Progetto interprete Scheme

Roberto Lucchi

Linguaggi di Programmazione 2005/06

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)))))))
```


Esempio funzione fact

- In base alla regola [GDEF1] l'ambiente che si ottiene è:

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

- Dove:

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

Esempio funzione fact

- Applicato (fact n) finiamo per valutare E nell'ambiente

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

- La valutazione di E potrebbe (se x>0) avere necessità di risolvere il nome fact al momento della chiamata ricorsiva
- Tuttavia, l'ambiente in cui questa valutazione avviene è l'ambiente di partenza in cui il valore dell'argomento x è correttamente legato, mentre il nome fact è sconosciuto!

Il problema di GDEF1

- 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

Le regola corretta per le definizioni globali

- In formule, la regola per le definizioni globali dovrebbe essere:

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

- Nota: nella premessa della regola, la valutazione di E viene svolta in un ambiente dove il valore resituito da E è già noto. Vedremo nella parte di implementazione che questa richiesta può essere soddisfatta in modo relativamente semplice, creando una struttura circolare.

Regola LDEF

- Anche la regola LDEF1 va rivista per gestire opportunamente la ricorsione:

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

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

Operatori primitivi

- Volendo si potrebbero aggiungere regole di valutazione per tutti gli operatori primitivi di MiniScheme, ad esempio

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

Operatori primitivi

- Ma questo non sarebbe sufficiente, perchè 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 anche la rispettiva regola di chiusura $\overline{\mathcal{E} \vdash \texttt{+} \rightarrow \texttt{Closure}(\mathcal{E}, \langle ?_1, \ldots \rangle, ?)}$
- 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

Operatori primitivi

- 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* 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.

Implementazione di operatori primitivi

- A livello di implementazione, si dovranno fornire opportune chiusure *ad-hoc* per ciascun operatore primitivo richiesto
- Per la descrizione dettagliata dei vari operatori di Scheme si veda:
  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

Operatori primitivi di MiniScheme da implementare

Operatore primitivo		Descrizione
(cons v_1 v_2)		crea una coppia
(list v_1 $\cdots$ v_n)	$n \geq 0$	crea una lista
(car v)		prima componente di una coppia
(cdr v)		seconda componente di una coppia
(null? v)		#t se v è la lista vuota
(bool? v)		#t se v è un valore booleano
(int? v)		#t se v è un valore intero
(pair? v)		#t se v è una coppia
(eqv? v_1 v_2)		uguaglianza fisica

Operatori primitivi di MiniScheme da implementare

Operatore primitivo		Descrizione
(+ $v_1 \cdots v_n$)	$n \geq 0$	somma intera n-aria
(* $v_1 \cdots v_n$)	$n \geq 0$	moltiplicazione intera n-aria
(- $v_1 \cdots v_n$)	$n \geq 1$	negazione, sottrazione
(/ $v_1 \cdots v_n$)	$n \geq 1$	reciproco, divisione
(= $v_1 \cdots v_n$)	$n \geq 0$	"uguale a" tra interi
(< $v_1 \cdots v_n$)	$n \geq 0$	"minore di" tra interi
(> $v_1 \cdots v_n$)	$n \geq 0$	"maggiore di" tra interi
(<= $v_1 \cdots v_n$)	$n \geq 0$	"minore o uguale a" tra interi
(>= $v_1 \cdots v_n$)	$n \geq 0$	"maggiore o uguale a" tra interi

Operatori and e or

- 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.
- Esempio: (and #f E) non richiede la valutazione di E in quanto, per la semantica di and, il risultato è già determinato dopo la valutazione di #f.
- L'osservazione duale vale per l'or.

COME IMPLEMENTARE?

Implementazione

- Vedremo come vanno definite le classi per la rappresentazione di
  - Valori
  - Espressioni
  - Ambienti

Valori

- L'interprete MiniScheme deve essere in grado di riconoscere i seguenti tipi di valori:
  - *Booleani*
  - *Numeri interi*
  - *Coppie*
  - *Chiusure*

L'interfaccia SchemeValue

- 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 (e.g. isBool per i booleani);

L'interfaccia SchemeValue

- 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) che l'oggetto rappresenta agli argomenti dati in un contenitore di tipo List. Se il valore rappresentato dall'oggetto non è una chiusura (o un operatore primitivo), solleva un'eccezione SchemeException;

L'interfaccia SchemeValue

- 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

Rappresentazione di liste

- 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 *coda* della lista, in questo caso (list 2 3).
- Decidere una rappresentazione adeguata per la lista vuota.

L'intefaccia SchemePairValue

- 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.
  - L'interfaccia estende SchemeValue

Interfaccia SchemeExpression

- Così come per i valori, anche le espressioni espongono un'interfaccia comune SchemeExpression con un solo metodo pubblico:
  - Evaluate
    - dato come parametro di input un oggetto che rappresenta l'ambiente corrente, risponde con il risultato della valutazione dell'espressione in quell'ambiente.

Espressioni

- Compito del progetto è definire un insieme di classi opportune per la rappresentazione delle espressioni MiniScheme (si veda la grammatica MiniScheme)

Definizioni

- Una definizione rappresenta un legame tra un nome x ed un'espressione E stabilito dal costrutto (indifferentemente che questa definizione sia globale o locale):

  (define x E)

Interfaccia SchemeDefinition

- Definire una classe che implementa l'interfaccia SchemeDefinition che ha i seguenti metodi (si assume che l'oggetto rappresenti l'associazione di x a E)
  - declare
    - deve modificare l'ambiente passato come argomento in modo che il nome x diventi *visibile*, pur non avendo ancora un valore associato
  - define
    - deve completare la dichiarazione di un nome x associandogli il valore dell'espressione E, valutata nell'ambiente passato come parametro

Interfaccia SchemeDefinition

- La distinzione tra dichiarazione e definizione di un'associazione nome/valore deve consentire di implementare correttamente le funzioni ricorsive
  - si ricordi [GDEF] in caso di def. ricorsive

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 implementare i metodi dell'interfaccia

Interfaccia SchemeEnvironment

- 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 viste determinare tutti i punti in cui è necessario effettuare una copia dell'ambiente.
    - gli oggetti contenuti nei due ambienti (originale e copiato) sono effettivamente condivisi, per cui una set su un ambiente E si "trasmette" automaticamente anche in tutti gli ambienti ottenuti copiando E.

Test di verifica implementazione

- 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")
```


