

Laboratorio di Programmazione A.A. 2004-2005

Progetto: interprete per MiniScheme

Luca Padovani
lpadovan@cs.unibo.it

1 Introduzione

Scopo del progetto è scrivere un interprete in Java per il linguaggio MiniScheme definito nella tabella 1. MiniScheme è un sottoinsieme del linguaggio Scheme [1, 2].

Il progetto deve essere svolto in gruppi di 3 persone. Un progetto viene valutato sufficiente se raggiunge il punteggio di 18/30. La valutazione del progetto si basa sull'effettivo funzionamento dell'interprete e sulla sua struttura, in particolare verranno *anche* considerate:

- la chiarezza del codice;
- la corretta applicazione dei meccanismi fondamentali dei linguaggi object-oriented, in particolare incapsulamento, polimorfismo ed ereditarietà;
- l'uso di classi della libreria di Java;
- la suddivisione del programma in classi coerenti e ben definite.

La consegna del progetto deve essere fatta elettronicamente, inviando *il codice sorgente* in un archivio `.tar.gz` o `.zip` (altri formati non verranno accettati) all'indirizzo email `lpadovan@cs.unibo.it`. In aggiunta al codice sorgente, *deve* essere presente un `Makefile` che causi la compilazione del progetto a seguito del comando `make` e *deve* essere presente un file `README.txt` (in formato testo) o `README.tex` (in formato \LaTeX) contenente tre sezioni ben distinte con:

1. l'elenco dei componenti del gruppo che hanno partecipato al progetto, con indirizzo di posta elettronica;
2. una breve descrizione dell'architettura del vostro progetto, identificando i gruppi di classi che formano moduli logici ben definiti;

3. note sulla vostra implementazione (tecniche particolari utilizzate, problematiche incontrate nell'uso del paradigma object-oriented e/o del linguaggio Java, ecc.).

1.1 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)))
```

L'output di tale programma deve essere

