) E1 E2)

  che riduce a E1+E2 se T riduce a #t e a E1*E2 altrimenti
- Possiamo pensare che + e * siano valori che rappresentano funzioni

Funzioni come valori

- In generale è possibile creare funzioni con il costrutto lambda:

  (lambda (x1 · · · xn) E)
- In presenza di una tale espressione, l'interprete deve generare un valore esattamente come accade quando si introduce una costante booleana o intera. Solo così, infatti, le funzioni possono essere passate come argomenti e ritornate come risultati

Funzioni come valori

- In prima approssimazione, possiamo pensare al valore che rappresenta una funzione come una versione "congelata" della funzione, in attesa di essere applicata ad argomenti.
- Tecnicamente questa versione congelata di una funzione è chiamata chiusura. Rappresentiamo una chiusura con Closure(<x1, . . . , xn>,E)

$$[\textsc{Lam1}]\ \frac{}{\mathcal{E} \vdash (\texttt{lambda}\ (x_1\ \cdots\ x_n)\ E) \rightarrow \texttt{Closure}(\langle x_1, \ldots, x_n \rangle, E)}$$

Assioma (nessuna premessa)

Funzioni come valori: regola App1

- Tentiamo di dare la regola di valutazione

$$[\textsc{App1}]\ \frac{\mathcal{E} \vdash E_0 \rightarrow \texttt{Closure}(\langle x_1, \ldots, x_n \rangle, E) \quad \mathcal{E} \vdash E_i \rightarrow v_i, \forall i \in \{1, \ldots, n\} \quad \mathcal{E}[x_1/v_1] \cdots [x_n/v_n] \vdash E \rightarrow v}{\mathcal{E} \vdash (E_0\ E_1\ \cdots\ E_n) \rightarrow v}$$

- Premesse della regola
  - E_0 riduce in Closure ..(quindi è un lambda..)
  - Gli E_i per i=1, ..,n vanno valutati nell'ambiente corrente e collegati alle var x_i
  - gli argomenti della funzione devono essere esattamente n come i parametri
  - la funzione va valutata nell'ambiente aggiornato

Regola App1

- Questa regola non è del tutto corretta
  - il corpo congelato di una funzione viene valutato in un ambiente che differisce da quello di partenza solo per quanto riguarda gli argomenti della funzione
    - in linea di principio, può essere un ambiente completamente diverso da quello che era visibile nel momento in cui la chiusura è stata creata

Esempio

```
(define a 1)
(define f (lambda (x) a))
(define g (lambda (a) (f 2)))
(define b (g #f))
```

- Nelle intenzioni del programmatore, f rappresenta la funzione costante che ritorna sempre l'intero 1, ma usa un nome *libero* a, definito cioè fuori dalla funzione e tale che non è nemmeno un argomento
- La chiusura di f è Closure(<x>, a)

Esempio

- Al momento della valutazione di (g #f), per determinare il valore di b, l'ambiente globale è:

$$\mathcal{E} = \{\mathtt{a} \mapsto 1,\ \mathtt{f} \mapsto \mathtt{Closure}(\langle \mathtt{x} \rangle, \mathtt{a}),\ \mathtt{g} \mapsto \mathtt{Closure}(\langle \mathtt{a} \rangle, (\mathtt{f}\ 2))\}$$

- La regola App1 ci dice che il corpo di g (cioè (f 2)) va valutato nell'ambiente

$$[\textsc{App1}]\ \frac{\mathcal{E} \vdash g \to \mathtt{Closure}(\langle a \rangle, (f\ 2)) \quad \mathcal{E} \vdash \#f \to \#f \quad \mathcal{E}[a/\#f] \vdash g \to v}{\mathcal{E} \vdash (g\ \#f) \to v} \qquad \mathcal{E}' = \{\mathtt{a} \mapsto \#f,\ \mathtt{f} \mapsto \mathtt{Closure}(\langle \mathtt{x} \rangle, \mathtt{a}),\ \mathtt{g} \mapsto \mathtt{Closure}(\langle \mathtt{a} \rangle, (\mathtt{f}\ 2))\}$$

Esempio

- La valutazione di (f 2) avviene in modo analogo, la solita regola di applicazione ci dice di valutare il corpo di f, che è a, nell'ambiente

$$\mathcal{E}'' = \{\mathtt{x} \mapsto 2,\ \mathtt{a} \mapsto \#\mathtt{f},\ \mathtt{f} \mapsto \mathtt{Closure}(\langle \mathtt{x} \rangle, \mathtt{a}),\ \mathtt{g} \mapsto \mathtt{Closure}(\langle \mathtt{a} \rangle, (\mathtt{f}\ 2))\}$$

- .. ottenendo come valore finale #f quando invece, come detto prima, ci saremmo aspettati 1

Scheme

- È un linguaggio con *scoping statico*
  - significa che il legame tra nomi liberi che compaiono in una funzione e valori è stabilito al momento della definizione della funzione
  - Invece App1 usa come ambiente di valutazione del corpo della funzione, esattamente l'ambiente disponibile al momento dell'applicazione, fatti i dovuti aggiornamenti per gli argomenti

La giusta regola è …

- La soluzione a questo problema consiste nell'arricchire la chiusura, facendo in modo di ricordarsi in essa non solo gli argomenti ed il corpo della funzione che essa rappresenta, ma anche l'ambiente che era visibile al momento della definizione.

Valutazione di funzioni (Lam e App)

- La giusta regola per la valutazione di un lambda è quindi:

$$[\textsc{Lam}]\ \frac{}{\mathcal{E} \vdash \texttt{(lambda } (x_1 \ \cdots \ x_n) \ E\texttt{)} \rightarrow \texttt{Closure}(\mathcal{E}, \langle x_1, \ldots, x_n \rangle, E)}$$

- … e per le funzioni:

$$[\textsc{App}]\ \frac{\begin{array}{l}\mathcal{E} \vdash E_0 \rightarrow \texttt{Closure}(\mathcal{E}', \langle x_1, \ldots, x_n \rangle, E) \\ \mathcal{E} \vdash E_i \rightarrow v_i, \forall i \in \{1, \ldots, n\} \\ \mathcal{E}'[x_1/v_1] \cdots [x_n/v_n] \vdash E \rightarrow v\end{array}}{\mathcal{E} \vdash (E_0 \ E_1 \ \cdots \ E_n) \rightarrow v}$$

La regola App

- Notare che le sotto-espressioni Ei vengono tutte valutate nell'ambiente relativo all'applicazione, mentre il corpo della funzione viene valutato nell'ambiente ottenuto dalla chiusura (e che rappresenta l'ambiente che era visibile quando la chiusura è stata creata) aggiornato con i legami relativi agli argomenti.

Definizioni ricorsive

- L'inclusione dell'ambiente nelle chiusure permette di gestire correttamente lo scoping statico, ma ha come effetto collaterale quello di complicare la gestione delle funzioni ricorsive.
- Supponiamo di definire una funzione fact che calcola il fattoriale di un numero:

```
(define fact (lambda (x)
   (cond ((= 0 x) 1)
         (else (* x (fact (- x 1)))))))
```