Analisi lessicale e sintattica

- Analisi lessicale
  - Scanner
    - Complessità O(n)
    - Automi a stati finiti (grammatiche regolari)
- Analisi sintattica
  - Parser
    - Complessità $O(n^3)$
    - Automi a pila (linguaggi liberi)

Scanner e parser

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

SchemeScannerImpl

- 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 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 costrutture 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.

SchemeParserImpl

- 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
  - un'istanza della factory per le espressioni

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
  - SchemeExpression per le espressioni

L'interfaccia SchemeFactory

- 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 x1 · · · xn) E)
  è un'abbreviazione di
  (define f (lambda (x1 · · · xn) E)
- Quindi la factory prevede solo quest'ultimo tipo di definizione
  - il parser messo a vostra disposizione provvede ad espandere adeguatamente ogni definizione del primo tipo in una del secondo tipo con un opportuno lambda.

Operatori primitivi

- Implementare gli operatori primitivi come chiusure *ad-hoc*
  - valori che rispondono al metodo 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

Funzioni di libreria da implementare in MiniScheme:

- Come verifica del buon funzionamento dell'interprete, usando gli operatori primitivi che supporta MiniScheme le seguenti funzioni della libreria standard Scheme e verificarne il corretto comportamento:
  - not, equal?, quotient, reminder, zero?, positive?, negative?, max, min, even?, odd?, abs, list?, length, append, reverse
  - si veda la documentazione Scheme sull'esatto comportamento di tali funzioni
  - Limitate la codifica in accordo a MiniScheme, cioè ad esempio non implementate i comportamenti relativi a tipi di dato non gestiti dalla vostra implementazione, fissate il numero di argomenti per quelle funzioni che ne prevedono un numero variabile, ecc.

Nel progetto ...

- Questo insieme di funzioni, memorizzato in un file init.scm, deve essere caricato automaticamente dal vostro interprete prima di cominciare a valutare definizioni dallo standard input
  - In altre parole, il programma interpretato potrà anche utilizzare tali funzioni

Interfaccie date: SchemeValue

```
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();
}
```

Interfaccie date

SchemePairValue

```
interface SchemePairValue extends SchemeValue {
    public SchemeValue car() throws SchemeException;
    public SchemeValue cdr() throws SchemeException;
}
```

SchemeDefinition

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


Interfaccie date

SchemeExpression

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

SchemeBranch

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


Interfaccie date

SchemeFactory

```
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);
}
```


Interfaccie SchemeEnvironment e SchemeScanner

```
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;
}

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();
}
```

Interfaccie SchemeParser

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


Classi date

SchemeException

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

SchemeSyntaxError

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


Classi date

- SchemeScannerImpl
- SchemeParserImpl
- che implementano le interfaccie SchemeScanner e SchemeParser

Appendice: riepilogo regole

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

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

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

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

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

$$[\textsc{Lam}]\ \frac{}{\begin{array}{l}\mathcal{E} \vdash \texttt{(lambda } (x_1\ \cdots\ x_n)\ E\texttt{)}\\ \quad\to \texttt{Closure}(\mathcal{E}, \langle x_1,\ldots,x_n\rangle, E)\end{array}}$$

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

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

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