```
(10 9 8 7 6 5 4 3 2 1)
```

Ai fini dell'implementazione del progetto, assumere 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.

2 Interpretazione di un linguaggio funzionale

Come in ogni progetto, prima di mettere mano al codice occorre farsi un'idea il più possibile precisa di *cosa* occorre per risolvere il problema. In questa sezione cercheremo di capire cosa significhi valutare un programma funzionale, e lo faremo per gradi, partendo dai costrutti più semplici e aggiungendo via via quelli più complicati.

Tabella 1: Grammatica del linguaggio MiniScheme

$\langle program \rangle ::= \langle define \rangle_1 \cdots \langle define \rangle_n$	$n \geq 1$
$\langle define \rangle ::= (\text{define } id \langle expr \rangle)$ $(\text{define } (id_1 \cdots id_n) \langle expr \rangle)$	$n \geq 1$
$\langle const \rangle ::= \#t \mid \#f$ int $string$	
$\langle expr \rangle ::= \langle const \rangle$ id $(\text{and } \langle expr \rangle_1 \cdots \langle expr \rangle_n)$	$n \geq 0$
 $(\text{or } \langle expr \rangle_1 \cdots \langle expr \rangle_n)$	$n \geq 0$
 $(\text{cond } \langle branch \rangle_1 \cdots \langle branch \rangle_n)$	$n \geq 1$
 $(\text{cond } \langle branch \rangle_1 \cdots \langle branch \rangle_n (\text{else } \langle expr \rangle))$	$n \geq 0$
 $(\text{local } (\langle define \rangle_1 \cdots \langle define \rangle_n) \langle expr \rangle)$	$n \geq 1$
 $(\text{lambda } (id_1 \cdots id_n) \langle expr \rangle)$	$n \geq 0$
 $(\langle expr \rangle_1 \cdots \langle expr \rangle_n)$	$n \geq 1$
$\langle branch \rangle ::= (\langle expr \rangle \langle expr \rangle)$	

Per interpretazione di una espressione intendiamo la riduzione di una espressione ad un *valore*, che è una speciale espressione non ulteriormente riducibile. Per esempio i numeri interi 1 e -3 sono valori, così come lo sono i due valori booleani #f e #t. Una espressione (+ 1 2) è riducibile attraverso un calcolo che possiamo rappresentare schematicamente così:

$$(+ \ 1 \ 2) \rightarrow 3$$

Espressioni più complesse possono richiedere più passi di calcolo prima di ottenere un valore non più riducibile. Ad esempio

$$(+ \ (* \ 2 \ 3) \ 4) \rightarrow (+ \ 6 \ 4) \rightarrow 10$$

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.

2.1 Espressioni

Un programma scritto in un linguaggio funzionale ha come meccanismo di calcolo essenziale la creazione e l'applicazione di funzioni, ma tali funzioni contengono comunque semplici *espressioni*. Vediamo per cominciare come interpretare espressioni booleane.

Consideriamo una espressione E della forma

$$E \equiv (\text{and } E_1 \cdots E_n)$$

Il valore di tale espressione dipende dal valore delle sotto-espressioni E_1, \dots, E_n . Se queste si riducono tutte al valore #t, allora anche E si riduce a #t. Esprimiamo questo concetto in modo formale e conciso con una regola come la seguente:

$$[\text{ANDT}] \frac{E_i \rightarrow \#t, \forall i \in \{1, \dots, n\}}{(\text{and } E_1 \cdots E_n) \rightarrow \#t}$$

Sopra la linea orizzontale si trovano le premesse, sotto la linea la conclusione. La regola dice che, qualora le premesse siano soddisfatte (in questo caso, tutte le E_i riducono a #t), vale la conclusione (in questo caso, l'intera espressione riduce a #t).

Se una delle E_i riduce a #f, allora l'espressione E riduce a #f. Vogliamo però essere più precisi a riguardo, dicendo che E riduce a #f *non appena* si scopre una qualsiasi E_j che riduce a #f. Ammettendo di ridurre le E_i da sinistra verso destra, possiamo indicare questo fatto con la regola seguente:

$$[\text{ANDF}] \frac{E_i \rightarrow \#t, \forall i \in \{1, \dots, j-1\} \\ E_j \rightarrow \#f, j \in \{1, \dots, n\}}{(\text{and } E_1 \cdots E_n) \rightarrow \#f}$$

Le regole per l'operatore **or** sono duali:

$$\begin{aligned} \text{[ORF]} \quad & \frac{E_i \rightarrow \#f, \forall i \in \{1, \dots, n\}}{(\text{or } E_1 \ \dots \ E_n) \rightarrow \#f} \\ \text{[ORT]} \quad & \frac{\begin{array}{l} E_i \rightarrow \#f, \forall i \in \{1, \dots, j-1\} \\ E_j \rightarrow \#t, j \in \{1, \dots, n\} \end{array}}{(\text{or } E_1 \ \dots \ E_n) \rightarrow \#t} \end{aligned}$$

Consideriamo ora il costrutto condizionale **cond**. Le regole **COND1** e **COND2** dicono che in un costrutto **cond** la prima guardia T_j che vale **#t** determina l'espressione E_j da valutare:

$$\begin{aligned} \text{[COND1]} \quad & \frac{\begin{array}{l} T_i \rightarrow \#f, \forall i \in \{1, \dots, j-1\} \\ T_j \rightarrow \#t, j \in \{1, \dots, n\} \\ E_j \rightarrow v \end{array}}{(\text{cond } (T_1 \ E_1) \ \dots \ (T_n \ E_n)) \rightarrow v} \\ \text{[COND2]} \quad & \frac{\begin{array}{l} T_i \rightarrow \#f, \forall i \in \{1, \dots, j-1\} \\ T_j \rightarrow \#t, j \in \{1, \dots, n\} \\ E_j \rightarrow v \end{array}}{(\text{cond } (T_1 \ E_1) \ \dots \ (T_n \ E_n) \\ \quad \quad \quad (\text{else } E_{n+1})) \rightarrow v} \end{aligned}$$

Se nessuna guardia vale **#t** ed è presente l'**else**, si ha

$$\text{[COND3]} \quad \frac{T_i \rightarrow \#f, \forall i \in \{1, \dots, n\} \quad E_{n+1} \rightarrow v}{(\text{cond } (T_1 \ E_1) \ \dots \ (T_n \ E_n) \\ \quad \quad \quad (\text{else } E_{n+1})) \rightarrow v}$$

Non c'è alcuna regola che spiega il comportamento del costrutto **cond** quando questo è privo di **else** e tutte le guardie T_i riducono a **#f**. Ciò significa che, se tale condizione dovesse verificarsi in fase di interpretazione, la valutazione terminerà con un errore.

2.2 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** $x \ y$)

Il valore di tale espressione dipende naturalmente dal valore di x ed y . Per tenere traccia del valore associato ai nomi, introduciamo un *ambiente* $\mathcal{E} : \text{nome} \mapsto \text{valore}$ che è una mappa da nomi a valori. Indicheremo quindi con $\mathcal{E}(x)$ il valore associato al nome x nell'ambiente \mathcal{E} . Dal momento che, in alcune situazioni, le

associazioni nome-valore nell'ambiente possono cambiare, annotiamo ogni regola con l'ambiente a cui facciamo riferimento.

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

$$\text{[VAR]} \quad \frac{\mathcal{E}(x) = v}{\mathcal{E} \vdash x \rightarrow v}$$

Le regole viste nella sezione 2.1 vengono adattate coseguentemente con l'aggiunta dell'ambiente, ma la loro struttura rimane invariata.

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).

Concentriamoci per il momento alla definizione di nomi locali, abbiamo:

$$\begin{aligned} \text{[LDEF1]} \quad & \frac{\begin{array}{l} \mathcal{E} \vdash E_i \rightarrow v_i, \forall i \in \{1, \dots, n\} \\ \mathcal{E}[x_1/v_1] \ \dots \ [x_n/v_n] \vdash E \rightarrow v \end{array}}{\mathcal{E} \vdash (\text{local } (\text{define } x_1 \ E_1) \\ \quad \quad \quad \vdots \\ \quad \quad \quad (\text{define } x_n \ E_n) \ E) \rightarrow v} \end{aligned}$$

in cui usiamo la notazione $\mathcal{E}[x/v]$ per indicare un nuovo ambiente ottenuto da \mathcal{E} aggiungendo l'associazione $x \mapsto v$. In altri termini

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

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. Indichiamo questo fatto con una variante delle regole viste finora:

$$\text{[GDEF1]} \quad \frac{\mathcal{E} \vdash E \rightarrow v}{\mathcal{E} \vdash (\text{define } x \ E) \Rightarrow \mathcal{E}[x/v]}$$

I legami introdotti al momento dell'applicazione di una funzione verranno descritti nella sezione 2.3.

2.3 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)
```

in cui viene eseguita la somma o la moltiplicazione di E_1 ed E_2 a seconda che T riduca a $\#f$ o $\#t$ rispettivamente. Possiamo pensare che $+$ e $*$ siano valori che rappresentano funzioni, uno dei due viene “scelto” ed applicato. Più 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 ritornare come risultati. 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 $\text{Closure}(\langle x_1, \dots, x_n \rangle, E)$. Otteniamo

$$[\text{LAM1}] \frac{}{\mathcal{E} \vdash (\text{lambda } (x_1 \dots x_n) E) \rightarrow \text{Closure}(\langle x_1, \dots, x_n \rangle, E)}$$

Si noti che la regola non ha premesse. In questo caso si dice che la regola è un assioma (ed in bibliografia la si troverebbe scritta senza la riga orizzontale). I nomi degli argomenti vengono ricordati nella chiusura in quanto saranno necessari per costruire i legami appropriati al momento dell'applicazione.

Possiamo tentare di esprimere la regola per l'applicazione di funzione in questo modo:

$$[\text{APP1}] \frac{\begin{array}{l} \mathcal{E} \vdash E_0 \rightarrow \text{Closure}(\langle x_1, \dots, x_n \rangle, E) \\ \mathcal{E} \vdash E_i \rightarrow v_i, \forall i \in \{1, \dots, n\} \\ \mathcal{E}[x_1/v_1] \dots [x_n/v_n] \vdash E \rightarrow v \end{array}}{\mathcal{E} \vdash (E_0 E_1 \dots E_n) \rightarrow v}$$

La regola dice che, qualora ci trovassimo a valutare una espressione della forma $(E_0 E_1 \dots E_n)$, dovremmo valutare E_0 per prima ed aspettarci di trovare una chiusura per una funzione con esattamente n argomenti. Successivamente valuteremo gli argomenti della funzione E_i per ottenere i rispettivi valori v_i . Solo a

quel punto saremmo in grado di valutare il corpo della funzione, che era stato congelato nella chiusura al momento della sua creazione.

Purtroppo le regole viste, per quanto vicine a quelle corrette, inducono un comportamento anomalo dei programmi. Il nocciolo del problema è che nella regola APP1 il corpo congelato di una funzione viene valutato in un ambiente \mathcal{E}' che differisce da \mathcal{E} solo per quanto riguarda gli argomenti della funzione. Esso però, in linea di principio, può essere un ambiente completamente diverso da quello che era visibile nel momento in cui la chiusura è stata creata. Per vedere come ciò sia causa di problemi, supponiamo di eseguire il seguente programma Scheme (e MiniScheme):

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

Il programma definisce un primo valore a ed una funzione f . Nelle intenzioni del programmatore, f rappresenta la funzione costante che ritorna sempre l'intero 1. Si noti che il corpo della funzione f fa riferimento ad un nome *libero*, a , che è definito all'esterno del corpo della funzione e che non è uno dei parametri della funzione. Possiamo immaginare che la chiusura per f sia $\text{Closure}(\langle x \rangle, a)$. Al momento della valutazione di $(g \#f)$, per determinare il valore di b , l'ambiente globale è

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

La regola APP1 applicata a $(g \#f)$ ci dice di valutare il corpo di g , che è $(f 2)$, nell'ambiente

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

ottenuto aggiornando \mathcal{E} con il legame $a \mapsto \#f$. La valutazione di $(f 2)$ avviene in modo analogo, e la solita regola di applicazione ci dice di valutare il corpo di f , che è a , nell'ambiente

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

ottenendo come valore finale $\#f$, quando invece ci saremmo aspettati 1.

Come abbiamo anticipato, il problema risiede in APP1. Scheme è un linguaggio con *scoping statico*, il che 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 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*. Rivediamo quindi la regola per la valutazione di un **lambda**:

$$[\text{LAM}] \frac{}{\mathcal{E} \vdash (\text{lambda } (x_1 \dots x_n) E) \rightarrow \text{Closure}(\mathcal{E}, \langle x_1, \dots, x_n \rangle, E)}$$

La regola per l'applicazione viene corretta conseguentemente:

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

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

2.4 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)))))))
```

In base alla regola GDEF1, quando questa definizione avviene in un ambiente \mathcal{E} , essa determina un nuovo ambiente \mathcal{E}' come segue

$$\frac{\mathcal{E} \vdash (\text{lambda } (x) E) \rightarrow \text{Closure}(\mathcal{E}, \langle x \rangle, E)}{\mathcal{E} \vdash (\text{define fact } (\text{lambda } (x) E)) \Rightarrow \mathcal{E}'}$$

dove

$$\mathcal{E}' = \mathcal{E}[\text{fact}/\text{Closure}(\mathcal{E}, \langle x \rangle, E)]$$

Ora, applicando **fact** ad un numero n , si ottiene la valutazione di E nell'ambiente

$$\mathcal{E}'' = \mathcal{E}[x/n]$$

ed eventualmente la valutazione di E necessiterà di risolvere il nome **fact**, al momento della chiamata ricorsiva. Tuttavia, l'ambiente in cui questa valutazione avviene è \mathcal{E} in cui il valore dell'argomento x è correttamente legato, mentre il nome **fact** è sconosciuto! Il problema è che l'ambiente memorizzato nella chiusura è l'ambiente visibile *prima* della dichiarazione, mentre, per fare in modo che le funzioni siano ricorsive, esso dovrebbe essere l'ambiente risultante *dopo* la dichiarazione. In formule, la regola per le definizioni globali dovrebbe essere

$$[\text{GDEF}] \frac{\mathcal{E}[x/v] \vdash E \rightarrow v}{\mathcal{E} \vdash (\text{define } x E) \Rightarrow \mathcal{E}[x/v]}$$

Purtroppo la premessa

$$\mathcal{E}[x/v] \vdash E \rightarrow v$$

richiede una cosa apparentemente assurda, ovvero la valutazione di E in un ambiente dove è già noto il valore restituito da E ! Vedremo nella parte di implementazione che questa richiesta può essere soddisfatta in modo relativamente semplice, creando una struttura circolare.

Anche la regola che riguarda le definizioni locali deve essere opportunamente aggiornata per gestire correttamente la ricorsione:

$$[\text{LDEF}] \frac{\begin{array}{l} \mathcal{E}[x_1/v_1] \dots [x_n/v_n] \vdash E_i \rightarrow v_i, \forall i \in \{1, \dots, n\} \\ \mathcal{E}[x_1/v_1] \dots [x_n/v_n] \vdash E \rightarrow v \end{array}}{\mathcal{E} \vdash (\text{local } (\text{define } x_1 E_1) \dots (\text{define } x_n E_n) E) \rightarrow v}$$

Si noti che la valutazione di ogni singola E_i necessita di *tutti* i v_i affinché le definizioni possano essere mutuamente ricorsive.

2.5 Operatori primitivi

Volendo si potrebbero aggiungere regole di valutazione per tutti gli operatori primitivi di MiniScheme. Per esempio, si potrebbero aggiungere le regole per l'operatore +

$$\frac{\mathcal{E} \vdash E_i \rightarrow v_i, \forall i \in \{1, \dots, n\}}{\mathcal{E} \vdash (+ E_1 \dots E_n) \rightarrow \sum_{i=1}^n v_i}$$

precisando che tutti i valori v_i devono essere numeri interi. Ma questo non sarebbe sufficiente, perché dovremmo anche specificare cosa fare di un operatore primitivo quando questo non è applicato ad alcun argomento. Dovendo trattare gli operatori primitivi al pari delle funzioni, dovremmo avere

$$\overline{\mathcal{E} \vdash + \rightarrow \text{Closure}(\mathcal{E}, \langle ?_1, \dots \rangle, ?)}$$

ma saremmo in difficoltà nel dire quali e quanti argomenti $+$ richiede (si ricordi infatti che $+$ è un operatore n -ario in Scheme e anche in MiniScheme) e soprattutto non sapremmo mostrare il corpo della funzione che $+$ rappresenta.

Supporremo quindi che gli operatori primitivi siano in realtà nomi come tutti gli altri, e che ogni programma MiniScheme verrà valutato in un ambiente iniziale \mathcal{E}_0 che contiene già i legami tra i nomi degli operatori primitivi e opportune chiusure, non meglio specificate, che ne implementano la semantica. In questo modo, la regola per l'applicazione delle funzioni si applica anche nel caso degli operatori primitivi. A livello di implementazione, si dovranno fornire opportune chiusure *ad-hoc* per ciascun operatore primitivo richiesto. L'elenco degli operatori primitivi da implementare è dato in Tabella 2. Per una descrizione più dettagliata del loro comportamento, fare riferimento a [2].

Notare che **and** e **or** non possono essere implementati come operatori primitivi, in quanto essi non richiedono necessariamente la valutazione di *tutti* i loro argomenti, ma solo di quelli strettamente necessari per determinare il risultato. Per esempio la valutazione di

(**and** #f E)

non necessita di valutare E in quanto, per la semantica di **and**, il risultato è già determinato guardando il primo argomento (**#f**). L'osservazione duale vale per l'**or**.

3 Implementazione

Definire classi Java per rappresentare valori, espressioni, definizioni e ambienti.

3.1 Valori

L'interprete MiniScheme deve essere in grado di riconoscere i seguenti tipi di valori: *booleani, nu-*

meri interi, coppie, chiusure. Ogni valore deve avere una corrispondente rappresentazione in Java, l'interfaccia comune che ogni valore deve esporre è **SchemeValue** ed i suoi metodi hanno il seguente significato:

isT restituisce **true** se il valore rappresentato ha tipo T ;

asT restituisce il tipo specifico T e solleva l'eccezione **SchemeException** se il valore rappresentato ha un tipo diverso da T (per esempio, invocare il metodo **asBool** su un oggetto di tipo **SchemeValue** che rappresenta un valore di tipo intero causa l'eccezione);

applyTo applica la funzione (o l'operatore primitivo) rappresentato dal valore agli argomenti dati in un contenitore di tipo **List**. Se il valore non è una chiusura (o un operatore primitivo), solleva un'eccezione **SchemeException**;

toString converte il valore nella sua rappresentazione testuale. Per esempio, l'oggetto Java che rappresenta il valore booleano "true" in Scheme risponde con la stringa **#t** al metodo **toString**.

La coppia è l'unità elementare per costruire liste di valori Scheme. La lista (**list** 1 2 3) deve essere internamente rappresentata come una coppia con due campi, in cui il primo campo è un oggetto che rappresenta il valore 1 ed il secondo campo è la rappresentazione della lista (**list** 2 3). Decidere una rappresentazione adeguata per la lista vuota. I valori che rappresentano coppie devono implementare l'interfaccia **SchemePairValue**, i cui metodi **car** e **cdr** consentono di accedere al primo ed al secondo campo rispettivamente.

3.2 Espressioni

Così come per i valori, anche le espressioni espongono un'interfaccia comune **SchemeExpression** con un solo metodo pubblico, **evaluate**, che prende in ingresso un oggetto che rappresenta l'ambiente corrente e risponde con il risultato della valutazione dell'espressione in quell'ambiente.

Definire un insieme opportuno di classi Java per rappresentare le espressioni di MiniScheme (Tabella 1).

Tabella 2: Elenco degli operatori primitivi da implementare in MiniScheme.

Operatore primitivo		Descrizione
<code>(cons v₁ v₂)</code>		crea una coppia
<code>(list v₁ ... v_n)</code>	$n \geq 0$	crea una lista
<code>(car v)</code>		prima componente di una coppia
<code>(cdr v)</code>		seconda componente di una coppia
<code>(null? v)</code>		#t se v è la lista vuota
<code>(bool? v)</code>		#t se v è un valore booleano
<code>(int? v)</code>		#t se v è un valore intero
<code>(pair? v)</code>		#t se v è una coppia
<code>(eqv? v₁ v₂)</code>		uguaglianza fisica
<code>(+ v₁ ... v_n)</code>	$n \geq 0$	somma intera n -aria
<code>(* v₁ ... v_n)</code>	$n \geq 0$	moltiplicazione intera n -aria
<code>(- v₁ ... v_n)</code>	$n \geq 1$	negazione, sottrazione
<code>(/ v₁ ... v_n)</code>	$n \geq 1$	reciproco, divisione
<code>(= v₁ ... v_n)</code>	$n \geq 0$	“uguale a” tra interi
<code>(< v₁ ... v_n)</code>	$n \geq 0$	“minore di” tra interi
<code>(> v₁ ... v_n)</code>	$n \geq 0$	“maggiore di” tra interi
<code>(<= v₁ ... v_n)</code>	$n \geq 0$	“minore o uguale a” tra interi
<code>(>= v₁ ... v_n)</code>	$n \geq 0$	“maggiore o uguale a” tra interi

3.3 Definizioni

Una definizione rappresenta un legame tra un nome x ed un’espressione E stabilito da un costrutto

```
(define x E)
```

indifferentemente che questa definizione sia globale o locale.

Definire una classe che implementi l’interfaccia `SchemeDefinition`. Il metodo `declare` deve modificare l’ambiente passato come argomento in modo che il nome x diventi visibile, pur non avendo ancora un valore associato. Il metodo `define` deve completare la dichiarazione di un nome x associandogli il valore dell’espressione E , valutata nell’ambiente passato come parametro.

La distinzione tra dichiarazione e definizione di un’associazione nome/valore deve consentire di implementare correttamente le funzioni ricorsive.

3.4 Ambiente

Fornire una classe che implementi l’interfaccia `SchemeEnvironment` e che serva a rappresentare ambienti, ovvero insiemi di associazioni nome/valore. Esistono vari modi per implementare ambienti, ma ogni implementazione deve fornire i seguenti metodi:

`add` aggiunge all’ambiente una nuova associazione nome/valore;

`get` ritorna il valore associato al nome passato come argomento, o solleva l’eccezione `SchemeException` se non vi è alcuna associazione per quel nome;

`set` modifica il valore associato ad un nome;

`copy` ritorna una copia dell’ambiente. Eventuali modifiche ai nomi nella copia dell’ambiente non devono avere effetto sull’ambiente originale che è stato copiato. In base alle regole di valutazione date nella sezione 2, determinare tutti i punti in cui è necessario effettuare una copia dell’ambiente.

Per verificare se l’ambiente è stato implementato correttamente, è possibile usare il seguente frammento di codice

```
SchemeEnvironment env = new MySchemeEnv();
env.add("x", null);
SchemeEnvironment copyEnv = env.copy();
env.add("y", null);

// deve sollevare eccezione
copyEnv.get("y");

copyEnv.set("x", new MySchemeValue(...));
// deve restituire true
env.get("x") == copyEnv.get("x")
```

3.5 Analisi lessicale e sintattica

Vengono fornite due implementazioni per le interfacce `SchemeScanner` e `SchemeParser` chiamate `SchemeScannerImpl` e `SchemeParserImpl` rispettivamente.

L'interfaccia pubblica dello scanner non è particolarmente utile ai fini dell'implementazione del progetto, in quanto solo il parser ne ha bisogno. L'unico metodo interessante è `getToken`: quando tale metodo restituisce il valore `SchemeScanner.EOF` significa che lo stream di input è terminato (end of file). Per creare un'istanza della classe `SchemeScannerImpl` è necessario fornire al costruttore il nome dello stream da cui si vuole leggere il programma `MiniScheme` da interpretare e lo stream stesso (interfaccia `InputStream`). Per convenzione, usare `<stdin>` come nome quando si usa `System.in` come stream di input.

L'interfaccia `SchemeParser` espone un solo metodo, `parseDefinition`, che causa il riconoscimento della successiva definizione globale nello stream di input. Per creare un'istanza della classe `SchemeParserImpl` è necessario fornire al costruttore un'istanza dello scanner che si intende utilizzare ed un'istanza della factory per le espressioni (vedi Sezione 3.6).

3.6 Factory

Man mano che le definizioni e le espressioni `MiniScheme` vengono riconosciute dal parser, questo richiede la creazione di opportuni oggetti di interfaccia `SchemeDefinition` (per le definizioni) e `SchemeExpression` (per le espressioni). Fornire una implementazione di `SchemeFactory` che istanzi la classe corretta in base al tipo di espressione incontrata dal parser.

Si ricordi che la definizione

```
(define (f  $x_1 \dots x_n$ ) E)
```

è solo un'abbreviazione per

```
(define f (lambda ( $x_1 \dots x_n$ ) E))
```

per cui la factory prevede solo quest'ultimo tipo di definizione ed il parser provvede ad espandere adeguatamente ogni definizione del primo tipo in una del secondo tipo con un opportuno `lambda`.

3.7 Operatori primitivi

Implementare gli operatori primitivi come chiusure *ad-hoc*, cioè valori che rispondono al me-

todo `applyTo`. A differenza delle chiusure di funzioni, gli operatori primitivi possono accettare un numero variabile di argomenti. Sfruttare questa caratteristica per implementare operatori primitivi il più possibile fedeli a quelli di un vero interprete `Scheme`.

4 Funzioni di libreria

Come verifica del buon funzionamento dell'interprete, usando gli operatori primitivi della Tabella 2 implementare in `MiniScheme` le seguenti funzioni della libreria standard `Scheme` e verificarne il corretto comportamento: `not`, `equal?`, `quotient`, `remainder`, `zero?`, `positive?`, `negative?`, `max`, `min`, `even?`, `odd?`, `abs`, `list?`, `length`, `append`, `reverse`. Fare riferimento a [2] per una definizione precisa del comportamento voluto per queste funzioni e limitarle in accordo con la vostra implementazione di `MiniScheme` (non implementare i comportamenti relativi a tipi di dato non gestiti dalla vostra implementazione, fissare il numero di argomenti per quelle funzioni che ne prevedono un numero variabile, ecc.)

Fare in modo che questo insieme di funzioni sia memorizzato in un file `init.scm` caricato automaticamente dal vostro interprete prima di cominciare a valutare definizioni dallo standard input.

Riferimenti bibliografici

- [1] Scheme, <http://www.swiss.ai.mit.edu/projects/scheme/>
- [2] Richard Kelsey, William Clinger, and Jonathan Rees (Editors), "Revised(5) Report on the Algorithmic Language Scheme", <http://www.schemers.org/Documents/Standards/R5RS/r5rs.pdf>, <http://www.swiss.ai.mit.edu/ftplibdir/scheme-reports/r5rs.ps.gz>, http://www.swiss.ai.mit.edu/~jaffer/r5rs_toc.html

A Interfacce date

```
----- SchemeValue.java -----
import java.util.List;

interface SchemeValue {
    public boolean isBool();
    public boolean isInt();
    public boolean isString();
    public boolean isSymbol();
    public boolean isPair();
    public boolean asBool() throws SchemeException;
    public int asInt() throws SchemeException;
    public String asString() throws SchemeException;
    public String asSymbol() throws SchemeException;
    public SchemePairValue asPair() throws SchemeException;
    public SchemeValue applyTo(List args) throws SchemeException;
    public String toString();
}
-----
```

```
----- SchemePairValue.java -----
interface SchemePairValue extends SchemeValue {
    public SchemeValue car();
    public SchemeValue cdr();
}
-----
```

```
----- SchemeDefinition.java -----
interface SchemeDefinition {
    public void declare(SchemeEnvironment env);
    public void define(SchemeEnvironment env) throws SchemeException;
}
-----
```

```
----- SchemeExpression.java -----
interface SchemeExpression {
    public SchemeValue evaluate(SchemeEnvironment env) throws SchemeException;
}
-----
```

```
----- SchemeBranch.java -----
interface SchemeBranch {
    public SchemeExpression getTest();
    public SchemeExpression getBody();
}
-----
```

```
----- SchemeFactory.java -----
import java.util.List;

interface SchemeFactory {
    public SchemeDefinition createDefinition(String name, SchemeExpression expr);
    public SchemeBranch createBranch(SchemeExpression test, SchemeExpression e);
    public SchemeExpression createApplyExpression(List exprs);
    public SchemeExpression createLambdaExpression(List params, SchemeExpression expr);
    public SchemeExpression createAndExpression(List exprs);
    public SchemeExpression createOrExpression(List exprs);
    public SchemeExpression createCondExpression(List branches, SchemeExpression e);
    public SchemeExpression createLocalExpression(List bindings, SchemeExpression e);
    public SchemeExpression createIdExpression(String id);
    public SchemeExpression createBoolExpression(boolean v);
    public SchemeExpression createIntExpression(int v);
    public SchemeExpression createStringExpression(String v);
    public SchemeExpression createSymbolExpression(String v);
}
-----
```

```

SchemeEnvironment.java
interface SchemeEnvironment {
    public SchemeEnvironment copy();
    public void add(String name, SchemeValue value);
    public SchemeValue get(String name) throws SchemeException;
    public void set(String name, SchemeValue value) throws SchemeException;
}

```

```

SchemeScanner.java
public interface SchemeScanner {
    static public final int EOF = 0;
    static public final int BOOL = 1;
    static public final int INT = 2;
    static public final int STRING = 3;
    static public final int SYMBOL = 4;
    static public final int OPEN = 10;
    static public final int CLOSE = 11;
    static public final int ID = 12;

    public boolean more() throws java.io.IOException;
    public void nextToken() throws java.io.IOException;
    public int getToken();
    public String getValue();
    public String getSourceName();
    public int getLine();
}

```

```

SchemeParser.java
public interface SchemeParser {
    public SchemeDefinition parseDefine() throws java.io.IOException;
}

```

B Classi date

```

SchemeException.java
class SchemeException extends Exception {
    public SchemeException(String msg) {
        super(msg);
    }
}

```

```

SchemeSyntaxError.java
class SchemeSyntaxError extends Error {
    public SchemeSyntaxError(String sourceName, int line, String msg) {
        super(sourceName + ":" + Integer.toString(line) + ": " + msg);
    }
}

```

```

SchemeScannerImpl.java
public class SchemeScannerImpl implements SchemeScanner {
    ...
}

```

```

SchemeParserImpl.java
public class SchemeParserImpl implements SchemeParser {
    ...
}

```